1. Grafos

Grafos

```
struct amGraph{                 // Adjency Matrix Graph
  int n, edge[MAX][MAX] ;
  void init(int s, int val = INF ) { n = s; FOR(i,n) FOR(j,n) edge[i][j] = val; }
};
struct alGraph{
  int n, nedge[MAX], edge[MAX][MAX], cost[MAX][MAX], prev[MAX] , low[MAX] , ordem[MAX];
  bool ap[MAX], mst[MAX][MAX];

  void init(int s)              { n = s; FOR(i,s) nedge[i] = 0; }
  void addEdge(int src, int dst, int c = 0) { // Com Dinic, o cost é matriz de adjacencia!!
    edge[src][nedge[src]] = dst; cost[src][nedge[src]++] = c; }
};
#define COST  first
#define V1   second.first
#define V2   second.second
struct elGraph{                 // Edge List Graph
  int ne , n , mst[MAXE];
  pair < int, PII > edge[MAXE];
  void init (int s)          { n = s;  ne = 0;  Set(mst , 0 );  }
  void addEdge(int a, int b, int c)  { edge[ne++]= make_pair(c,PII(a, b)); }
};
```

Propriedades de Grafos

- A graph is bipartite if and only if it does not contain an odd cycle. Therefore, a bipartite graph cannot contain a clique of size 3 or more.
- A graph is bipartite if and only if it is 2-colorable, (i.e. its chromatic number is less than or equal to 2).
- The size of minimum vertex cover is equal to the size of the maximum matching (König's theorem).
- The size of the maximum independent set plus the size of the maximum matching is equal to the number of vertices.
- For a connected bipartite graph the size of the minimum edge cover is equal to the size of the maximum independent set.
- For a connected bipartite graph the size of the minimum edge cover plus the size of the minimum vertex cover is equal to the number of vertices.
- To extend to the case of undirected graphs, simple expand the edge as two arcs in opposite directions.
- **If you want to limit the amount of traffic through any node, split each node into two nodes, an in-node and and out-node. Put all the in-arcs into the in-node, and all of the out-arcs out of the out-node and place an arc from the in-node to the out-node with capacity equal to the capacity of the node.**
- If you have multiple sources and sinks, create a `virtual source' and `virtual sink' with arcs from the virtual source to each of the sources and arcs from each of the sinks to the virtual sink. Make each of the added arcs have infinite capacity.
- If you have arcs with real-valued weights, then this algorithm is no longer guaranteed to terminate, although it will asymptotically approach the maximum.
- **Given a weight undirected graph, what is the set of edges with minimum total weight such that it separates two given nodes. The minimum total weight is exactly the flow between those two nodes. This can be extended to node cuts by the same trick as nodes with limited capacity.**
- Dados dois conjuntos de nós com ligações entre os 2 conjuntos, mas não dentro do conjunto, descobrir o maior sub-conjunto de arestas que ligue cada nó de um conjunto a outro (só um) do outro conjunto. Resolução: Criar uma source e sink virtuais, modelar todas as arestas com peso 1, ligar todos os nós de um dos conjuntos à source e o outro à sink. Correr um algoritmo de maximum flow. **É equivalente ao minimum vertex cover em grafos bipartidos.**
- Para resolver problemas como: Temos uma grelha 2D com clones nos pontos inteiros. Podemos destruir todos os clones numa linha ou coluna num único tiro. Qual o número mínimo de tiros necessário para destruir todos os clones? Criamos um grafo bi-partido com as linhas de um lado e as colunas de um outro, com um arco entre uma dada linha e uma coluna se existir um clone nesse ponto, e achamos o emparelhamento máximo (ponto anterior).

Kruskal

Calcula uma Minimum Spanning Tree usando UnionFind e um grafo representado como uma lista de arestas. O algoritmo poe a true a variavel mst nas arestas que fazem parte da MinimumSpanningTree.

```
int elGraph::kruskal(){
 init_uf(n);        // inicializa o union find
 sort(edge, edge + ne);
 int res = 0;
 FOR(i,ne)
  if( find_set(edge[i].V1) != find_set(edge[i].V2)) {
   union_set(edge[i].V1, edge[i].V2);
   mst[i] = true;
   res += edge[i].COST;
  }
 return res;
}
```

Bellman-Ford

Calcula a distância mínima entre um nodo e todos os outros mesmo quando existem arestas com peso negativo. O algoritmo retorna false se houver ciclos negativos.

```
bool elGraph::bellmanford( int source, int dist[], int prev[]) {
  FOR(i,n)  dist[i] = INF;
  dist[source] = 0; int d;
  FOR(i,n) FOR(j,ne)  if (dist[edge[j].V1] != INF) {
    if ( (d = dist[edge[j].V1] + edge[j].COST) < dist[edge[j].V2])
      dist[edge[j].V2] = d , prev[edge[j].V2] = j;
  }
  FOR(i,ne)  if (dist[edge[i].V2] > dist[edge[i].V1] + edge[i].COST)
          return false;
  return true;
}
```

Dinic

Determina o fluxo máximo entre 2 pontos de um grafo em O(N^2 * E). **Nota**: "cost" é matriz de adjacencias com as capacidades, enquanto que "edge" é lista de adjacências.

```
int alGraph::maxFlowDinic( int s, int t ) {
  for ( int flow = 0 , bot , i , u , v , qf , qb , q[MAX];  true ; ) {
    Set( prev , 0 );
    qf = qb = 0 , prev[q[qb++] = s] = -2;
    while( qb > qf && prev[t] == -1 )
      for( u = q[qf++], i = 0; i < nedge[u]; i++ )
        if( prev[v = edge[u][i]] == -1 && cost[u][v] )
          prev[q[qb++] = v] = u;
    if( prev[t] == -1 ) return flow;
```

```
    FOR(z,n)  if( cost[z][t] && prev[z] != -1 ) {
      bot = cost[z][t];
      for( v = z, u = prev[v]; u >= 0; v = u, u = prev[v] )
        bot = min( bot , cost[u][v] );
      if( !bot ) continue;

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

Minimum Cost Maximum Flow

```
#define MAX 610
set<pair<int,int> > h;
int cap[MAX][MAX],cost[MAX][MAX], fnet[MAX][MAX], adj[MAX][MAX], deg[MAX],pi[MAX] ,par[MAX],d[MAX];

#define Pot(u,v) (d[u] + pi[u] - pi[v])
inline void add( int v , int dis , int d_old ) {
  pair<int,int> p(dis,v) , old( d_old , v);  h.erase(old);    h.insert( p );
}
bool dijkstra( int n, int s, int t ) {
  Set( d, 0x3F ) ;  Set( par, -1 );  h.clear(); par[s] = n ; d[s] = 0; add( s , 0 , 0 );

  while( h.size() ) {
    int old, u = h.begin()->second;    h.erase(h.begin());

    for( int k = 0, v = adj[u][k]  ;  old = d[v] , k < deg[u] ; v = adj[u][++k] ){
      if( fnet[v][u] && d[v] > Pot(u,v) - cost[v][u] )
        d[v] = Pot(u,v) - cost[v][par[v] = u];
      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 )
        add( v , d[v], old );
    } }
  for( int i = 0; i < n; i++ )  if( pi[i] < INF )  pi[i] += d[i];
  return par[t] >= 0;
}
int mcmf( int n, int s, int t, int &fcost ) {
  Set( deg, 0 );  Set( fnet, 0 ); Set( pi, 0 );
  int i , j , bot , u , v , flow = fcost = 0;
  FOR(i,n) FOR(j,n) if( cap[i][j] || cap[j][i] ) adj[i][deg[i]++] = j;

  for( ; dijkstra( n, s, t ) ; flow += bot ) {
    for( bot = INF , v = t, u = par[v]; v != s; u = par[v = u] )
      bot = min( bot , fnet[v][u] ? fnet[v][u] : ( cap[u][v] - fnet[u][v] ));

    for( 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]; }
  }
  return flow;
}
```

```
int main() {
  int N,E, i , j , k , p , casos , sink , source , fcost , flow ;

  scanf("%d",& casos);
  while (casos--)  {
    scanf("%d %d %d", & N , & E , & p );
    source = N+E , sink = source+1;
    Set( cap, 0 );
    while (p--){
      scanf("%d %d %d" , & i, & j , & k );
      cap[i][N+j] = 1 ,   cost[i][N+j] = k;
    }
    for (i = 0 ; i < N ; i++)  cap[source][i] = 1 ,   cost[source][i] = 0;
    for (i = N ; i < source ; i++)  cap[i][sink] = 1 ,   cost[i][sink] = 0;
    flow = mcmf( sink+1, source, sink, fcost );
    printf( "%d\n\n" , fcost );
  }
  return 0; }
```

Minimum Cut de um Grafo

Calcula o custo mínimo das arestas que é preciso remover de forma a separar dois nodos do grafo. Esta implementação usa o algoritmo de

```
#define MAXW 1000       // Maximum edge weight (MAXW * NN * NN must fit into an int)
int amGraph::minCutAllGraph() {
  int i, j, best = MAXW * n * n, prev, x, v[MAX], w[MAX], na[MAX], a[MAX];
  FOR(i,n) v[i] = i;
  while( n > 1 )  {
    for( a[v[0]] = i = 1; i < n; i++ )
      a[v[i]] = 0 , na[i - 1] = i , w[i] = edge[ v[0] ][ v[i] ];
    for( prev = v[0] , i = 1; i < n; i++ )  {
      for( x = -1 , j = 1; j < n; j++ )
        if( !a[v[j]] && ( x < 0 || w[j] > w[x] ) )
          x = j;
      a[v[x]] = 1;
      if( i == n - 1 ) {
        best = min( best , w[x] );
```

```
        for( j = 0; j < n; j++ )
          edge[v[j]][prev] = edge[prev][v[j]] += edge[v[x]][v[j]];
        v[x] = v[--n];
        break;
      }
      for( prev = v[x] , j = 1; j < n; j++ ) if( !a[v[j]] )
        w[j] += edge[v[x]][v[j]];
    } }
  return best; }
```

```
amGraph g;
int main() {
  int C , nc , N , E , i , j, k;
  scanf("%d" , &C);
  for (nc = 1 ; nc <= C ; nc++) {
    scanf("%d %d" , &N , &E );
    g.init(N,0);
    while (E-- ) {
      scanf("%d %d %d\n" , &i , &j , &k);
      --i , --j;
      g.addEdge( i , j , k );
      g.addEdge( j , i , k );
    }
    printf("Case #%d: %d\n", nc, g.minCut() );
  }
  return 0;
}
```

```
const int M = 128 , N = 128; // grafo bipartido tem dimensão máxima MxN
int matchL[M], matchR[N] , n, m, graph[M][N] , vis[N];

bool bpm_dfs( int u ) {
  FOR (v,n)  if( graph[u][v] && !vis[v] ) {
          vis[v] = true;
          if( matchR[v] < 0 || bpm_dfs( matchR[v] ) ) {
            matchL[u] = v, matchR[v] = u;
            return true;
          }
  }
  return false;
}
int bpm() {
  Set( matchL , -1 ); Set( matchR , -1 );
  int res = 0;
  for( int i = 0; i < m; i++ ) {
    Set( vis , 0 );
    if( bpm_dfs( i ) ) res++;
  }
  // cnt contains the number of happy pigeons
  // matchL[i] contains the hole of pigeon i or -1 if pigeon i is unhappy,
  // matchR[j] contains the pigeon in hole j or -1 if hole j is empty
  return res;
}
```

Maximum Matching Unweighted Generic Graphs

Algoritmo de Edmonds O(N^4) para encontrar o emparelhamento maximo em grafos genéricos sem pesos. Recebe a matriz de adjacencias.

