Sommaire

Gale Shapley 1

Input 2

Tableaux

- Polynomial 2
- Strings 3
- KMPPlus 4
- ArrayOperations 4
- Knapsack 5
- SubsetSum + kFenetre 6
- Dichotomie 6
- Sort 7
- PermArrays 7
- MatrixOperations 10

Arithmétique

- Binomial 8
- GCD + crible 8
- Base 9

Graphes

- Edge 11
- Graph2 12
- Graph 17
- Couplage Max Biparti 21
- Finding Nemo 24

Géométrie

- ClosestPoint + Convexe 22
- Rectangle Sous Histo 23
- Surface Rectangle 23


```java
public class GaleShapley {
	// Input : (int) n, number of men & women, int[][] preferences of men and
	// women
	// Finds the best matching
	// Output : int[] brides' grooms
	public int[] constructStableMatching(int n, int[][] menPrefs, int[][]
womenPrefs) {
		int[] numberAsked = new int[n], brides = new int[n], grooms =
new int[n];
		int g, f, g2;
		LinkedList<Integer> listG = new LinkedList<Integer>();
		for (int i = 0; i < n; i++) {
			brides[i] = -1; grooms[i] = -1; listG.add(i);
		}
		while (!listG.isEmpty()) {
			g = listG.poll();
			f = menPrefs[g][numberAsked[g]++];
			g2 = grooms[f];
			if (g2 == -1) {
				brides[g] = f; grooms[f] = g;
			} else {
				if (prefers(g, g2, womenPrefs[f], n)) {
					brides[g2] = -1; grooms[f] = g;
					listG.add(g2); brides[g] = f;
				} else listG.add(g);
			}
		}
		return brides;
	}
	public boolean prefers(int g, int g2, int[] fPrefs, int n) {
		for (int i = 0; i < n; i++) {
			if (fPrefs[i] == g) return true;
			if (fPrefs[i] == g2) return false;
		}
		return false; }

}
```

```java
public class Input {
	public static void main(String[] args) throws IOException {
		BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
		int N = Integer.parseInt(br.readLine());
		System.out.println(N);
		br.close();
	}
}
public final class Polynomial {
	public static final int LOG_MAX_LENGTH = 27;
	public static final int MODULUS = 2013265921;
	private static final int PRIMITIVE_ROOT = 137;
	private static final int PRIMITIVE_ROOT_INVERSE = 749463956;
```

```java
	public static void main(String[] args) {
		int[] a = new int[] { 1, 1, 1 };
		int[] b = new int[] { 1, 1, 0, 1 };
		int[] result = multiply(a, b);

		for (int i = 0; i < result.length; i++)
			System.out.println(result[i]);
	}
```

```java
	static int addMultiply(int x, int y, int z) {
		return (int) ((x + y * (long) z) % MODULUS);
	}
```

```java
	private static int[] transform(int[] a, int logN, int primitiveRoot) {
		int[] tA = new int[1 << logN];
		for (int j = 0; j < a.length; j++) {
			int k = j << (32 - logN);
			k = ((k >>> 1) & 0x55555555) | ((k & 0x55555555) << 1);
			k = ((k >>> 2) & 0x33333333) | ((k & 0x33333333) << 2);
			k = ((k >>> 4) & 0x0f0f0f0f) | ((k & 0x0f0f0f0f) << 4);
			k = ((k >>> 8) & 0x00ff00ff) | ((k & 0x00ff00ff) << 8);
			tA[(k >>> 16) | (k << 16)] = a[j];
		}
		int[] root = new int[LOG_MAX_LENGTH];
		root[root.length - 1] = primitiveRoot;
		for (int i = root.length - 1; i > 0; i--) {
			root[i - 1] = addMultiply(0, root[i], root[i]);
		}
		for (int i = 0; i < logN; i++) {
			int twiddle = 1;
			for (int j = 0; j < (1 << i); j++) {
				for (int k = j; k < tA.length; k += 2 << i) {
					int x = tA[k];
					int y = tA[k + (1 << i)];
					tA[k] = addMultiply(x, twiddle, y);
					tA[k + (1 << i)] = addMultiply(x,
MODULUS - twiddle, y);
				}
				twiddle = addMultiply(0, root[i], twiddle);
			}
		}
		return tA;
	}
```

```java
	public static int[] multiply(int[] a, int[] b) {
		int minN = a.length - 1 + b.length;
		int logN = 0;
		while ((1 << logN) < minN) { logN++; }
		int[] tA = transform(a, logN, PRIMITIVE_ROOT);
		int[] tB = transform(b, logN, PRIMITIVE_ROOT);
		int[] tC = tA;
		for (int j = 0; j < tC.length; j++) {
			tC[j] = addMultiply(0, tA[j], tB[j]);
		}
		int[] nC = transform(tC, logN, PRIMITIVE_ROOT_INVERSE);
		int[] c = new int[minN];
		int nInverse = MODULUS - ((MODULUS - 1) >>> logN);
		for (int j = 0; j < c.length; j++) {
