1 Algorithmes sur les graphes

```cpp
/*****************************************************************************
* Algorithme de Dijkstra
*
* Recherche de plus court chemin entre une source et tous les sommets du graphe.
* Restriction : les aretes doivent avoir des poids positifs.
* Complexite : temporelle O(m*log(n)), spatiale O(m)
*****************************************************************************/

#include <vector>
#include <queue>
#include <climits>
using namespace std;

struct edge {
  int x, y, w;
  edge(void): x(),y(),w() {};
  edge(int a, int b, int c): x(a), y(b), w(c) {}
  bool operator< (const edge & e) const {
    // Attention, on met un ">" car le top d'une file de priorite est le plus grand element
    // pour la relation consideree (et donc ici une arete de poids minimum).
    return w > e.w;
  }
};
typedef vector<vector<edge> > graph;

void dijsktra(graph & g, vector<int> & pere, vector<int> & dist, int source) {
  priority_queue<edge> file;
  vector<bool> vu (g.size(), false);
  edge e,f;
  int i = 0;
  dist.clear();
  dist.resize(g.size(), INT_MAX);
  file.push(edge(source,source,0));
  while (!file.empty()) {
    e = file.top();
    file.pop();
    if (vu[e.y]) continue;
    // La distance de source a e.y est la bonne
    pere[e.y] = e.x;
    dist[e.y] = e.w;
    vu[e.y] = true;
    // Parcours des voisins de e.y
    for (i = 0; i < (int) g[e.y].size(); ++i) {
      f = g[e.y][i];
      f.w += e.w;
      if (dist[f.y] > f.w)
        file.push(f);
    }
  }
  return;
}
```

```cpp
/*****************************************************************************
 * Algorithme de Bellman-Ford
 *
 * Recherche de plus court chemin entre une source et tous les sommets du graphe.
 * Detection de circuits absorbants, poids negatifs possibles.
 * Complexite : O(n*m)
 *****************************************************************************/

#include <vector>
#include <climits>
using namespace std;

struct edge {
  int x, y, w;
  edge(void) :
    x(), y(), w() {}
  edge(int a, int b, int c) :
    x(a), y(b), w(c) {}
  bool operator<(const edge & e) const {
    return w < e.w;
  }
};

typedef vector<edge> edges;

// Renvoie true ssi il y a un circuit absorbant
// Tableau des predecesseurs sur un chemin de la source a i --> pred
bool bellman(edges & e, vector<int> & d, vector<int> pred, int borne, int source) {
  int i, k = 0;
  bool changed = true;
  d[source] = 0;
  for (k = 0; changed && k < borne; ++k) {
    changed = false;
    for (i = 0; i < (int) e.size(); ++i) {
      if (d[e[i].x] != INT_MAX && d[e[i].y] > d[e[i].x] + e[i].w) {
        d[e[i].y] = d[e[i].x] + e[i].w;
        pred[e[i].y] = e[i].x;
        changed = true;
      }
    }
  }
  return changed;
}
```

```cpp
/*****************************************************************************
 * Algorithme de Roy-Warshall
 *
 * Recherche de plus court chemin entre tout couple de sommets.
 * Detection de circuits absorbants, poids negatifs possibles.
 * Complexite : O(n^3)
 *****************************************************************************/

#include <vector>
#include <climits>
using namespace std;

typedef vector<vector<int> > matrix;

// Renvoie true ssi il y a un circuit absorbant
// Matrice des successeurs sur un chemin de i a j --> r
bool roy_warshall(matrix & g, matrix & r) {
  int i, j, k, n = g.size();
  // Initialisation de la table de routage
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      r[i][j] = (g[i][j] == INT_MAX ? i : j);
  // Algorithme principal
  for (k = 0; k < n; ++k)
    for (i = 0; i < n; ++i)
      for (j = 0; j < n; ++j) {
        if (g[i][k] == INT_MAX) continue;
        if (g[k][j] == INT_MAX) continue;
        if (g[i][k] + g[k][j] >= g[i][j]) continue;
        g[i][j] = g[i][k] + g[k][j];
        r[i][j] = r[i][k];
      }
  // Detection des circuits absorbant
  for (i = 0; i < n; ++i)
    if (g[i][i] < 0)
      return true;
  return false;
}
```

```cpp
/*****************************************************************************
 * Union-Find ; Algorithme de Kruskal
 *
 * Recherche d'arbre couvrant de poids minimum.