```
int N;        // Numero de vertices
VI mat, vis;  // match[] e visited
#define couple(a,b) mat[a] = b , mat[b] = a
bool dfs(int n, VVI &a, VI &b) {
  vis[n]=0;
  FOR(i, N) if(a[n][i]) {
    if(vis[i]==-1) {
      vis[i]=1;
      if(mat[i]== -1 || dfs(mat[i],a,b)) {couple(n,i);return true;}
    }
    if(vis[i]==0 || b.size()) {   // found flower
      b.PB(i); b.PB(n);
      if(n==b[0]) { mat[n]=-1; return true; }
      return false;
    }
  }
  return false;
}
bool augment(VVI &a) {
  FOR(m, N) if(mat[m]==-1) {
    VI b;
    vis=VI(N,-1);
    if(!dfs(m, a, b))  continue;
    if(! b.size() )   return true;   // augmenting path found

    int n , op=b[0], S=b.size();
    VVI w=a;
    REP(i, 1, S-1) FOR(j, N) w[op][j]=w[j][op]|=a[b[i]][j];
    REP(i, 1, S-1) FOR(j, N) w[b[i]][j]=w[j][b[i]]=0;
    w[op][op]=0;
    if(!augment(w)) return false;

    if( (n=mat[op]) !=-1) FOR(i, S) if(a[b[i]][n]) {
      couple(b[i], n);
      if(i&1) for(int j=i+1; j<S; j+=2) couple(b[j],b[j+1]);
      else   for(int j=0 ; j<i; j+=2) couple(b[j],b[j+1]);
```

```
      break;
    }
    return true;
  }
  return false;
}
int edmonds(VVI &a) {  // matriz de adjacencia
  int res=0;
  mat=VI(N,-1);
  while(augment(a)) res++;
  return res;
}
```

```
VVI conn;
void add(int i , int j ) {  conn[i][j] = conn[j][i] = 1;  }

int main() {
  N = 6;
  conn = conn = VVI(N, VI(N,0));
  add(0,1) ,add(0,2) ,add(1,2),add(2,3), add(0,4), add(1,5), add(4,5), add(3,4) , add(3,5);
  cout << edmonds(conn) << endl;
  return 0;
}
```

Articulation Points / Bridges / Biconnected Components

Procura os pontos e pontes de articulação num grafo não dirigido. Coloca o resultado numa variavel ap e as pontes no set pontes. Preferi colocar como métodos do struct alGraph para ser preciso escrever menos no código (senão ficava cheio de g.X)

```
set< pair<int,int> > pontes; // pontes guardadas num par i->j , com i < j

int alGraph::dfs_part_pontes(int v, int lastv , int & count) {
  low[v] = ordem[v] = ++count;
  int nfilhos = 0;
  for (int e = 0 , w = edge[v][e]; e < nedge[v]; w = edge[v][++e]){
    if ( ordem[w] == INF ) {
      nfilhos++;
      dfs_part_pontes(w, v , count);
      low[v] = min( low[v] , low[w] );
      if ( low[w] >= ordem[v] )  {
        ap[v] = true;
        if ( low[w] == ordem[w] )
          pontes.insert( make_pair( min(v,w) , max(v,w) ) );
      }
    }
    else if (lastv != w && ordem[w] < low[v]) low[v] = ordem[w];
  }
  return nfilhos;
}
int alGraph::particulacao_pontes() {
  int i , count = 0 , npontosart = 0;
  for ( i = 0 ; i < nvertex ; i++ )  low[i] = ordem[i] = INF , ap[i] = false;
  for ( i = 0 ; i < nvertex ; i++ )
    if ( ordem[i] == INF && dfs_part_pontes( i , -1 , count ) < 2)
      ap[i] = false;

  for ( i = 0 ; i < nvertex ; i++ ) // ver pontos de articulação
    npontosart += ap[i] ? 1 : 0;
  return npontosart;
}
```

```
alGraph g;
int main() {
  int v , i , j , k , na , N ;
  while ( scanf("%d", &N) != EOF ) {
    g.init(N);
    for ( i = 0 ; i < N ; i++ )  {
      scanf("%d (%d)", &j , &na);
      for (k = 0 ; k < na ; k++) {
        scanf("%d" , &v);
        g.addEdge( j , v );
      }
    }
    pontes.clear();
    g.particulacao_pontes();

    printf("%d critical links\n", pontes.size());
    for (set< pair<int,int> >::iterator it = pontes.begin(); it != pontes.end() ; it++)
      printf("%d - %d\n", it->first , it->second );
    printf("\n");
  }
  return 0;
}
```

Componentes Fortemente Conexas - Algoritmo de Tarjan

A directed graph is called strongly connected if for every pair of vertices u and v there is a path from u to v and a path from v to u. The strongly connected components (SCC) of a directed graph are its maximal strongly connected subgraphs. No array comp[] está o index da

```
int n , m , nc , root[MAX], comp[MAX], size[MAX], vis[MAX];
stack<int> s;
vector<int> adj[MAX];  // grafo

void tarjan(int v, int d) {
```

```
  root[v] = d;  vis[v]=1;  s.push(v);
  for ( size_t i = 0 ; i < adj[v].size() ; i++ ) {
    if ( ! vis[ adj[v][i] ] )  tarjan( adj[v][i] , d+1 );
    if ( comp[ adj[v][i] ] == 0 )  root[v] = min( root[v] , root[ adj[v][i] ] );
  }
  if (root[v]==d) {
    comp[v] = ++nc;
    for ( size[nc] = 1  ;    s.top() != v  ;  s.pop() , size[nc]++ )
      comp[ s.top() ] = nc;
    s.pop();
  }
}
```

```
int main() {
  int i, a, b, cand, res=0;
  scanf("%d %d", &n, &m);
  for (i=0; i<m; i++) {
    scanf("%d %d", &a, &b);
    adj[a].push_back(b);
  }
  for (i=1, nc=0; i<=n; i++) /* Find Strong Components */
    if (!vis[i])
      tarjan(i, 0);
  printf("%d\n", nc);
  return 0;
}
```

Stable Marriage Problem

Descobre a melhor forma de casar casais de forma a nenhum elemento nao casar com uma pessoa que preferisse e que também o preferisse a ele. Neste algoritmo tem de existir algum grupo que tem preferência: homens ou mulheres (escolhem primeiro), o grupo que tem preferência é o "optimal" e os outros other. É usado o algoritmo de Gale-Shapley - O(N^2).
Marriage Problem" do SEERC07 (male optimal) na UVA e "stablemp" no spoj.

```
#define MAX 1010
int N , other[MAX][MAX] , optimal_pair[MAX] , other_pair[MAX]; // se other_pair[i] < 0
queue<int> optimal[MAX];          // pessoa i nao tem par

void StableMarriage( ) {
  memset( other_pair , -1 , sizeof(other_pair));
  queue<int> optimal_free;
  for ( int i = 1 ; i <= N ; i++)  // adicionar pessoas do grupo a optimizar sem par
    optimal_free.push(i);
  while ( ! optimal_free.empty() )  // enquanto nao casamos toda a gente
  {
    int b = optimal_free.front();  optimal_free.pop();
    int m = optimal[b].front();  optimal[b].pop();
    if (other_pair[m] < 0 ) {
      other_pair[m] = b;  // match other "m" com optimal "b"
      optimal_pair[b] = m;
    }
    else if ( other[m][b] < other[m][other_pair[m]] ) {
        optimal_free.push( other_pair[m] );// 'm' prefere 'b' ao par actual
        other_pair[m] = b;    // other_pair[m] fica sem par
        optimal_pair[b] = m;    // casa "other" m com "optimal" b
      }
      else  optimal_free.push(b);  // rejeitado
  }
}
```

```
int main()
{
  int nc , i , j , k;
  scanf("%d\n", &nc);
  while (nc--) {
    scanf("%d", &N);
    for (i = 1 ; i <= N ; i++) {  // preferencias de quem tem preferencia de escolha
      optimal[i] = queue<int>();  // limpar a queue
      for (j = 1 ; j <= N ; j++) {
        scanf("%d", &k);
        optimal[i].push(k);
      }
    }
    for (i = 1 ; i <= N ; i++)    // preferencias do outro grupo
      for (j = 1 ; j <= N ; j++) {
        scanf("%d", &k);
        other[i][k] = j;
      }
    StableMarriage();
    for (i = 1 ; i <= N ; i++)
      printf("%d\n" , optimal_pair[i]);
    if (nc)
      printf("\n");
  }
  return 0;
}
```

Deterministic Finite Automata Minimization

```
struct dfa
{
 int nvertex;
 int nalphabet;
```

```
 int start;

 int symbols[256];
 int next[MAXV][MAXE];
 bool final[MAXV];
 void init(int nv, int na) { start = 0; nvertex = nv; nalphabet = na; memset(final, 0, sizeof(final)); }
};

dfa minimize(dfa & target)
{
 dfa minimal;
 bool different[MAXV][MAXV];
 int conversion[MAXV];

 bool done = false;
 int nvertex = 0;

 memset(different, 0, sizeof(different));

