```cpp
/** Auteur : Esseiva Nicolas | Date : 03.12.2019 | version : 1.0 **/
#include <iostream>
#include <vector>
#include <ctime>
#include <fstream>
#include <random>
#include <iomanip>
#include <algorithm>

using namespace std;

constexpr double min_delta_p = 0.000001;
constexpr double max_error = 0.01;

// Définition de types récurrents
typedef vector<vector<bool>> matrice;
typedef vector<bool> ligne;

// Type Point : utilisé pour le mode c pour créer une liste de paire (p,p')
struct Point {
    double p;
    double pPrime;

    Point(double p, double pPrime) {
        this->p = p;
        this->pPrime = pPrime;
    }
};

// Lecture de la grille et retourne n ou -1 si n <= 0
int read_grid(matrice &grid);
// Génération d'une grille lines x columns (lines et columns >= 0)
void generate_grid(matrice &grid, int lines, int columns,
                   bernoulli_distribution &b, default_random_engine &e);

// Affiche dans la console la grille au format pbm
void output_grid_pbm(const matrice &grid);

// Résolution du passage par méthode récursive
bool resoudre_passage(const matrice &libre, matrice &passage);
// Valide le passage pour cette cellule et vérifie pour pour celles adjacentes
void construire_passage(int line, int column,
                        const matrice &libre, matrice &passage,
                        bool &isLastLineReached);

// résolution de 'nbt' grilles avec p défini
int resoudre_nbt_grilles(int n, double p, int nbt,
                         bernoulli_distribution &b, default_random_engine &e);

// résolution dichotomique de p'
void resoudre_dichotomique(double min, double max, double minPrime, double maxPrime,
                           vector<Point> &points, int n, int nbt,
                           default_random_engine &e);

```

```cpp
// Comparaison de deux Points (p,p') pour trier la liste
bool comparer_points(Point p1, Point p2);

// Mode c normal : pas constant de 1/(NBP-1)
void echantillonnage_constant();
// Mode c dichotomique : pas variable
void echantillonnage_adaptif();

/***************** IMPLEMENTATIONS ****************/
// Méthode principale, entrée du programme
int main() {
    srand(time(0));
    string mode;
    cin >> mode;

    if(mode ==  "a") {
        matrice libre, passage;
        int n = read_grid(libre); // Lecture de la grille
        if(n <= 0) return 2;
        passage = matrice(n, ligne(n));

        resoudre_passage(libre, passage); // Résolution de la grille
        output_grid_pbm(passage); // Affichage du résultat dans la console

    } else if (mode == "b") {
        int n, nbt;
        double p;
        cin >> n; // Demande de n, p, nbt
        cin >> p;
        cin >> nbt;
        bernoulli_distribution b(p);
        default_random_engine e;
        if(n <= 0 || nbt <= 0) return 3;

        // Affichage de p'
        cout << setprecision(6); // 6 chiffres à droite de la virgule
        cout << fixed;
        cout << (double)resoudre_nbt_grilles(n, p, nbt, b, e)/nbt << endl;

    } else if (mode == "c") {
        echantillonnage_adaptif(); // Résolution par méthode dichotomique (3.3.2)
        //echantillonnage_constant(); // Résolution par méthode normale (3.3.1)

    } else { // Mode inconnu
        cout << "ERROR : Unknown mode : " << mode << endl;
        return 1;
    }
    return 0;
}

// Méthode permettant de comparer deux Points (p,p')
bool comparer_points(Point p1, Point p2) {
    return (p1.p < p2.p);
}
```

```cpp
// Mode c normal : pas constant (3.3.1)
void echantillonnage_constant() {
    constexpr int nbp(102); // Nombre de points
     // pas constant (NBP de 3 doit donner un pas de 1/2, donc NBP-1 et non NBP-2)
    constexpr double step(1. / (nbp - 1));
    int n, nbt;
    cin >> n; // Demande de n, nbt
    cin >> nbt;
    default_random_engine e;
    double p(0);

    cout << setprecision(6); // 6 chiffres à droite de la virgule
    cout << fixed;
    for (int i = 0; i < nbp; i++)
    {
        bernoulli_distribution b(p);
        // affichage de "<p> <p'>"
        cout << p << " " << (double)resoudre_nbt_grilles(n, p, nbt, b, e)/nbt << endl;
        p += step;
    }
}

// Mode c dichotomique : pas variable (3.3.2)
void echantillonnage_adaptif() {
    int n, nbt;
    cin >> n; // Demande de n, nbt
    cin >> nbt;
    default_random_engine e;
    if(n <= 0 || nbt <= 0) return;
    // Liste des points initialisé avec (0,0) et (1,1)
    vector<Point> points({Point(0,0),Point(1,1)});

    // Résolution dichotomique et récursive
    resoudre_dichotomique(0, 1, 0, 1, points, n, nbt, e);

    // tri croissant de liste avec la méthode de tri comparer_points
    sort(points.begin(), points.end(), comparer_points);

    cout << setprecision(6); // 6 chiffres à droite de la virgule
    cout << fixed;
    for (size_t i = 0; i < points.size(); i++)
    {
        // Affichage de "<p> <p'>"
        cout << points[i].p << " " << points[i].pPrime << endl;
    }
}
```

```