 *****************************************************************************/
#include <vector>
#include <algorithm>
using namespace std;
// Structure d'union-find
vector<int> pere;

void init(int n) {
  int i;
  pere.resize(n);
  for (i = 0; i < n; ++i)
    pere[i] = i;
  return;
}
int find(int i) {
  if (pere[i] == i) return i;
  pere[i] = find(pere[i]);
  return pere[i];
}
void merge(int i, int j) {
  pere[find(i)] = find(j);
  return;
}

// Algorithme de Kruskal
struct edge {
  int x, y, w;
  edge(void) :
    x(), y(), w() {}
  edge(int a, int b, int c) :
    x(a), y(b), w(c) {}
  bool operator<(const edge & e) const {
    return w < e.w;
  }
};
typedef vector<edge> tree;
typedef vector<edge> adj;

void kruskal(adj & e, tree & t, int n) {
  int i;
  init(n);
  sort(e.begin(), e.end());
  for (i = 0; i < (int) e.size(); ++i) {
    if (find(e[i].x) != find(e[i].y)) {
      t.push_back(e[i]);
      merge(e[i].x, e[i].y);
    }
  }
}
```

```cpp
/*****************************************************************************
 * Algorithme de Prim
 *
 * Recherche d'arbre couvrant de poids maximum, version avec des priority_queue.
 *****************************************************************************/

#include <vector>
#include <queue>
#include <climits>
using namespace std;

struct edge {
  int x, y, w;
  edge(void): x(),y(),w() {};
  edge(int a, int b, int c): x(a), y(b), w(c) {}
  bool operator< (const edge & e) const {
    return w < e.w; // Attention, changer "<" en ">" pour obtenir un arbre couvrant de poids
        minimal.
  }
};

typedef vector<vector<edge> > graph;
typedef vector<edge> tree;

// Retourne le poids d'un arbre couvrant de poids maximal + la liste des aretes d'un tel arbre
int prim(graph & g, tree & t) {
  priority_queue<edge> file;
  vector<bool> vu (g.size(), false);
  edge e;
  int i,s = 0;
  vu[0] = true;
  for (i = 0; i < (int) g[0].size(); ++i)
    file.push(g[0][i]);
  while (!file.empty()) {
    e = file.top();
    file.pop();
    if (vu[e.y]) continue;
    // Ajout de l'arete "e"
    t.push_back(e);
    s += e.w;
    vu[e.y] = true;
    // Parcours des voisins de "y"
    for (i = 0; i < (int) g[e.y].size(); ++i)
      file.push(g[e.y][i]);
  }
  return s;
}
```

```cpp
/*****************************************************************************
 * Algorithme d'Edmonds-Karp
 *
 * Recherche d'un flot maximum. Complexite : O(n*m^2)
 *****************************************************************************/

#include <vector>
#include <queue>
using namespace std;

class graph {
public:
  vector<vector<int> > adj; // Liste d'adjacence
  vector<vector<int> > capa; // Matrice des capacites de liens
  vector<vector<int> > flot; // Matrice des valeurs du flot
  graph(void) {}
  graph(int n) {
    adj.resize(n);
    capa.resize(n, vector<int> (n, 0));
    flot.resize(n, vector<int> (n, 0));
  }
  void reset(void) {
    flot.clear();
    flot.resize(adj.size(), vector<int> (adj.size(), 0));
  }
private:
  vector<int> pere;

  int cout(int x, int y) {
    return capa[x][y] - flot[x][y];
  }

  // Parcours en largeur conditionnel
  void parcours(int source, int target) {
    int x, y, i;
    queue<int> file;
    pere.clear();
    pere.resize(adj.size(), -1);
    file.push(source);
    pere[source] = source;
    while (!file.empty()) {
      x = file.front();
      file.pop();
      for (i = 0; i < (int) adj[x].size(); ++i) {
        y = adj[x][i];
        if (pere[y] != -1)
          continue;
        if (cout(x, y) <= 0)
          continue;
        pere[y] = x;
        file.push(y);
        if (y == target)
          return;
```

```
      }
    }
    return;
  }

  // Calcul du cout residuel le long d'un chemin
  int cout_residuel(int target) {
    int x, y, c;
    y = target;
    x = pere[target];
    c = cout(x, y);
    while (pere[x] != x) {
      y = x;
      x = pere[x];
      c = min(c, cout(x, y));
    }
    return c;
  }

  // Augmenter le flot le long du chemin
  void augmenter(int target, int c) {
    int x, y;
    y = target;
    x = pere[target];
    while (x != y) {
      flot[x][y] += c;
      flot[y][x] -= c;
      y = x;
      x = pere[x];
    }
    return;
  }
public:
  // Calcul d'un flot maximum
  int flot_max(int source, int target) {
    int c, s = 0;
    while (true) {
      parcours(source, target);
      if (pere[target] == -1)
        break;
      c = cout_residuel(target);
      s += c;
      augmenter(target, c);
    }
    return s;
  }

};
```

```cpp
/*****************************************************************************
 * Couplage maximum.