 /* Two states, initially, are different if one is final and the other one isn't */
 for (int i = 0; i < target.nvertex; i++)
  for (int j = i+1; j < target.nvertex; j++)
   different[i][j] = different[j][i] = target.final[i] != target.final[j];

 while (!done)
 {
  done = true;
  for (int i = 0; i < target.nvertex; i++)
   for (int j = i+1; j < target.nvertex; j++)
    if (!different[i][j])
     for (int k = 0; k < target.nalphabet; k++)
      if (different[target.next[i][k]][target.next[j][k]])
      { /* Two states are different if their 'next' states are different */
       done = !(different[i][j] = different[j][i] = true);
       break;
      }
 }

 /* Calculate the sets of different states */
 for (int i = 0; i < target.nvertex; i++)
  for (int j = 0; j <= i; j++)
   if (!different[i][j])
   { conversion[i] = i == j ? nvertex++ : conversion[j]; break; }

 /* Build the minimal DFA */
 minimal.init(nvertex, target.nalphabet);
 for (int i = 0; i < target.nvertex; i++)
  for (int k = 0; k < target.nalphabet; k++)
   minimal.next[conversion[i]][k] = conversion[target.next[i][k]];

 /* Fill the "final state" information on the minimal DFA */
 for (int i = 0; i < target.nvertex; i++)
  minimal.final[conversion[i]] = target.final[i];

 memcpy(minimal.symbols, target.symbols, sizeof(minimal.symbols));
 return minimal;
}
```

Algoritmo Húngaro

```
#define Set(S,x)  memset( S , x , sizeof(S) );
#define FOR(i,j,k)  for ( int i = j ; i < k ; i++ )
const int N = 355 , INF = 100000000 ;
int cost[N][N], n, work, max_match, lx[N], ly[N], xy[N], yx[N], slack[N], slackx[N], prev[N], S[N], T[N];

void add2tree(int x, int p) {
  S[x] = 1 , prev[x] = p ; FOR(y,0,n) if ( ( p = lx[x] + ly[y] - cost[x][y] ) < slack[y])
            slack[y] = p , slackx[y] = x;
}
int hungarian() {
  int ret = max_match = 0 ,cx, cy, ty, x, y, root, delta, q[N], wr, rd;
  Set( xy , -1 ); Set( yx , -1 ); Set( lx , 0 ); Set( ly , 0 );
  FOR(x,0,n) FOR(y,0,n)  lx[x] = max(lx[x], cost[x][y]);
  while (max_match != n) {
    Set( S , 0 ); Set( T , 0 ); Set( prev , -1 );
    for (x = wr = rd = 0 ; x < n ; x++) if (xy[x] == -1)
      q[wr++] = root = x , prev[x] = -2 , S[x] = 1 , x = n;
    for (y = 0 ; y < n ; slackx[y++] = root )
      slack[y] = lx[root] + ly[y] - cost[root][y];
    while ( y >= n ) {
      for ( y = 0 ; rd < wr && y < n ; ) {
        x = q[rd++];
        for (y = 0; y < n; y++)
          if (cost[x][y] == lx[x] + ly[y] && !T[y])
            if (yx[y] == -1) break;
            else T[y] = 1 , q[wr++] = yx[y] , add2tree(yx[y], x);
      }
      if (y < n) break;
      for (y = 0 , delta = INF ; y < n ; y++)  if (!T[y])  delta = min(delta, slack[y]);
      FOR(a,0,n) {  if (S[a])  lx[a] -= delta;
          T[a] ? ly[a] += delta : slack[a] -= delta; }
      for (y = wr = rd = 0; y < n; y++)
        if (!T[y] && slack[y] == 0)
```

```
          if (yx[y] == -1) {
            x = slackx[y];  break;
          } else {
            T[y] = 1 ;
            if (!S[yx[y]])  q[wr++] = yx[y] , add2tree(yx[y], slackx[y]);
          }
    }
    if (y < n)  for ( max_match++ , cx = x, cy = y ; cx != -2; cx = prev[cx], cy = ty)
          ty = xy[cx] , yx[cy] = cx , xy[cx] = cy;
  }
  FOR(x,0,work)  ret += cost[ x ][ xy[ x ] ];
  return ret;
}
```

```
int main() {
  int i , ncases , p , a , b , c;
  scanf("%d" , &ncases );
  while (ncases--) {        // como a matriz podia nao ser quadrada neste exercício
    scanf("%d %d %d" , & work, & n , &p);  // 'n' tem o valor máximo de vértices de um lado do grafo

    memset( cost , 0xBF , sizeof(cost) );  // queremos o emparelhamento minimo,inicializar os custos
              // com um valor muito negativo (menor que -1.000.000.000)
    for (i = 0 ; i < p ; i++) {
      scanf("%d %d %d" , &a , &b , &c );
      cost[a][b] = -c;    // colocar peso  com sinal contrario para encontrar
    }           // o emparelhamento de peso minimo
    if (ncases)  printf("%d\n\n", -hungarian());  // como os pesos sao negativos a solucao também
    else    printf("%d\n", -hungarian());  // é negativa: inverter o sinal
  }
  return 0;
}
```

Euler Path and Euler Circuit

Determina um circuito euleriano num grafo não dirigido. Se existirem 2 (e só 2) vertices de grau impar, apenas existe um caminho euleriano entre esses pontos.

```
struct Edge {
  int v1,v2, d;    // d tem o numero de arestas entre os vertices
};
struct Nodo {
  int ind , le , nadj , grau;
  Edge * e[MAX];
} g[MAX];
int circuit[MAXE+5] , circuitpos;

void find_circuit( int v ) {
  if ( g[v].le >= g[v].nadj)
    circuit[circuitpos++] = v;
  else {
    while ( g[v].le < g[v].nadj ) {
      Edge * e = g[v].e[g[v].le];

      if ( e->d )   e->d-- , find_circuit( e->v1 == v ? e->v2 : e->v1);
      else      g[v].le++;
    }
    circuit[circuitpos++] = v;
} }
```

```
int main(){
  int F , i , j , k;
  scanf("%d",&F);
  for (k=0; k<F ; k++) {    // ler ligacoes entre vertices
    scanf("%d %d",&i,&j);
       g[i].grau++, g[j].grau++;
    if (g[i].e[ j ] == NULL) {
      Edge *e = new Edge;
      e->d = 1 , e->v1 = i , e->v2 = j;
      g[j].e[ i ] = g[i].e[ j ] = e;
    }
    else  g[i].e[ j ]->d++;  // incrementar o degree da aresta
  }
  for (i=1; i<MAX ; i++) {         // transformar matriz adjacencias
    g[i].ind = i, g[i].le = g[i].nadj = 0;// em lista de adjacencias
    for (j=1; j<MAX ; j++)
      if (g[i].e[j] != NULL)
        g[i].e[ g[i].nadj++ ] = g[i].e[j];
  }
  circuitpos = 0;
  find_circuit(i); // começar com 1vértice c arestas e que tenha grau impar caso exista
  for (i=circuitpos-1 ; i>=0; i--)
    printf("%d\n",circuit[i]);
  return 0;
}
```

2-SAT

Determina se um conjunto de clausulas com no máximo 2 variáveis é satisfazivel.

```
#define T(x) 2*(x)
#define F(x) 2*(x)+1
int N ;  // número de variáveis  1..N
vector<int> adj[MAX] , rev[MAX]; // rev só necessário no algoritmo de kosaraju

void addClause( int x , bool bx , int y , bool by ) { //vars e valor booleano na clausula
```

```
  int hipx, hipy , tx,ty;

  if (bx)    hipx = F(x) , ty = T(x);
  else    hipx = T(x) , ty = F(x);

  if (by)    hipy = F(y) , tx = T(y);
  else    hipy = T(y) , tx = F(y);

  adj[hipx].PB(tx);  rev[tx].PB(hipx); // rev só necessário no algoritmo de kosaraju
  adj[hipy].PB(ty);  rev[ty].PB(hipy);
}
bool eSatisfazivel() {
  bool ok = true;          // Se A e ~A estiverem na mesma
  for (int i = 1 ; ok && i <= N ; i++)    // componente, não
    ok = comp[ T(i) ] != comp[ F(i) ];  // é satisfazivel
  return ok;
}
```

```
int main() {
  // ler o input e adicionar as clausulas
  // calcular componentes fortemente conexas
  // ver o resultado de eSatisfazivel()
}
```

2. Matemática Discreta

Adittion Chain

Calcular x ^ y mod n de uma forma eficiente.

```
long long addchain(long long int x, int y, int n)
{
  long long t = 1;
  for(; y; x = (x*x) % n, y >>= 1) if (y & 1) t = (t*x) % n;
  return t;
}
```

```
int main(){
    cout << addchain(423,288, 10) << endl;
    cout << addchain(1424,1288, 10) << endl;
    cout << addchain(1651073,13, 1651071) << endl; //testar overflows
}
```

Combinações

Quando queremos calcular combinacoes e sabemos que o resultado cabe num long long, mas os calculos intermedios nao, podemos usar o truque de dividir sempre os dois numeros pelo maior divisor comum antes de fazer as multiplicações.

```
long long gcd(long long a, long long b) {
  while(b) swap(a%=b, b);
  return a;
}
void divbygcd(long long & a,long long & b) {
 long long g=gcd(a,b);
 a/=g;
 b/=g;
}
long long comb(int n,int k) {
 long long numerator = 1 , denominator = 1 , toMul , toDiv , i ;

 if ( k > n/2 ) k = n - k ; /* use smaller k */
 for ( i = k ; i ; i-- ) {
  toMul = n - k + i ;
  toDiv = i ;
  divbygcd( toMul , toDiv );    /* always divide before multiply */
  divbygcd( numerator , toDiv );
  divbygcd( toMul , denominator );
  numerator *= toMul ;
  denominator *= toDiv ;
 }

 return numerator / denominator;
}
```

```
int main(void) {
 cout << comb(100, 10) << endl;
 return 0;
}
```

Crivo de Erastótenes (Bitwise)

Versão com bastantes optimizações e que consome menos memória (100.000.000 → 6MB). Necessita de num/16 bytes. Apenas necessário quando o crivo convencional não é suficiente, dado que é mais extenso e complexo.