163 // Résolution de 'nbt' grilles, retourne le nombre de grilles qui ont une 'traversée'
164 int resoudre_nbt_grilles(int n, double p, int nbt,
165                          bernoulli_distribution &b, default_random_engine &e) {
166     matrice libre, passage;
167     int compteur_traversee(0);
168 
169     for (int i = 0; i < nbt; i++)
170     {
171         generate_grid(libre, n, n, b, e);
172         passage = matrice(n, ligne(n));
173         // Si une cellule sur la dernière ligne est accessible, augmenter le compteur
174         if(resoudre_passage(libre, passage))
175             compteur_traversee++;
176     }
177     return compteur_traversee;
178 }
179 
180 // Valide le passage pour cette cellule et vérifie pour celles adjacentes
181 void construire_passage(int line, int column,
182                         const matrice &libre, matrice &passage,
183                         bool &isLastLineReached) {
184     // si cellule déjà visitée
185     if(passage[line][column]) return;
186 
187     passage[line][column] = true;
188 
189     int lines(libre.size());
190     int columns(libre[0].size());
191 
192     // Si dernière ligne
193     if((line+1) == lines) isLastLineReached = true;
194 
195     // Vérifier la cellule :
196     // en haut
197     if(line > 0)
198         if(libre[line-1][column])
199             construire_passage(line-1, column, libre, passage, isLastLineReached);
200 
201     // à gauche
202     if(column > 0)
203         if(libre[line][column-1])
204             construire_passage(line, column-1, libre, passage, isLastLineReached);
205 
206     // en bas
207     if((line+1) < lines)
208         if(libre[line+1][column])
209             construire_passage(line+1, column, libre, passage, isLastLineReached);
210 
211     // à droite
212     if((column+1) < columns)
213         if(libre[line][column+1])
214             construire_passage(line, column+1, libre, passage, isLastLineReached);
215 }
216 
```

```
217 // Résolution dichotomique de p'
218 void resoudre_dichotomique(double min, double max, double minPrime, double maxPrime,
219                            vector<Point> &points, int n, int nbt,
220                            default_random_engine &e) {
221     double p((min+max)/2); // Point milieu
222     bernoulli_distribution b(p);
223 
224     // Calcul de p'
225     double pPrime((double)resoudre_nbt_grilles(n, p, nbt, b, e)/nbt);
226     // Calcul de l'erreur absolue
227     double error(pPrime - (minPrime+maxPrime)/2);
228 
229     points.push_back(Point(p, pPrime)); // Sauvegarde du point (p,p')
230 
231     // Vérification des conditions
232     if((max-min) < min_delta_p)
233         return;
234 
235     if(error < -max_error)
236         resoudre_dichotomique(p, max, pPrime, maxPrime, points, n, nbt, e);
237     else if(error > max_error)
238         resoudre_dichotomique(min, p, minPrime, pPrime, points, n, nbt, e);
239 }
240 
241 // Lecture de la grille et retourne n ou -1 si n <= 0
242 int read_grid(matrice &grid) {
243     int n;
244     cin >> n;
245 
246     if(n <= 0) return -1;
247 
248     grid = matrice(n, vector<bool>(n));
249 
250     // Lire chaque cellule, et stocker l'inverse logique
251     bool valeur;
252     for (int i = 0; i < n; i++)
253     {
254         for (int j = 0; j < n; j++)
255         {
256             cin >> valeur;
257             grid[i][j] = !valeur;
258         }
259     }
260     return n;
261 }
262 
263 
264 
265 
266 
267 
268 
269 
270 
```

```cpp
// Résolution du passage en partant de la première ligne
bool resoudre_passage(const matrice &libre, matrice &passage) {
    // Indique si la dernière ligne est accessible
    bool isTraversee = false;
    int columns = libre[0].size();

    // Recherche sur la première ligne
    for (int i = 0; i < columns; i++)
    {
        if(libre[0][i])
            construire_passage(0, i, libre, passage, isTraversee);
    }
    return isTraversee;
}

// Génération d'une grille lines x columns (lines et columns >= 0)
void generate_grid(matrice &grid, int lines, int columns,
                   bernoulli_distribution &b, default_random_engine &e) {
    grid = matrice(lines, ligne(columns, false));

    for (int i = 0; i < lines; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            grid[i][j] = b(e); // Stockage d'un booléen aléatoire
        }
    }
}

// Affiche dans l'output la grille au format pbm
void output_grid_pbm(const matrice &grid) {
    int lines = grid.size();
    int columns = grid[0].size();

    cout << "P1" << endl; // 1e ligne : P1
    cout << lines << " " << columns << endl; // 2e ligne : <colonnes> <lignes>

    for (int i = 0; i < lines; i++)
    {
        for (int j = 0; j < columns; j++)
        {
            cout << !grid[i][j] << " ";
            if((j+1) % 35 == 0) // Si trop de caractères, retour à la ligne
                cout << endl;
        }
        cout << endl;
    }
}
```