 *
 * Recherche d'un couplage maximum dans le cas d'un graphe biparti.
 * Complexite : O(n*m)
 *****************************************************************************/

#include <vector>
using namespace std;

class bigraph {
public:
  vector<vector<int> > adj_left;
  vector<int> match_left, match_right;
  int nb_coupled_edges;
  bigraph(void) {
  }
  bigraph(int p, int q) {
    adj_left.resize(p);
    match_left.resize(p, -1);
    match_right.resize(q, -1);
    nb_coupled_edges = 0;
  }
  void reset(void) {
    int p = match_left.size();
    int q = match_right.size();
    // Clear
    match_left.clear();
    match_right.clear();
    // Resize
    match_left.resize(p, -1);
    match_right.resize(q, -1);
    nb_coupled_edges = 0;
  }

private:
  vector<bool> vu_left;

  // Parcours en profondeur pour trouver un chemin augmentant
  bool parcours(int x) {
    int i, y, z;
    if (x == -1)
      return true; // Sommet non sature : on s'arrete
    if (vu_left[x])
      return false; // Sommet deja explore
    vu_left[x] = true;
    for (i = 0; i < (int) adj_left[x].size(); ++i) {
      y = adj_left[x][i];
      z = match_right[y];
      if (parcours(z)) {
        match_right[y] = x;
        match_left[x] = y;
        return true;
```

```
      }
    }
    return false;
  }

  // Recherche d'un chemin augmentant
  bool find_path(void) {
    int i;
    vu_left.clear();
    vu_left.resize(adj_left.size(), false);
    for (i = 0; i < (int) adj_left.size(); ++i) {
      if (match_left[i] == -1 && parcours(i))
        return true;
    }
    return false;
  }

public:
  // Couplage glouton
  void greedy_matching(void) {
    int i, j, k;
    for (i = 0; i < (int) adj_left.size(); ++i) {
      for (k = 0; k < (int) adj_left[i].size(); ++k) {
        j = adj_left[i][k];
        if (match_left[i] == -1 && match_right[j] == -1) {
          match_left[i] = j;
          match_right[j] = i;
          ++nb_coupled_edges;
        }
      }
    }
  }

  // Calcul d'un couplage maximum
  void couplage_max(void) {
    while (true) {
      if (find_path())
        ++nb_coupled_edges;
      else
        break;
    }
    return;
  }

};
```

```cpp
/*****************************************************************************
 * Parcours en largeur
 *
 * Implementation a l'aide de deux piles. Complexite : O(n + m)
 *****************************************************************************/

#include <vector>
#include <stack>
using namespace std;

typedef vector<vector<int> > graph;

// Renvoie la distance de la source a la cible en nombre de sauts
// L'algorithme s'arrete des que la cible est atteinte
int bfs(graph & g, int source, int target) {
  vector<bool> vu(g.size(), false);
  stack<int> current, next;
  int i, j, d = 0;
  current.push(source);
  vu[source] = true;
  while (not current.empty()) {
    i = current.top();
    current.pop();
    for (j = 0; j < (int) g[i].size(); ++j) {
      if (vu[g[i][j]]) continue;
      if (g[i][j] == target) return d;
      next.push(g[i][j]);
      vu[g[i][j]] = true;
    }
    if (current.empty()) {swap(current, next); ++d;}
  }
  return -1;
}

/*****************************************************************************
 * Parcours en profondeur
 *
 * Implementation a l'aide d'une pile. Complexite : O(n + m)
 *****************************************************************************/

#include <vector>
#include <stack>
using namespace std;

typedef vector<vector<int> > graph;

// Renvoie true ssi la source et la cible sont dans la meme composante connexe
bool dfs(graph & g, int source, int target) {
  vector<bool> vu(g.size(), false);
  stack<int> current;
  int i, j;
  current.push(source);
  vu[source] = true;
```

```
  while (not current.empty()) {
    i = current.top();
    current.pop();
    for (j = 0; j < (int) g[i].size(); ++j) {
      if (vu[g[i][j]]) continue;
      if (g[i][j] == target) return true;
      current.push(g[i][j]);
      vu[g[i][j]] = true;
    }
  }
  return false;
}
```

```
/*****************************************************************************
 * Composantes fortement connexes
 *
 * Calcul des composantes fortement connexes d'un graphe oriente.