```
#define MAX 100000000
#define PR_POS( v ) ( v >> 4 )
#define PR_BIT( v ) (unsigned char)( (v >> 1) & 7 )
#define PRIMOS( v ) ( ( ( v & 1 ) && ( primos[PR_POS(v)] & ( 1 << PR_BIT(v) ) ) ) || v == 2 )
#define SET_NP( v ) ( primos[PR_POS(v)] &= ~( 1 << PR_BIT(v) ) ) // Marca como não sendo primo

typedef unsigned long long Tipo;

unsigned char primos[ PR_POS(MAX) + ( PR_BIT(MAX) ? 1 : 0 ) ]; // Apenas MAX

void crivo() {
```

```
  memset( primos, 0xFF, sizeof( primos ) );
  primos[0] = 0x6E; // 1, 9 e 15 não são primos (bits por ordem inversa)

  for( Tipo i = 9; i < MAX; i += 6 ) SET_NP(i); // Múltiplos de 3
  for( Tipo i = 5, f = 2; i <= sqrt(MAX); i += f, f = 6-f ) // Mais eficiente a percorrer
    if( PRIMOS(i) )
      for( Tipo val = i*i; val < MAX; val += ( i << 1 ) ) SET_NP(val);
}

int main() {
  crivo();
  for( Tipo i = 0; i < MAX; ++i )
    if( PRIMOS(i) )
    printf( "%llu\n", i ); // Ajustar de acordo com o Tipo
  return 0;
}
```

Números de Catalan

Os números de Catalan aparecem em diversos problemas de contagem.
Exemplos:UVA 10223
Referencia:wikipedia
1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324

```
long long catalan[50] = {1};

void init_catalan() {
  for(int i=1; i<50; i++)
    for(int j=0; j<i; j++)
      catalan[i] += catalan[j]*catalan[i-j-1];
}
```

Teorema de bezout

Determina a e b tais que a*x + b*y == gcd(x , y). **find_bezout(v,m).first** dá a inversa modular de **v** módulo **m**, **sE gcd(v,m)=1**, ou seja, v e m são primos entre si. Nota: o resultado.first pode ser negativo!!!

```
typedef pair<int,int> bezout;

bezout find_bezout(int x, int y){
  if (y == 0)  return bezout(1,0);
  bezout u = find_bezout(y , x % y);
  return bezout( u.second , u.first - (x/y) * u.second);
}
```

Congruencias

Acha a menor solução não-negativa de a * x = b mod m .

```
int mod( int x , int m) {  return x % m + ( x < 0 ) ? m : 0;  }

// retorna -1 se a congruencia for impossivel
int solve_mod( int a , int b , int m ) {
  if ( m < 0 )  return solve_mod( a , b , -m );
  if ( a < 0 || a >= m || b < 0 || b >= m)
    return solve_mod( mod( a , m ) , mod( b , m ) , m );

  bezout t = find_bezout( a , m );
  int d = t.first * a + t.second * m;
  if (b % d)  return -1;
  else    return mod( t.first * ( b / d ) , m );
}
```

Eliminação Gaussiana Modular

```
#define MAX 75
int eq[MAX][MAX] , res[MAX];
void gauss( int neq , int p ) {         // numero de equacoes , modulo p
  int i , j , k;

  for (i = 0 ; i < neq ; i++) {
    if (eq[i][i] != 1 ) {
      int inversa = ( find_bezout( eq[i][i] , p ).first + p ) % p;

      for (j = i+1 ; j < neq ; j++)
        eq[i][j] = ( eq[i][j] * inversa ) % p ;
      res[i]=(res[i] * inversa ) % p ;    // eq[i][i]=1;desnecessario
    }
    for ( k = i+1 ; k < neq ; k++) {
      for (j = i+1 ; j < neq ; j++)
        eq[k][j] = ( eq[k][j] + p - ( (eq[k][i]*eq[i][j] ) % p ) ) % p;

      res[k]=(res[k]+p-((eq[k][i] * res[i] )%p) ) % p;// eq[k][i]=0;desnecessario
    }
  }
  for (i = neq -1 ; i >= 0 ; i--)
    for (k = 0 ; k < i ; k++)
      res[k]=(res[k]+p-((eq[k][i] * res[i] )%p) ) % p;// eq[k][i]=0;desnecessario
}

int main() {
  int t , p , i , j , n , w;
  char s[100];
```

```
  scanf("%d" , &t);
  while (t--) {
    scanf("%d %s\n", &p , s );
    for (i = 0 ; s[i] ; i++) {
      if (s[i] == '*')  res[i] = 0;
      else      res[i] = (s[i]-'a'+1) % p;
    }
    n = i;
    for (i = 0 ; i < n ; i++) {
      w = i+1;
      eq[i][0] = 1;
      for (j = 1 ; j < n ; j++)
        eq[i][j] = ( eq[i][j-1] * w ) % p ;
    }
    gauss(n,p);

    printf("%d" , res[0]);
    for (i = 1 ; i < n ; i++)
      printf(" %d" , res[i]);
    printf("\n");
  }
  return 0;
}
```

Número de Divisores

Um número na forma $p_1^{k_1} * p_2^{k_2} * p_3^{k_3}$.. sendo p_i um número primo, tem $(k_1+1)(k_2+1)(k_3+1)$.. divisores.

```
int num_divisores(int n) {
  int div = 1;
  for(int i=2; i*i<=n; i++) {
    int ct = 1;
    while(n % i == 0) n/=i, ct++;
    div *= ct;
  }
  return div*((n != 1)?2:1);
}
```

Número de Coprimos (Euler Totient Function)

Para N na forma $p_1^{k_1} * p_2^{k_2} * p_3^{k_3}$.. sendo p_i um número primo, há $\Pi(p_i-1)*(p_i^{(k_i-1)})$ coprimos de 1 até N.
1 é coprimo de todos os inteiros ⇒ gcd(1, X) = 1.

```
int num_coprimos(int n) {
 int coprimes = n;
  for(int p=2; p*p<=n; ++p) {
   bool any = false;
   while(n % p == 0) n/=p, any = true;
   if(any) coprimes = coprimes/p*(p-1);
  }
  return (n != 1 ? coprimes/n*(n-1) : coprimes);
}
```

Números de Fibonacci

Usando multiplicação de matrizes e divide and conquer para calcular as potências.

```
int nth_fibonacci(int n) {
  int p[2][2] = { {1,1}, {1,0} }, r[2][2] = { {1,0}, {0,1} };
  for(; n; n>>=1)
  {
    if(n & 1) mul(r, r, p);
    mul(p, p, p);
  }
  return r[0][0];
}
```

Calcula o n-ésimo elemento da sequência de f-bonacci. Utiliza a Estrutura de matrizes.

```
matrix mk_nacci(int n) {
 matrix m;
 m.rows = m.columns = n;
 for (int i = 0; i < n; i++)
  m.v[0][i] = m.v[i==n-1?0:i+1][i] = 1;
 return m;
}
int nacci(int f, int n) {
 if (n < f-1) return 0;
 matrix m = mk_nacci(f);
 return (m^n).v[f-1][0];
}
```

Miller-Rabin Primality Test

```
/* this function calculates (a*b)%c taking into account that a*b might overflow */
long long mulmod(long long a,long long b,long long c){
  long long x = 0,y=a%c;
  for ( ; b > 0 ; y = (y*2)%c , b >>= 1)
    if( b & 1)
      x = (x+y)%c;
  return x%c;
}
/* This function calculates (a^b)%c */
long long modulo(long long a,long long b,long long c){
```

```
  long long x=1,y=a;
  for ( ; b > 0 ; y = mulmod(y,y,c) , b >>= 1)
    if(b & 1)
      x=mulmod(x,y,c);
  return x%c;
}
/* Test, iteration signifies the accuracy of the test. 20 should be ok */
bool Miller(long long p,int iteration){
  if(p<2)       return false;
  if(p!=2 && p%2==0)  return false;

  long long a , temp , mod , s=p-1;
  while( (s & 1) ==0)  s >>= 1;

  while (iteration--){
    a=rand()%(p-1)+1,temp=s;

    for( mod=modulo(a,temp,p); temp!=p-1 && mod!=1 && mod!=p-1 ; temp <<= 1 )
      mod=mulmod(mod,mod,p);

    if(mod!=p-1 && temp%2==0)
      return false;
  }
  return true;
}
```

)

Fraccionários

Fracções. O expand() serve para resolver dízima periódicas: 5.8(144) = expand(58, 1, 144, 3) = 3227/555. o 1 é o número de casas decimais antes da dízima, eo 3 é o comprimento da dízima.

```
struct frac{
  long num, den;
  frac() { num = 0; den = 1; }
  frac(long d) { num = d; den = 1; }
  frac(long d, long n) { num = d; den = n; }
  frac(const frac & o) { num = o.num; den = o.den; }
  double v() const{return (double)num/den;}
  frac operator +(const frac & o) {frac r(num * o.den + o.num * den, den * o.den); r.s(); return r;}
  frac operator -(const frac & o) {frac r(num * o.den - o.num * den, den * o.den); r.s(); return r;}
  frac operator *(const frac & o) {frac r(num * o.num, den * o.den); r.s(); return r;}
  frac operator /(const frac & o) {frac r(num * o.den, den * o.num); r.s();return r;}
  void operator =(const frac & o) {den = o.den; num = o.num;}
  bool operator ==(const frac & o) {return num * o.den == o.num * den;}
  bool operator != (const frac & o) {return num * o.den != o.num * den;}
  bool operator <(const frac & o) {return num * o.den < o.num * den;}
  bool operator >(const frac & o) {return num * o.den > o.num * den;}
  void s() {
    long mdc = gcd(abs(num), abs(den));
    if( den < 0 ) mdc = -mdc;
    num /= mdc; den /= mdc;
  }
};
// expand(n, j, m, k) = ( n + 0.(m) ) / 10^j | lpow(a, b) = a^b
frac expand(long n, int j, long m, int k) {
 long j10 = lpow(10, j), k10 = lpow(10, k);
 frac f(m, (k10 - 1) * j10);
 return frac(n, j10) + f;
}
```

3. Matrizes

Estrutura básica para matrizes. contém multiplicação e potenciação. Multiplicação: O(n^3), Potenciação: O(n^3 * log y) Útil para o *nacci, e utilizado no MCM.

```
struct matrix { int v[MAX][MAX]; int rows, columns; };

matrix operator*(matrix & a, matrix & b) {
 matrix c;
 c.rows = a.rows;
 c.columns = b.columns;

 for (int i = 0; i < a.rows; i++)
  for (int j = 0; j < b.columns; j++)
   for (int k = c.v[i][j] = 0; k < a.columns; k++)
    c.v[i][j] += a.v[i][k]*b.v[k][j];
 return c;
}
matrix identity(int n) {
 matrix m; m.rows = m.columns = n;
 for (int i = 0; i < n; i++)
 for (int j = 0; j < n; j++) m.v[i][j] = (i==j);
 return m;
}
matrix operator^(matrix & m, int n) {
 if (n == 0) return identity(m.rows);
 if (n == 1) return m;
 matrix mt = m^(n/2), r = mt*mt;
 return (n % 2) ? r*m : r;
}
```

Matrix Chain Multiplication

Dada uma cadeia de multiplicação de matrizes, calcula a ordem de multiplicações óptima, de modo a minimizar o número de operações.