 * Algorithme de Tarjan. Complexite : O(n + m)
 *****************************************************************************/
class graph {
public:
  vector<vector<int> > adj;
  vector<int> comp; // Indice de la CFC d'un sommet
  int nb_cfc;

private:
  vector<int> index; // Ordre dans lequel les sommets ont ete visites
  vector<int> lowlink; // Sert a determiner la "racine" d'une CFC
  stack<int> s; // Les sommets non classes
  int id;

  // Parcours en profondeur sur un sommet
  void tarjan(int i) {
    int j, k;
    index[i] = id;
    lowlink[i] = id;
    ++id;
    s.push(i);
    for (k = 0; k < (int) adj[i].size(); ++k) {
      j = adj[i][k];
      if (index[j] == -1)
        tarjan(j);
      if (comp[j] == -1)
        lowlink[i] = min(lowlink[i], lowlink[j]);
    }
    if (lowlink[i] == index[i]) {
      do {
        j = s.top();
        s.pop();
        comp[j] = nb_cfc;
      } while (i != j);
      ++nb_cfc;
    }
  }
```

```
42
43  public:
44    // Calcul des CFC du graphe
45    void cfc() {
46      id = 0;
47      nb_cfc = 0;
48      comp.clear();
49      index.clear();
50      lowlink.clear();
51      comp.resize(adj.size(), -1);
52      index.resize(adj.size(), -1);
53      lowlink.resize(adj.size(), -1);
54      for (int i = 0; i < (int) adj.size(); ++i) {
55        if (index[i] == -1)
56          tarjan(i);
57      }
58    }
59  };
```

```cpp
/*****************************************************************************
 * Effeuillage de graphe
 *
 * Effeuillage de graphe oriente, on enleve les sommets de degre entrant egal a 1
 * Complexite : O(n+m)
 *****************************************************************************/

#include <vector>
#include <list>
using namespace std;

void effeuille(vector<vector<int> > & adj, vector<int> & order) {
  int i, j, k, index = 0;
  vector<bool> vu(adj.size(), false);
  vector<int> deg(adj.size(), 0);
  list<int> visit;
  // On calcule le degre entrant de chaque sommet
  for (i = 0; i < (int) adj.size(); ++i)
    for (k = 0; k < (int) adj[i].size(); ++k)
      ++deg[adj[i][k]];
  // On commence par les sommets de degre entrant nul
  for (i = 0; i < (int) adj.size(); ++i)
    if (deg[i] == 1) {
      vu[i] = true;
      visit.push_back(i);
    }
  // On effeuille
  while (not visit.empty()) {
    i = visit.front();
    visit.pop_front();
    // Appliquer une operation sur "i"
    order[i] = index;
    ++index;
    for (k = 0; k < (int) adj[i].size(); ++k) {
      j = adj[i][k];
      if (vu[j]) continue; // Sommet deja visite
      // On retire "i", donc on decremente le degre entrant de "j"
      --deg[j];
      // Si "j" devient une feuille
      if (deg[j] == 1) {
        vu[j] = true;
        visit.push_back(j);
      }
    }
  }
  return;
}
```

2 Arithmétique

```
// Exponentiation modulaire rapide : y = (x ^ n) % m
int mod_pow(int x, int n, int m) {
  int y = 1;
  while (n != 0) {
    if ((n & 1) == 1)
      y = (y * x) % m;
    x = (x * x) % m;
    n = n >> 1;
  }
  return y;
}

// Si p est premier, renvoie y tel que x*y = 1 [p].