```
#define MMAX 100
int nm , c[MMAX][MMAX] , p[MMAX][MMAX] ;
matrix m[MMAX];

int mcm_find() {
 int v;
 for (int i = 0; i < nm; i++)
 for (int j = 0; j < nm; j++)
  c[i][j] = i == j ? 0 : numeric_limits<int>::max();

 for (int s = 1; s <= nm; s++) /* s = Chain Length - 1*/
  for (int i = 0; i < nm-s; i++) /* i = Chain Start index */
   for (int k = i, j = i+s; k < j; k++) /* Cut after k; j = Chain End index */
    if ((v = c[i][k] + c[k+1][j] + m[i].rows * m[k].columns * m[j].columns) < c[i][j])
     c[i][j] = v , p[i][j] = k;
 return c[0][nm-1];
}
matrix mcm(int i, int j) {
 matrix a, b;
 if (i == j) return m[i];
 a = mcm(i, p[i][j]);
 b = mcm(p[i][j]+1, j);
 return a*b;
}
```

4. Numéricos

```
#define MAX 10
typedef int Type;
struct polynomial {
    Type c[MAX];

    polynomial() { memset(c, 0, sizeof(c)); }
    Type & operator[](unsigned int i) { return c[i]; }

    int max(int i = MAX) { while (--i >= 0) if (c[i]) return i; return MAX; }

    polynomial operator+(polynomial q) { for (int i = 0; i < MAX; i++) q[i] += c[i]; return q; }
    polynomial operator-() const { polynomial q;
        for (int i = 0; i < MAX; i++) q[i] = -c[i];
        return q;
    }
    polynomial operator-(polynomial q) { return *this + -q; }
    polynomial operator*(polynomial q) { polynomial r;
        for (int i = 0; i < MAX; i++) if (c[i])
            for (int j = 0; j+i < MAX; j++) r[i+j] += c[i]*q[j];
        return r;
    }
    double operator()(double v) {
        double r = 0;
        for (int i = MAX-1; i >= 0; i--) r = c[i] + r*v;
        return r;
    }
};

polynomial operator/(polynomial num, polynomial & denum) {
    polynomial quo;
    int mn = num.max(), md = denum.max();
    while(md <= mn && mn < MAX) {
        Type tmp = num[mn] / denum[md];
        for (int i = mn; i >= mn-md; i--)
            num[i] -= tmp*denum[i-(mn-md)];

        quo[mn-md] += tmp;
        mn = num.max(mn);
    }
    return quo;
}
polynomial operator%(polynomial & num, polynomial & denum) { return num - num/denum*denum; }
```

5. Strings

Knuth-Morris-Pratt

O algoritmo Knuth-Morris-Pratt de pesquisa em strings permite pesquisar strings dentro de strings de uma forma mais eficiente que as funções da API de c++.

```
#define MAXP 50
#define MAXA 7

void initnext(char *p,int next[])
{
 int i,j,M=strlen(p);
 next[0]=-1;
 for (i=0,j=-1;i<M;i++,j++,next[i]=(p[i]==p[j]) ? next[j] : j)
  while((j>=0) && (p[i]!=p[j])) j=next[j];
}
int kmpsearch(char *a, char *p)
{
 int i,j, M=strlen(a), N=strlen(p);
 int next[MAXP];
 initnext(a,next);
 for (i=0, j=0;j<M && i<N; i++, j++)
  while((j>=0) && (p[i] != a[j])) j=next[j];
```

```
 if (j==M) return i-M; else return -1;
}

int main(void) {
 char palheiro[MAXP]="olaolaolaolaolahohohohohohobbesola";
 char agulha[MAXA]="hobbes";

 cout << kmpsearch(agulha,palheiro) << endl;
 return 0;
}
```

Aho-Corasick

O algoritmo de pesquisa Aho-Corasick permite pesquisar um conjunto de substrings numa string em O(n + m + z), sendo n o somatório dos comprimentos das strings do dicionário, m o tamanho do texto sobre o qual se efectua a pesquisa e z o número de ocorrências de padrões do dicionário no texto.

```
#define ALPHABET 52 // 'a'..'z' + 'A'..'Z'
#define MAXD 1000 // tamanho maximo do dicionario

int out(char c)
{
  if (c >= 'a')
    return c-'a';
  return c-'A'+26;
}

struct Node
{
  Node *go[ALPHABET];
  Node *failure;
  list<int> matches;

  Node()
  {
    failure = NULL;
    memset(go, 0, sizeof(go));
  }

  ~Node()
  {
    for (int i = 0; i < ALPHABET; i++)
      if (go[i])
        delete go[i];
  }

  void insert(const char *s, int i)
  {
    Node *n = this;
    while (*s)
    {
      int v = out(*s);
      if (!n->go[v])
        n->go[v] = new Node();
      n = n->go[v];
      s++;
    }
    n->matches.push_back(i);
  }
};

struct Automaton
{
  Node *root;
  int words;
  int matches[MAXD];

  Automaton(char *dictionary[], int size)
  {
    root = new Node();
    words = 0;
    for (int i = 0; i < size; i++)
      insert(dictionary[i]);
    completeTransitions();
  }

  ~Automaton()
  {
    delete root;
  }

  void insert(const char *s)
  {
    int n = words++;
    root->insert(s, n);
  }

  // completeTransitions deve ser chamada depois de todos os inserts sobre o autómato
  void completeTransitions()
  {
    queue<Node *> q;
    for (int i = 0; i < ALPHABET; i++)
    {
      if (root->go[i])
      {
```

```
        q.push(root->go[i]);
        root->go[i]->failure = root;
      }
      else
        root->go[i] = root;
    }
    while (q.size())
    {
      Node *current = q.front();
      q.pop();
      for (int i = 0; i < ALPHABET; i++)
      {
        if (current->go[i])
        {
          q.push(current->go[i]);
          Node *f = current->failure;
          while (!f->go[i])
            f = f->failure;
          current->go[i]->failure = f->go[i];
          current->go[i]->matches.insert(
              current->go[i]->matches.end(),
              f->go[i]->matches.begin(),
              f->go[i]->matches.end());
        }
      }
    }
  }

  void match(const char *s)
  {
    Node *n = root;
    memset(matches, 0, sizeof(matches));
    while (*s)
    {
      int v = out(*s);
      while (!n->go[v])
        n = n->failure;
      n = n->go[v];
      for (list<int>::iterator itr = n->matches.begin(); itr
          != n->matches.end(); itr++)
        matches[*itr]++;
      s++;
    }
  }
};

int main()
{
  char *dic[] = {"abc",  // palavra 0
          "abAB"}; // palavra 1
  Automaton *a = new Automaton(dic, 2);
  a->match("abcdefghABCDEFGH");
  printf("abc: %d\n", a->matches[0]);
  printf("abAB: %d\n", a->matches[1]);
  return 0;
}
```

Levenshtein Distance

Calcula a distância de levenshtein entre duas strings: número de modificações do tipo inserir, apagar e substituir necessárias para passar de uma string para outra. As primeiras 3 variáveis definidas definem o custo de cada uma dessas operações.
Atenção: Não testei o algoritmo foi copiado directamente da wikipedia.

```
const unsigned int cost_del = 1 , cost_ins = 1 , cost_sub = 1;

unsigned int edit_distance( const string & s1, const string & s2 )
{
 unsigned int i,j,tmp, n1 = s1.length() , n2 = s2.length();
 unsigned int * p = new unsigned int[n2+1] , * q = new unsigned int[n2+1];

 p[0] = 0;
 for( j = 1; j <= n2; ++j )
  p[j] = p[j-1] + cost_ins;

 for( i = 1; i <= n1; ++i )
  {
   q[0] = p[0] + cost_del;
   for( j = 1; j <= n2; ++j )
    {
     unsigned int d_del = p[j] + cost_del;
     unsigned int d_ins = q[j-1] + cost_ins;
     unsigned int d_sub = p[j-1] + ( s1[i-1] == s2[j-1] ? 0 : cost_sub );
     q[j] = min( min( d_del, d_ins ), d_sub );
   }
   swap( p , q );
  }
 unsigned int tmp = p[n2];
 delete[] p; delete[] q;

 return tmp;
}
```

Permutação N de uma string / Obter o número da permutação (String)

Considerando um alfabeto, gera a permutação N ou obtém o número K da permutação.

```
#define ULL unsigned long long
#define MAX 80    // permite até C(80,40)
#define FAC_LIMIT 16 // 22! nao cabe num ull

using namespace std;

// c: comb. f: fact. u: ocorrencias dos chars. sum_u: somatorio de u[]
ULL sum_u = 0, f[FAC_LIMIT] = {1}, len_u, u[MAX], c[MAX][MAX];
char alphabet[MAX];

ULL distinct(ULL n, ULL u[]) {
 ULL calc = 1, x = n-1, single = 0;

 for( ULL i = 0; i < len_u; ++i) {
  if(u[i] < 2) { single += u[i]; continue; }

  calc *= comb(x, u[i]), x -= u[i];
 }

 return calc*f[single];
}

// u: contador de cada caracter. w: permutacao k dos caracteres de u.
void nth_perm(ULL k, ULL u[], char w[], ULL len_u) {
 ULL n = sum_u, picked, a, i;

 for (i = 0; n > 0; --n, ++i) {
  for (picked = 0; picked < len_u; ++picked) {
   if(!u[picked]) continue;
   --u[picked], a = distinct(n, u), ++u[picked];

   if (k < a) break;
   k -= a;
  }
  --u[picked], w[i] = alphabet[picked];
 }

 w[i] = '\0'; // Para poder ser impressa directamente com o printf
}

// u: ocorrencias de cada caracter. w: string permutada.
ULL perm_k(ULL u[], const char w[], ULL len_u) {
 ULL n = sum_u, picked, k = 0;

 for (ULL i = 0; n > 0; n--, ++i, --u[picked])
  for (picked = 0; picked < len_u; ++picked) {
   if (w[i] <= alphabet[picked]) break;
   if(!u[picked]) continue;

   --u[picked], k += distinct(n, u), ++u[picked];
  }

 return k;
}

void alphabetize(string &s) {
  alphabet[0] = s[0]; len_u = u[0] = 1; sum_u = 1;

  for (unsigned int i = 1; i < s.size(); ++i, ++sum_u) {
    if (s[i] != alphabet[len_u-1]) {
      u[len_u] = 1;
      alphabet[len_u++] = s[i];
    }
    else ++u[len_u-1];
  }
}
```

```
int main() {
  string str, s;
  memset(c, 0, sizeof(c));
  char w[MAX];

  while (cin >> str && str != "#") {
    sum_u = len_u = 0;
    s = str;
    sort(s.begin(), s.end());

    alphabetize(s);
    printf("%10llu\n", perm_k(u, str.c_str(), len_u)+1);
  }

  ULL k;
  cin >> s;
  sort(s.begin(), s.end());

  while(cin >> k && k) {
    alphabetize(s);
    nth_perm(k-1, u, w, len_u);
    printf("%s\n", w);
  }
```

```
  return 0;
}
```

Gramática sem contexto

```
grammar loglan;
loglan.add_rule('S', "XY$");
loglan.add_rule('X', "AB");
loglan.add_rule('Y', "C");
loglan.add_rule('Y', "");

loglan.generate_first();
loglan.generate_follow();
loglan.generate_table();

queue<token> sequence; // encher cos tokens
cout << (loglan.valid(sequence) ? "Good" : "Bad!") << endl;
```

Código:

```
typedef char token;

#define NTERMINALS 8
token terminals[] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', '$' };
bool terminal(token t) { return ( t >= 'A' && t <= 'G' ) || t == '$'; }

#define HAS_EPSILON(set) ((set).find( EPSILON ) != (set).end())

struct rule {
 token t; string p; set<token> f;
 rule(token _t, string _p) { t = _t; p = _p; }
};

struct grammar {
 map<token, set<token> > first, follow;
 vector<rule> rules;
 map<pair<token, token>, int> parse_table;

 void add_rule(token s, string tokens) { rules.push_back(rule(s, tokens)); }
 void generate_first() {
  for (int i = 0; i < NTERMINALS; i++) first[ terminals[i] ].insert( terminals[i] );
  for (int i = 0; i < (int)rules.size(); i++) if (rules[i].p.empty()) rules[i].f.insert( EPSILON );
  for (bool changes = true; changes; ) { changes = false;
   for (int i = 0; i < (int)rules.size(); i++) {
    for (int j = 0; j < (int) rules[i].p.size(); j++) {
     changes |= insert( rules[i].f, first[ rules[i].p[j] ] );
     if (!HAS_EPSILON( rules[i].f ) ) break;
    }
    insert( first[ rules[i].t ], rules[i].f, true );
   }
  }
 }

 void generate_follow() {
  for (bool changes = true ; changes; ) { changes = false;
   for ( int i = 0; i < (int)rules.size() ; i++ )
    for ( int j = rules[i].p.size() - 1, epsilon = 1; j >= 0; j-- ) {
     if ( j + 1 < (int)rules[i].p.size() )
      changes |= insert( follow[ rules[i].p[j] ], first[ rules[i].p[j + 1] ] );
     if ( epsilon )
      changes |= insert( follow[ rules[i].p[j] ], follow[ rules[i].t ] );
     epsilon = HAS_EPSILON( first[ rules[i].p[j] ] );
    }
  }
 }
 void generate_table() {
  for ( int i = 0; i < (int) rules.size(); i++ ) {
   for ( set<token>::iterator it = rules[i].f.begin(); it != rules[i].f.end(); it++) {
    pair<int, int> entry(rules[i].t, *it);
    if (parse_table[entry] > 0)
     printf("error\n");
    parse_table[entry] = i + 1;
   }
   if ( HAS_EPSILON( rules[i].f ) )
    for ( set<token>::iterator it = follow[rules[i].t].begin(); it != follow[rules[i].t].end(); it++)
    {
     pair<int, int> entry(rules[i].t, *it);
     if (parse_table[entry] > 0)
      printf("error\n");
     parse_table[entry] = i + 1;
    }
  }
 }

 bool valid(queue<token> & se) { stack<token> st; st.push('S'); se.push('$'); return valid(se, st); }
 bool valid(queue<token> & sentence, stack<token> & tokens)
 {
  while (!(sentence.empty() || tokens.empty()))
  {
   if (terminal( tokens.top() ) && sentence.front() == tokens.top())
   {
    tokens.pop();
    sentence.pop();
```

```
    continue;
   }

   int r = parse_table[make_pair(tokens.top(), sentence.front())] - 1;
   if (r < 0)
    return false;

   tokens.pop();
   for (int i = rules[r].p.size() - 1; i >= 0; i--)
    if (rules[r].p[i] != '&')
     tokens.push(rules[r].p[i]);
  }

  return sentence.empty() && tokens.empty();
 }
};
```

Huffman Codes

Algoritmo de compressão de Huffman. Garante a solução óptima.

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <queue>
#include <string>

int childs;
#define U unsigned
#define MAX 256
#define CHILD 10    // MAX 10 => (char)(i|0x030)
#define FOR_N for(int i = 0; i < childs; ++i)
using namespace std;

struct Node { Node *n[CHILD]; U long f; U char c; string code;
 Node(U long u=0, U char ch='\0') { FOR_N n[i]=NULL; f=u; c=ch; code=""; }
 bool operator<(const Node &a) const { return f > a.f; }  // reverse
 static Node* link(Node* arr[], U long sum){ Node *a = new Node(sum);
  FOR_N a->n[i] = arr[i]; return a;
 }
 void coder(string s) {
  FOR_N if(n[i]) n[i]->coder(s+(char)(i|0x30)); if(c) code=s; else delete this; }
};

bool cmp(const Node *a, const Node *b) { return a->f > b->f; } // reverse

void analyze(string str, vector<Node*> &vec, Node freq[]) {   // vec vazio
  U char ch;
  for(size_t i = 0; i < str.size(); ++i)
    if(++freq[ch = (U char)str[i]].f < 2)
      freq[ch].c = ch, vec.push_back(&freq[ch]);
}

int generate(string str, Node freq[]) {
  vector<Node*> v; Node *x[CHILD]; U long sum = 0;
  v.reserve(MAX); int i, size, count;

  analyze(str, v, freq); count = size = v.size(); FOR_N x[i]=NULL;
  sort(v.begin(), v.end(), cmp); make_heap(v.begin(), v.end(), cmp);

  while(size > 1) {
   for(i = 0; i < childs && size > 0; ++i)
    x[i] = v[0], pop_heap(v.begin(), v.begin()+size--, cmp), sum += x[i]->f;
   v[size] = Node::link(x, sum);
   push_heap(v.begin(), v.begin() + size++, cmp);
  }

  v[0]->coder(""); return count;
}
```

```
int main() {
  string str; int cases, count; Node freq[MAX]; // unsigned char as key
  cin >> cases;

  while(cases--) {
   cin >> str >> childs; count = generate(str,freq); sort(freq, freq+MAX);

   for(int i = 0; i < count; ++i)
     cout << freq[i].c << " => " << freq[i].code << ": " << freq[i].f << endl;
  }

  return 0;
}
```

6. Sequências

Longest Common Subsequence

A Maior Subsequência Comum (LCS) consiste em encontrar a maior sequência comum entre dois conjuntos, não necessariamente consecutiva. Por Exemplo, se A={1,2,3,4,5,0} e B={0,1,3,0}, então LCS(A,B)={1,3,0}.

```
#define MAX 100   // change this constant if you want a longer subsequence

char X[MAX],Y[MAX];
int i,j,m,n,c[MAX][MAX],b[MAX][MAX];
```

```
int LCSlength() {
 m=strlen(X);
 n=strlen(Y);

 for (i=1;i<=m;i++) c[i][0]=0;
 for (j=0;j<=n;j++) c[0][j]=0;

 for (i=1;i<=m;i++)
  for (j=1;j<=n;j++) {
   if (X[i-1]==Y[j-1]) {
    c[i][j]=c[i-1][j-1]+1;
    b[i][j]=1; /* from north west */
   }
   else if (c[i-1][j]>=c[i][j-1]) {
    c[i][j]=c[i-1][j];
    b[i][j]=2; /* from north */
   }
   else {
    c[i][j]=c[i][j-1];
    b[i][j]=3; /* from west */
   }
  }

 return c[m][n];
}
void printLCS(int i,int j) {
 if (i==0 || j==0) return;

 if (b[i][j]==1) {
  printLCS(i-1,j-1);
  printf("%c",X[i-1]);
 }
 else if (b[i][j]==2)
  printLCS(i-1,j);
 else
  printLCS(i,j-1);
}
```

```
int main() {
  gets(X);
  gets(Y);
  printf("LCS length -> %d\n",LCSlength()); /* count length */
  printLCS(m,n); /* reconstruct LCS */
  printf("\n");
  return 0;
}
```

Longest Increasing Subsequence

Encontrar a maior sequência crescente dentro de um array. Os elementos não têm de ser contíguos. O(n log k) Truque: em caso de empate, o algoritmo determina a última subsequência de tamanho máximo. Se quisermos a primeira, basta inverter o array e calcular a LDS. A última subsequência do reverso corresponde à primeira sequência crescente do array original.

```
vector<int> find_lis(vector<int> & a)
{
  vector<int> b, p( a.size() );
  int u, v , c;
  b.push_back(0);
  for (size_t i = 1; i < a.size(); i++) {
    if (a[b.back()] < a[i]) {    // trocar < por > se for Dec
      p[i] = b.back() , b.push_back(i);
      continue;
    }
    for (u = 0 , v = b.size()-1; u < v; ) {  // trocar < por > se for Dec
      c = (u + v) / 2;
      a[b[c]] < a[i] ? u = c+1 : v = c;
    }
    if (u <= v && a[i] < a[b[u]]) {    // trocar < por > se for Dec
      if (u > 0) p[i] = b[u-1];
      b[u] = i;
    }
  }
  for (u = b.size(), v = b.back(); u-- ; v = p[v])
    b[u] = v;  // b tem os indices dos elementos da subsequencia
  return b;    // b.size() tem o tamanho
}
```

```
int main() {
  int n;
  vector<int> seq;
  while (scanf("%d" , &n) != EOF)
    seq.push_back(n);

  vector<int> lis = find_lis(seq); // lis tem os indices dos elementos da subsequencia
  printf("%d\n-\n", lis.size() );
  for (size_t i = 0; i < lis.size(); i++)
    printf( "%d\n" , seq[ lis[i] ] );
  return 0;
}
```