// Utilise le petit theoreme de fermat qui dit que x^(p-1) = 1 [p] si p premier.
int inv_mod(int x, int p) {
  return mod_pow(x, p - 2, p);
}

// Remplace un couple (x,y) par (px,py) premiers entre eux, avec m = pgcd(x,y)
void normalize(int &x, int &y) {
  int m = x, n = y;
  while (n) {
    m %= n;
    swap(m, n);
  }
  x /= m;
  y /= m;
  return;
}

// Calcule (u,v) tel que a*u + b*v = r, et renvoie r = pgcd(a,b).
int bezout(int &a, int &b, int &u, int &v) {
  int r = a, r1 = b;
  int q, rs, us, vs;
  int u1, v1;
  u  = 1, v = 0;
  u1 = 0, v1 = 1;
  while (r1) {
    q = r / r1;
    rs = r; us = u; vs = v;
    r = r1; u = u1; v = v1;
    r1 = rs - q * r1;
    u1 = us - q * u1;
    v1 = vs - q * v1;
  }
  return r;
}
```

```
// Coefficients binomiaux
int binom(int k, int n) {
  int i, res = 1;
  for (i = 1; i <= k; ++i)
    res = (n - k + i) * res / i;
  return res;
}

// Test de primalite
bool isprime(int n) {
  if (n == 0 || n == 1)
    return false;
  else if (n == 2 || n == 3 || n == 5 || n == 7)
    return true;
  else if (n % 2 == 0 || n % 3 == 0)
    return false;
  else {
    for (int i = 5; i * i <= n; i += 6)
      if (n % i == 0 || n % (i + 2) == 0)
        return false;
    return true;
  }
}

/*****************************************************************************
 * Crible d'Eratosthenes
 *
 * Et test de primalite avec precalcul des nombres premiers.
 *****************************************************************************/

#include <vector>
using namespace std;

// Crible d'Eratosthenes
void sieve(vector<bool> & crossed, int limit) {
  int i,j;
  crossed.clear();
  crossed.resize(limit / 2, false);
  crossed[0] = true;
  for (i = 1; 4*i*i < limit; ++i) {
    if (crossed[i]) continue;
    for (j = 2*i*(i+1); 2*j < limit; j += 2*i+1)
      crossed[j] = true;
  }
  return;
}

// Test de primalite
bool isprime(vector<bool> & crossed, int i) {
  if (i == 2) return true;
  if (i % 2 == 0) return false;
  return not crossed[(i-1)/2];
}
```

```cpp
/*****************************************************************************
 * Pivot de Gauss
 *
 * Calcule le determinant d'une matrice carree.
 * Calcule l'inverse d'une matrice carree.
 *****************************************************************************/

#include <vector>
#include <cmath>
using namespace std;

typedef long double elem;
typedef vector<vector<elem> > matrix;

elem inline eabs(elem x) {
  return fabs(x);
}

bool inline zero(elem x) {
  return (eabs(x) < 1e-20);
}

// Echange de deux lignes de la matrice
void swap_line(matrix & m, int i, int j) {
  swap(m[i], m[j]);
  return;
}

// Effectue M[i] <-- M[i] - M[j] * y/x
void sub_line(matrix & m, int i, int j, elem x, elem y) {
  int k;
  for (k = 0; k < (int) m.size(); ++k)
    m[i][k] = m[i][k] - m[j][k] * y / x;
  return;
}

// Effectue M[i] <-- M[i] * x
void mult_line(matrix & m, int i, elem x) {
  int k;
  for (k = 0; k < (int) m.size(); ++k)
    m[i][k] = m[i][k] * x;
  return;
}

elem det(matrix & m) {
  int i, j, n;
  elem d = 1;
  n = m.size();
  for (i = 0; i < n; ++i) {
    // Recherche du plus grand element non nul de la colonne i (entre les lignes i et n-1)
    for (j = i + 1; j < n; ++j) {
      if (zero(m[j][i]))
        continue;
```

```
54      if (eabs(m[j][i]) > eabs(m[i][i]))
55        swap_line(m, i, j);
56    }
57    // S'il n'y a pas de nombre non nul sur la diagonale, le determinant est nul
58    if (zero(m[i][i]))
59      return 0;
60    d *= m[i][i];
61    // Sinon, soustraire la ligne i aux lignes suivantes
62    for (j = i + 1; j < n; ++j) {
63      sub_line(m, j, i, m[i][i], m[j][i]);
64    }
65  }
66  // Le determinant est alors le produit des elements diagonaux
67  return d;
68 }
69
70 // Inverse la matrice m
71 // Stocke le resultat dans inv, renvoie true ssi m est inversible
72 bool inverse(matrix & m, matrix & inv) {
73  int i, j, n;
74  n = m.size();
75  inv.clear();
76  inv.resize(n, vector<elem> (n, 0));
77  // Au debut inv = la matrice identite
78  for (i = 0; i < n; ++i)
79    inv[i][i] = 1.;
80  // On diagonalise la matrice
81  for (i = 0; i < n; ++i) {
82    // Recherche du plus grand element non nul de la colonne i (entre les lignes i et n-1)
83    for (j = i + 1; j < n; ++j) {
84      if (zero(m[j][i]))
85        continue;
86      if (eabs(m[j][i]) > eabs(m[i][i])) {
87        swap_line(inv, i, j);
88        swap_line(m, i, j);
89      }
90    }
91    // S'il n'y a pas de nombre non nul sur la diagonale, m n'est pas inversible
92    if (zero(m[i][i]))
93      return false;
94    // Sinon, soustraire la ligne i aux autres lignes
95    for (j = 0; j < n; ++j) {
96      if (i != j) {
97        sub_line(inv, j, i, m[i][i], m[j][i]);
98        sub_line(m, j, i, m[i][i], m[j][i]);
99      }
100   }
101   // Normalisation
102   mult_line(inv, i, 1. / m[i][i]);
103   mult_line(m, i, 1. / m[i][i]);
104 }
105 return true;
106 }
```

3 Géométrie

```cpp
#include <vector>
#include <complex>
using namespace std;

typedef complex<double> point;
typedef vector<point> poly;
point e(1000, 1000); // Un point a l'exterieur de la zone de travail


double inline det(const point & u, const point & v) {
  return imag(conj(u) * v);
}



// Teste si les points (a,b,c,d) sont cocycliques
bool cocycle(const point & a, const point & b, const point & c, const point & d) {
  // Vérifie que les points ne soient pas alignés
  if (det(c - a, c - b) == 0 || det(d - a, d - b) == 0)
    return false;
  return imag(((a - c) * (b - d)) / ((b - c) * (a - d))) == 0;
}


// Teste si les segments [a,b] et [c,d] s'intersectent
bool inline intersect(const point & a, const point & b, const point & c,
    const point & d) {
  double z = det(b - a, d - c);
  double x = det(c - a, b - a);
  double y = det(d - c, a - c);
  int sx = (x > 0 ? 1 : -1);
  int sy = (y > 0 ? 1 : -1);
  // On fait le test en n'utilsant que des entiers si possible
  return (z != 0 && x * z >= 0 && sx * x < sx * z && y * z >= 0 && sy * y
      < sy * z);
}


// Calcul le point d'intersection de deux droites
// Renvoie "true" si et seulement si le point d'intersection appartient aux deux segments (?)
bool segInter(const point & a, const point & b, const point & c,
    const point & d, point & inter) {
  double detSeg = det(b - a, d - c);
  double detABC = det(b - a, c - a);
  if (detSeg == 0)
    return false;
  inter = c - (d - c) * detABC / detSeg;
  return det(b - a, c - a) * det(b - a, d - a) <= 0 && det(d - c, a - c)
      * det(d - c, b - c) <= 0;
}


```

```
53 // Teste si le point a est a l'interieur du poligone p
54 bool is_inside(const point & a, const poly & p) {
55   int i, n = p.size();
56   bool tmp, b = false;
57   for (i = 0; i < (int) p.size(); ++i) {
58     tmp = intersect(a, e, p[i], p[(i + 1) % n]);
59     b = (b != tmp);
60   }
61   return b;
62 }
63
64
65 // Renvoie l'aire du poligone t
66 double get_surface(poly & t) {
67   double sum = 0;
68   int i;
69   for (i = 1; i + 1 < (int) t.size(); ++i) {
70     sum += det(t[i] - t[0], t[i + 1] - t[0]);
71   }
72   return abs(sum / 2);
73 }
```

```cpp
/*****************************************************************************
 * Algorithme de Graham
 *
 * Calcul de l'enveloppe convexe d'un ensemble de points.