7. Estruturas de Dados

Union Find

Permite fazer merge de conjuntos. **Não** é possível descobrir ~~quantos conjuntos existem ou~~ quais os elementos de um conjunto de forma eficiente mas é possível descobrir se dois elementos pertencem ao mesmo conjunto.
Usado em: Kruskal

```
struct set{
  int p[MAX],rank[MAX], number[MAX];
  int size, sets;

  void init(int s){
    size = sets = s; // cada elemento é um conjunto
    for (int i = 0; i < size; i++)
      {p[i]=i; rank[i]=0; number[i]=1;}
  }

  void link(int x, int y) {
    if(x == y) return; //andre.sp fix: ja tao ligados nao queremos contar o comprimento 2 vezes
    if (rank[x] <= rank[y]) {
      p[x] = y;
          --sets; //pedro.silva: se unimos dois conjuntos temos menos um
      number[y] += number[x];
      if (rank[x] == rank[y])
       rank[y]++;
    } else link(y, x);
  }

  int find_set(int x) {
   if (x != p[x]) p[x] = find_set(p[x]);
   return p[x];
  }
  int find_length(int x) {
   return number[ find_set(p[x]) ];
  }
  void union_set(int x,int y) {
   link(find_set(x), find_set(y));
  }
};
```

```
int main(void)
{
 set s; s.init(6);

 s.union_set(0,2);
 s.union_set(3,2);
 s.union_set(4,1);

 for (int i = 0; i < 6; i++)
  cout << s.find_set(i) << endl;
}
```

Trie

A Trie Example In computer science, a trie, or prefix tree, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest.

```
#define MAX 26
struct trie {
  trie * next[MAX];
  int val;

  trie() { val = 0; memset(next, 0, MAX * sizeof(trie *)); }
  ~trie() { for(int i=0;i<MAX;i++) if(next[i]) delete next[i]; }
};

//insere caso a palavra nao exista, retorna o numero de ocorrências
int trie_insert(trie * t, const char *s) {
  while(*s) {
    if(!t->next[*s-'a']) t->next[*s-'a'] = new trie();
    t = t->next[*s-'a'];
    s++;
  }
  return ++t->val;
}

int trie_search(trie *t, const char *s) {
  while(*s) {
    if(!t->next[*s-'a']) return 0;
    t = t->next[*s-'a'];
    s++;
  }
  return t->val;
}
```

```
int main(void) {
 trie t;
 printf("Inseri %s %d\n","texto1",trie_insert(&t,"texto1"));
 printf("Procurei %s %d\n","texto1",trie_search(&t,"texto1"));
 printf("Procurei %s %d\n","texto3",trie_search(&t,"texto3"));
 return 0;
}
```

Hash Table

Implementa uma hash table. Usa resolução de colisões com pesquisa linear (em principio é suficiente).

```
#define HASH_SIZE 240007  // Numero primo !!!

struct TYPE {
  char key[51] ;      // string, numero, o que for
  char used;
  bool operator == ( TYPE &t ) {
    return strcmp( key , t.key ) == 0;
  }
} hash[HASH_SIZE];

inline long hash_function( TYPE & s)
{
  unsigned long key=0;
  for (int i=0; s.key[i] ; i++)
    key = (key * 37 + s.key[i]);
  return key % HASH_SIZE;
}

inline void insert(TYPE &s)  // resolucao de colisoes com pesquisa linear
{
  long key = hash_function(s);  // assumindo que nao ha repetidos
  while (hash[key].used != 0)
  {
    key++;
    if (key==HASH_SIZE)
      key=0;
  }
  memcpy( &hash[key] , &s , sizeof(TYPE));
  hash[key].used = 1;
}

inline bool find( TYPE &s)
{
  long key = hash_function(s);
  bool found = ( s == hash[key] && hash[key].used );

  for ( ++key ; hash[key].used && !found ; found = ( s == hash[key++] ) )
    if (key==HASH_SIZE)
      key=0;
  return found;
}
```

```
int main()
{
  int i;
  TYPE word;

  while (fgets(word.key,50,stdin)!=NULL)
  {
    for (i=0; word.key[i] && word.key[i]!='\n' ; i++);
    word.key[i] = '\0';

    insert(word);
  }
  strcpy( word.key , "hobbes");

  if (find(word))    printf("encontrou\n");
  else      printf("nao encontrou\n");
  return 0;
}
```

Kd-Tree

Implementa uma Kd-Tree. Ou seja, em cada ciclo de K níveis, ordena

```
#define MAX 100100
struct Point {
  double x ,y;
  int ind;
  double dist( Point & p ) {
    double a = x-p.x , b = y-p.y;
    return a*a + b*b;
  }
  bool operator < ( const Point & p ) const {
    if ( x != p.x )  return x < p.x;
    return y < p.y;
  }
} v[MAX] , v2[MAX];

struct Node {
  Point p;
  int level;
  Node * left , *right ;
  Node( Point & pp , int lv ) {
    p = pp , level = lv, left = right = NULL;
  }
};

double R , rs ;
```

```
int N , W ,inicio;
Node * tree;

bool comparaY( const Point & a , const Point & b ) {
  if ( a.y != b.y )  return a.y < b.y;
  return a.x < b.x;
}
void balance( Node * & no , int a , int b , int level ) {
  if ( a > b )    return;
  if (level & 1 )    sort( v+a , v+b+1 );
  else      sort( v+a , v+b+1 , comparaY );
  int mid = (a+b) / 2;
  no = new Node( v[mid] , level );
  balance( no->left , a   , mid-1 , level+1 );
  balance( no->right , mid+1 , b   , level+1 );
}
Point p;  // ponto a pesquisar
int go( Node * no ) {
  int r = N*2;
  if (no == NULL)  return r;

  if ( no->p.ind >= inicio && p.ind - no->p.ind > W  &&  p.dist( no->p ) <= rs )
    r = no->p.ind;
  // range search
  if ( no->level & 1 ) {     // select em xx
    if ( p.x - R < no->p.x )  r = min( r , go( no->left ) );
    if ( p.x + R > no->p.x )  r = min( r , go( no->right ) );
  } else {      // select em yy
    if ( p.y - R < no->p.y )  r = min( r , go( no->left ) );
    if ( p.y + R > no->p.y )  r = min( r , go( no->right ) );
  }
  return r ;
}
```

```
bool check( Point & a ) {
  p = a;
  int i = go( tree );
  if ( i > N )
    return false;
  printf("%d %d\n", i , a.ind);
  return true;
}

int main() {
  scanf("%d %lf %d\n", &N , &R , &W);
  rs = R*R;
  for (int i = 0 ; i < N; i++) {
    scanf("%lf %lf\n" , &v[i].x , &v[i].y);
    v[i].ind = i+1;
    v2[i] = v[i];
  }
  balance( tree , 0 , N-1 , 1 );
  inicio = 0;
  for (int i = 0 ; i < N; i++)
    if ( (v2[i].ind-inicio > W) && check( v2[i] ) )
      inicio = v2[i].ind+1;
  return 0;
}
```

Binary Indexed Tree

```
#define MAXVAL 131072    //MAXVAL deve ser da forma 2^k
int tree[MAXVAL+5];

int read(int x) {  //Ler frequência acumulada até ao index "idx" - O(log idx)
  int sum = 0;
  for ( sum = 0 ; x > 0 ; x -= (x & -x) )
    sum += tree[x];
  return sum;
}
void update(int x ,int val) { //Adicionar val ao index "idx" - O(log MAXVAL)
  for ( ; x < MAXVAL ; x += (x & -x) )
    tree[idx] += val;
}
int readsingle(int x) {  return read(x) - read(x-1);    }
```

```
int tree[MAXVAL+5][MAXVAL+5] , singleVal[MAXVAL+5][MAXVAL+5];

int read(int x, int y) {
  int sum = 0;
  for ( ; x > 0 ; x -= (x & -x) )
    for (int i = y ; i > 0 ; i -= (i & -i))
      sum+=tree[x][i];
  return sum;
}
void update(int x, int y ,int val) {
  for ( ; x < MAXVAL ; x += (x & -x) )
    for (int i = y ; i < MAXVAL ; i += (i & -i) )
      tree[x][i] += val;
}
```

Suffix Array

```
char MSG[MAX+1];
int LEN, H, D, id[16][MAX], suffix[MAX]; // 16 seria o log2(MAX)

bool cmp_suffix(int a, int b) {
  return id[D][a] == id[D][b] ? id[D][a+H] < id[D][b+H] : id[D][a] < id[D][b];
}
void build() {
  D = 0, LEN = strlen(MSG)+1;
  for(int i=0; i<LEN; i++)
    suffix[i] = i, id[D][i] = MSG[i];

  for(; D==0 || id[D][suffix[LEN-1]] != LEN-1; D++)  {
    H = (1<<D);
    sort( suffix, suffix+LEN, cmp_suffix );

    id[D+1][suffix[0]] = 0;
    for(int i=1; i<LEN; i++)
      id[D+1][suffix[i]] = id[D+1][suffix[i-1]] + cmp_suffix(suffix[i-1], suffix[i]);
  }
}
int lcp(int a, int b) {
  int prefix = 0;
  for(int i=D; i>=0; i--)
  if(id[i][a] == id[i][b])
    prefix += (1<<i), a += (1<<i), b += (1<<i);
  return prefix;
}
```

8. Computação Gráfica

Biblioteca de Geometria

Contém os algoritmos geométricos presentes na wiki, usando o struct Point uniforme.

```
#define INF   2000000000
#define EPS  1e-7
#define CP  const Point
typedef double CoordType;

struct Point {
  CoordType x , y ;
  Point( CoordType xx = 0 , CoordType yy = 0 ) { x = xx , y = yy; }

  Point operator + ( CP & a )  const  { return Point( x+a.x , y+a.y ); }
  Point operator - ( CP & a )  const  { return Point( x-a.x , y-a.y ); }
  Point operator * ( double n )  const  { return Point( x * n , y * n ); }
  Point operator / ( double n )  const  { return Point( x / n , y / n ); }
  CoordType operator * ( CP & a )  const  { return x * a.x + y * a.y;    } // dot
  CoordType operator ^ ( CP & a )  const  { return x * a.y - y * a.x;    } // cross

  bool operator < ( CP & p ) const {
    return x < p.x || ( fabs(x-p.x) < EPS && y < p.y );
  }
};
struct Linha {
  double a,b,c;  // ax + by + c = 0
  Point p[2];  // 2 pontos da recta
};

// 2D cross product. Return a positive value, if ABC makes a counter-clockwise turn,
// negative for clockwise turn, and zero if the points are collinear.
CoordType cross( CP & A, CP &B, CP &C)  { return (B-A) ^ (C-A);  }
CoordType sqrDist(CP & a, CP & b)  { return (b-a)*(b-a) ;     }
double distancia( CP & a , CP & b )   { return sqrt( (b-a)*(b-a) );  }
double distancia(CP & a)      { return sqrt( a*a );     }
Point unit(CP & p)        { return p/distancia(p);   } /* norma de vector */
Point rotateCCW(CP & a)       { return Point( -a.y , a.x ) ;   }

Linha points_to_line(Point &a , Point &b) {
  Linha ret;
  ret.p[0] = a;
  ret.p[1] = b;
  if ( fabs(a.x - b.x) < EPS) {
    ret.a = 1.0;
    ret.b = 0.0;
    ret.c = -(a.x);
  } else {
    ret.a = - ( (a.y - b.y) / (a.x - b.x) );
    ret.b = 1.0;
    ret.c = -(ret.a * a.x) - (ret.b * a.y);
  }
  return ret;
}
Linha rotate(Linha a) {  // Calcula a mediana de um segmento de recta. Contudo dá jeito saber
  Linha b;   // qual é a recta que passa por esse segmento de recta
  b.p[1] = b.p[0] = ( a.p[1] + a.p[0] ) / 2 ;

  b.a = a.b;
  b.b = -a.a;
  b.c = -(b.a * b.p[0].x) - (b.b * b.p[0].y);
  return b;
}
```

```
bool is_line_paralel(Linha &a , Linha &b){ return ( (fabs(a.a-b.a) < EPS) && (fabs(a.b-b.b) < EPS)); }

bool is_line_equal(Linha &a , Linha &b) { return ( is_line_paralel(a,b) && (fabs(a.c - b.c) < EPS)); }

bool intersects( Linha &a , Linha &b , Point &p) {
  if (is_line_equal(a,b))  {
    p = a.p[0];
    return true;
  }
  if (is_line_paralel(a,b))
    return false;
  p.x = (b.b * a.c - a.b *b.c) / (b.a * a.b - a.a * b.b);

  if ( fabs(a.b) > EPS )  p.y = -( (a.a * p.x + a.c) / a.b );
  else      p.y = -( (b.a * p.x + b.c) / b.b );
  return true;
}
double area( vector<Point> & P ){
  double a=0;
  for (size_t i = 1 ; i+1 < P.size() ; i++)
    a += cross( P[0] , P[i] , P[i+1] );
  return fabs(a/2);
}
//distancia do ponto p na direccao u(tem que ser unitario), a um ponto da recta que passa em l0 e l1
double rayLineInter(Point & p, Point & u, Point & l0, Point & l1) {
  Point t = l1-l0 , w = p - l0 , ld = unit(t);
  double mDist = w ^ ld;
  return -mDist / ( u ^ ld);
}
Point circle_center(Point & a, Point & b, Point & c) {
  Point m1 = (a+b) * 0.5 , m2 = (b+c)*0.5;
  Point d1 = unit(rotateCCW(b-a));
  Point d2 = m2 + rotateCCW(c-b);
  return m1+d1*rayLineInter(m1, d1, m2, d2);
}
//Compute the distance from AB to C
//if isSegment is true, AB is a segment, not a line.

double linePointDist( CP & A, CP & B , CP & C, bool isSegment = false){
  if(isSegment){            // verifica se C pertence a AB.
    CoordType dotp = (C-B)*(B-A);      // senao pertencer, retorna distancia
    if(dotp > 0) return distancia( B , C );    // a um dos extremos
    dotp = (C-A)*(A-B);
    if(dotp > 0) return distancia( A , C );
  }
  return fabs( cross( A , B , C ) / distancia( B , A ) );
}
// Returns a list of points on the convex hull in counter-clockwise order.
// Note: the last point in the returned list is the same as the first one.
// Note2: < inclui TODOS os pontos na fronteira, <= apenas inclui os vértices.
int convexHull(vector<Point> & P , vector<Point> & H) {
  int i , t, k = 0 , j = 0 , n = P.size();
  H.resize(2*n);
  sort( P.begin() , P.end() );  // Sort points lexicographically
  // Build lower hull
  for (i = 0; i < n; i++) {
    while (k > 1 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
  }
  j = k;      // index de separacao entre lower e upper hull
  // Build upper hull
  for (i = n-2, t = k; i >= 0; i--) {
    while (k > t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
  }
  H.resize(k);
  return j;
}
// usada apenas no Closest points devido ao operador < ter de ser definido em Y!
bool comparaX( CP & a , CP & b ) {  return a.x < b.x ; }

double ClosestPointsDistance( Point * p , int np ) {
  Point tmp;
  double   dist = INF;  tmp.x = - INF;
  set<Point> pontos;         // operador < de Point tem de ser definido em Y!!!
  set<Point>::iterator it;      // pois o set não aceita funções de comparação (acho eu)
  sort( p , p+np , comparaX );    // por isso é necessária esta função de comparação

  pontos.insert( p[0] );
  for ( int i = 1 , left = 0 ; i < np ; pontos.insert(p[i++]) ) {
    while ( p[i].x > p[left].x + dist )
      pontos.erase( p[left++] );
    tmp.y = p[i].y - dist;

    for (it = pontos.lower_bound( tmp ); it != pontos.end() && it->y < p[i].y+dist ; it++)
      dist = min( dist , distancia( p[i] , *it ) );
  }
  return dist;
}
double PolygonDiameter( vector<Point> & P ) {
  vector<Point> L;     // calcular o convex hull, i tem o indice de
  int i = convexHull( P , L );   // separacao entre o lower/upper hull
  reverse( L.begin()+i , L.end() ); // reverse dos pontos do upper hull (biblio algorithm)
  L.push_back( L[i-1] );     // duplicar ultimo ponto do lower hull
```

```
  CoordType dist = -1;
  int j = i - 1 , m = L.size()-1; // ver textos de apoio: i percorre o upper
  while ( i < m || j > 0 ) {  // e o j percorre o lower hull.
    dist = max( dist , sqrDist( L[i] , L[j] ) );
    if (i == m)   j--;
    else if (j == 0) i++;
    else if ((L[i+1].y-L[i].y)*(L[j].x-L[j-1].x) > (L[j].y-L[j-1].y)*(L[i+1].x-L[i].x))
      i++;
    else
      j--;
  }
  return sqrt(dist);
}
// cn_PnPoly(): crossing number test for a point in a polygon
int cn_PnPoly( Point P, Point* V, int n ) {
  int cn = 0;
  for (int i=0; i<n; i++)
    if (((V[i].y <= P.y) && (V[i+1].y > P.y)) || ((V[i].y > P.y) && (V[i+1].y <= P.y))) {
      double vt = (double)(P.y - V[i].y) / (V[i+1].y - V[i].y);
      if ( P.x < V[i].x + vt * (V[i+1].x - V[i].x) )
        ++cn;
    }
  return (cn&1);  // 0 if even (out), and 1 if odd (in)
}
```

Teorema de Pick

Dado um polígono "simples" (polígono que não contém buracos e é constituido por apenas uma peça) cujas coordenadas são inteiras, a sua área é dada por:
A = i + b/2 - 1
Onde **i** é o número de pontos de coordenadas inteiras no interior do polígono e **b** é o número de pontos de coordenadas inteiras ao longo do seu perímetro.

9. Jogos

Grundy Numbers

grundy(X) = menor grundy number onde não se consegue chegar

- multiple stack, e só pode retirar de uma stack de cada vez: losing position iff XOR grundy_numbers = 0
- multiple stack, e pode retirar entre 1 e N stacks: losing position iff all grundy_numbers = 0
- multiple stack, e têm que retirar de todas as stacks: losing position iff existir pelo menos 1 grundy_number = 0

http://en.wikipedia.org/wiki/Nimber
http://en.wikipedia.org/wiki/Sprague%E2%80%93Grundy_theorem

10. Knapsacks

0/1 Knapsack

Dados um conjunto de items, cada um com um valor e um custo, os items de cada tipo a incluir de forma a que o custo não ultrapasse um determinado valor e o valor seja o mais alto possível. Este problema é NP-completo.
Nesta variante cada elemento só pode ser usado ou não usado.
A solução utiliza programação dinâmica ao guardar a melhor solução encontrada para cada espaço disponível.

```
#define MAXOBJ  1001  // num maximo de objectos
#define MAXW  31  // valor maximo da capacidade da bag

int W[ MAXOBJ ] , P[ MAXOBJ] ;  // Peso e preco de cada objecto
int C[ MAXOBJ ][ MAXW ];  // matriz de PD para o knapsack

int knapsack01( int N , int MW ) {
  int i , j ;
  for (i = 0 ; i <= N ; i++)  C[i][0] = 0;
  for (i = 0 ; i <= MW ; i++)  C[0][i] = 0;

  for (i=1; i <= N ; i++)
  {
    for (j=1; j <= MW ; j++)
    {
      if (W[i] > j)   C[i][j] = C[i-1][j];
      else    C[i][j] = max( C[i-1][j] , C[i-1][ j-W[i] ] + P[i] );
    }
  }
  return C[N][MW];
}
```

```
int main(){
  int i , N , MW;
  scanf("%d %d",&N , &MW);    // num de objectos e maximo da bag
  for (i=1 ; i <= N ; i++)    // indice nos arrays começa em 1 !!
    scanf("%d %d",&P[i] , &W[i]);  // ler preco , peso

  printf("%d\n", knapsack01( N , MW ) );
  return 0;
}
```

11. Java

```
try {
  BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
  String str = "";
  while (str != null) {
   str = in.readLine();
   process(str);
  }
} catch (IOException e) {}
```

```
StringTokenizer st = new StringTokenizer(line); // Precisa ser testado
while (st.hasMoreTokens()){
```

```
 int i = new Integer(st.nextToken()).intValue();
}
```

```
class Main {
  public static void main (String [] args) {
   BigInteger b[] = new BigInteger[101];
   b[0] = new BigInteger("1");
   Scanner sc = new Scanner (System.in);
   int n , t = sc.nextInt ();
   for ( n = 1 ; n<= 100 ; n++ ) {
     Integer i = new Integer( n );
     b[n] = b[n - 1].multiply( new BigInteger( i.toString() ));
   }
   while (t-- > 0) {
     n = sc.nextInt();
     System.out.println( b[n].toString() );
   }  } }
```