 * Complexite : O(n*log(n))
 *****************************************************************************/
#include <vector>
#include <complex>
#include <algorithm>
using namespace std;

typedef complex<double> point;
typedef vector<point> poly;

double inline det(const point & u, const point & v) {
  return imag(conj(u) * v);
}

// p0 is assumed to be the leftmost point of the poly, and should be kept at pos. 0 in the vector
struct Compare {
  point p0;
  bool operator ()(const point & p1, const point & p2) {
    double d = det(p1 - p0, p2 - p0);
    if (p1 == p0) return true;
    if (p2 == p0) return false;
    return (d < 0 || (d == 0 && abs(p1 - p0) < abs(p2 - p0)));
  }
};
bool inline angle(point & a, point & b, point & c) {
  return (det(b - a, c - a) >= 0);
}

// Compute the convex hull of the set of point t, store it in r.
// Does not preserve the ordering of t's vertices
void convex_hull(poly & t, poly & r) {
  int i;
  Compare order;
  // Search leftmost vertex
  order.p0 = t[0];
  for (i = 1; i < (int) t.size(); ++i)
    if (t[i].real() < order.p0.real())
      order.p0 = t[i];
  sort(t.begin(), t.end(), order);
  for (i = 0; i < (int) t.size(); ++i) {
    r.push_back(t[i]);
    // Pop vertices that become internal
    while (r.size() > 3u && angle(r.end()[-3], r.end()[-2], r.end()[-1])) {
      r.end()[-2] = r.back();
      r.pop_back();
    }
  }
  return;
}
```

4 Annexe

```cpp
/*****************************************************************************
 * Algorithme de Dijkstra
 * Implementation avec des set
 *
 * Recherche de plus court chemin entre une source et tous les sommets du graphe.
 * Restriction : les aretes doivent avoir des poids positifs.
 * Complexite : temporelle O(m*log(n)), spatiale O(n)
 *****************************************************************************/
#include <vector>
#include <set>
#include <climits>
using namespace std;

struct edge {
  int to;
  int weight;
  edge(void) :
    to(), weight() {}
  edge(int a, int b) :
    to(a), weight(b) {}
};
typedef vector<vector<edge> > graph;

// pred[i] = predecesseur de "i" dans un plus court chemin de "source" a "i"
// dist[i] = longueur d'un tel plus court chemin
void dijsktra(graph & g, vector<int> & pred, vector<int> & dist, int source) {
  set<pair<int, int> > Q;
  int i, j, k, cost;
  dist.clear();
  dist.resize(g.size(), INT_MAX);
  pred.resize(g.size(), -1);
  dist[source] = 0;
  pred[source] = source;
  Q.insert(make_pair(0, source));
  while (not Q.empty()) {
    i = Q.begin()->second;
    Q.erase(Q.begin());
    for (k = 0; k < (int) g[i].size(); ++k) {
      j = g[i][k].to;      // Voisin de "i"
      cost = g[i][k].weight; // Poids de l'arete (i,j)
      if (dist[j] > dist[i] + cost) {
        // Mise a jour de l'element "j" en temps log(n)
        Q.erase(make_pair(dist[j], j));
        dist[j] = dist[i] + cost;
        pred[j] = i;
        Q.insert(make_pair(dist[j], j));
      }
    }
  }
  return;
}
```

```cpp
/*****************************************************************************
 * Algorithme de Prim
 * Implementation avec des set
 * Recherche d'arbre couvrant de poids minimum.
 *****************************************************************************/
#include <vector>
#include <set>
using namespace std;

struct edge {
  int from;
  int to;
  int weight;
  edge(void) :
    from(), to(), weight() {}
  edge(int a, int b, int c) :
    from(a), to(b), weight(c) {}
  bool operator<(const edge & e) const {
    return weight < e.weight || (weight == e.weight && (from < e.from
        || (from == e.from && to < e.to)));
  }
};
typedef vector<vector<edge> > graph;
typedef vector<edge> tree;

// Retourne le poids d'un arbre couvrant de poids minimal + la liste des aretes d'un tel arbre
int prim(graph & g, tree & t) {
  set<edge> Q;
  vector<set<edge>::iterator> old (g.size(), Q.end());
  vector<bool> vu (g.size(), false);
  int i, j, k, s = 0;
  vu[0] = true;
  for (i = 0; i < (int) g[0].size(); ++i) {
    old[g[0][i].to] = Q.insert(g[0][i]).first;
  }
  while (not Q.empty()) {
    t.push_back(*Q.begin());
    i = Q.begin()->to;
    s += Q.begin()->weight;
    vu[i] = true;
    Q.erase(Q.begin());
    for (k = 0; k < (int) g[i].size(); ++k) {
      j = g[i][k].to;
      if (not vu[j] && (old[j] == Q.end() || old[j]->weight > g[i][k].weight)) {
        // Mise a jour de l'element "j" en temps log(n)
        if (old[j] != Q.end())
          Q.erase(old[j]);
        old[j] = Q.insert(g[i][k]).first;
      }
    }
  }
  return s;
}
```

```cpp
/*****************************************************************************
 * Nombre d'arbres couvrants d'un graphe
 *
 * Soit A la matrice d'adjacence du graphe (avec des A[i][j] = 1 ssi (i,j) est une arete),
 * D la matrice diagonale contenant les degres des sommets (D[i][i] = deg(i) et 0 ailleurs).
 * On note alors Q = D-A.
 *
 * On peut montrer que le nombre d'arbres couvrants de G est alors la valeur absolue du
 * determinant d'un des cofacteurs de Q (obtenu en supprimant une ligne et une colonne de Q).
 *
 * L'algorithme utilise pour le calcul du determinant est celui du pivot de Gauss :
 *
 *  Pour i = 0 a n-1 faire
 *      Trouver une ligne j telle que Q[j][i] != 0
 *      Inverser les lignes i et j
 *      Pour j = i+1 a n-1 faire
 *          Soit x = Q[j][i] / Q[i][i]
 *          Faire l'operation Q[j] Q[j] - x * Q[i]
 *      Fin pour
 *  Fin pour
 *  Renvoyer le produit des elements diagonaux de Q
 *****************************************************************************/

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

typedef long double elem;
typedef vector<vector<elem> > matrix;

elem inline eabs(elem x) {
  return fabs(x);
}


int main(void) {
  matrix m;
  int p, k, i, j, n;
  while (cin >> n >> p >> k) {
    // Reinitialiser m.
    m.clear();
    m.resize(n, vector<elem> (n, -1));
    // Au depart m contient des -1 partout sauf sur la diagonale
    for (i = 0; i < n; ++i)
      m[i][i] = n - 1;
    for (; p; --p) {
      cin >> i >> j;
      --i; --j;
      if (m[i][j] == 0) continue;
      // Quand on rencontre une arete, on met la case correspondante a 0
      m[i][j] = 0;
      m[j][i] = 0;
```

```
      // Et on decremente le degre de chaque sommet incident
      --m[i][i];
      --m[j][j];
    }
    // On enleve une ligne/colonne a la matrice : la premiere par exemple
    for (i = 1; i < n; ++i) {
      m[i][0] = 0;
      m[0][i] = 0;
    }
    m[0][0] = 1;
    // On renvoie le determinant calcule (det = fonction qui calcule le determinant !!)
    cout << (long long) roundl(eabs(det(m))) << endl;
  }
  return 0;
}
```

5 Programmation dynamique

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

typedef vector<vector<int> > matrix;

// Calcul de la plus longue sous-sequence commune par programmation dynamique
int lcs(string & s1, string & s2) {
  int i, j;
  int n = s1.length();
  int m = s2.length();
  matrix v(n + 1, vector<int> (m + 1, 0));
  for (i = 1; i <= n; i++) {
    for (j = 1; j <= m; j++) {
      if (s1[i - 1] == s2[j - 1])
        v[i][j] = 1 + v[i - 1][j - 1];
      else
        v[i][j] = max(v[i][j - 1], v[i - 1][j]);
    }
  }
  return v[n][m];
}
```

6 Changer le layout clavier

```
#!/bin/bash

setxkbmap us
xmodmap -e "keycode 66 = Escape"
xmodmap -e "remove lock = Escape"


#!/bin/bash

setxkbmap fr -variant oss
xmodmap -e "keycode 66 = Escape"
xmodmap -e "remove lock = Escape"
```

7 Fichier de configuration `vimrc`

```
set noexpandtab
set tabstop=3
set softtabstop=3
set shiftwidth=3
set background="dark"
set textwidth=0
set autoindent
set hlsearch
```