			c[j] = addMultiply(0, nInverse, nC[j]);
		}
		return c;
	}
}
```

```java
public class Strings {
	// renvoie le plus long palindrome
	public static String LongestPalindrome(char[] s) {
		if (s == null || s.length == 0)
			return "";

		char[] s2 = addBoundaries(s); // .toCharArray());
		int[] pal = new int[s2.length]; // Plus long palindrome
		int centre = 0, limiteDroite = 0;
		pal[0] = 0;// Le premier element est de taille 0
		// Les 2 indices pour accroite pal[i] en comparant de chaque cote
		int aGauche = 0, aDroite = 0;
		for (int i = 1; i < s2.length; i++) {
			if (i > limiteDroite) {
				pal[i] = 0;
				aGauche = i - 1;
				aDroite = i + 1;
			} else {
				int iOppose = centre * 2 - i;
				if (pal[iOppose] < (limiteDroite - i)) {
					pal[i] = pal[iOppose];
					aGauche = -1; // This signals bypassing
the while loop
												// below.
				} else {
					pal[i] = limiteDroite - i;
					aDroite = limiteDroite + 1;
					aGauche = i * 2 - aDroite;
				}
			}
			// extension du palindrome
			while (aGauche >= 0 && aDroite < s2.length &&
s2[aGauche] == s2[aDroite]) {
				pal[i]++;
				aGauche--;
				aDroite++;
			}
			if ((i + pal[i]) > limiteDroite) {
				centre = i;
				limiteDroite = i + pal[i];
			}
		}

		// On retrouve le maximum des palindrome
		int len = 0;
		centre = 0;
		for (int i = 1; i < s2.length; i++) {
			if (len < pal[i]) {
				len = pal[i];
				centre = i;
			}
		}
		// retour du palindrome (on peut ne que renvoyer sa longueur)
		char[] ss = Arrays.copyOfRange(s2, centre - len, centre + len + 1)
		return String.valueOf(removeBoundaries(ss));
	}
```

```java
	// sert dans le palindrome
	private static char[] addBoundaries(char[] cs) {
		if (cs == null || cs.length == 0)
			return "||".toCharArray();

		char[] cs2 = new char[cs.length * 2 + 1];
		for (int i = 0; i < (cs2.length - 1); i = i + 2) {
			cs2[i] = '|';
			cs2[i + 1] = cs[i / 2];
		}
		cs2[cs2.length - 1] = '|';
		return cs2;
	}
	private static char[] removeBoundaries(char[] cs) {
		if (cs == null || cs.length < 3)
			return "".toCharArray();

		char[] cs2 = new char[(cs.length - 1) / 2];
		for (int i = 0; i < cs2.length; i++) {
			cs2[i] = cs[i * 2 + 1];
		}
		return cs2;	}}
```

```java
public class KMPplus {
	private String pattern;
	private int[] next;
	// creation du tableau de prefixes
	public KMPplus(String pattern) {
		this.pattern = pattern;
		int M = pattern.length();
		next = new int[M];
		int j = -1;
		// j=longueur du + long suffixe&prefixe; i=tete lecture
		for (int i = 0; i < M; i++) {
			if (i == 0)
				next[i] = -1;
			else if (pattern.charAt(i) != pattern.charAt(j))
				next[i] = j;
			else
				next[i] = next[j];
			while (j >= 0 && pattern.charAt(i) != pattern.charAt(j)) {
				j = next[j];
			}
			j++;
		}}
```

```java
// indice ou commence la premi�re occurence, -1 sinon.
	public int search(String text) {
		int M = pattern.length();
		int N = text.length();
		int i, j;
		for (i = 0, j = 0; i < N && j < M; i++) {
			while (j >= 0 && text.charAt(i) != pattern.charAt(j))
				j = next[j];
			j++;
		}
		if (j == M)
			return i - M;
		else return -1 ; }
}
```

```java
public class ArrayOperations {
	// reverse array
	public static int[] reverse(int[] t) {
		int n = t.length;
		int[] a = Arrays.copyOf(t, n);
		int c;
		for (int i = 0; i < (n + 1) / 2; i++) {
			c = a[i];
			a[i] = a[n - 1 - i];
			a[n - 1 - i] = c;
		}
		return a;
	}
```

```java
	// longest common subsequence
	public static int longestSubsequence(int[] a, int[] b) {
		int[][] longest = new int[a.length][b.length];
		int result = longestSubsequence(a, b, a.length - 1, b.length - 1,
longest);
		PermArrays.print(longest);
		return result;
	}
```

```java
	// recursive function for longest subsequence
	public static int longestSubsequence(int[] a, int[] b, int i, int j, int[][]
longest) {
		if (i == -1 || j == -1)  return 0;

		if (a[i] == b[j])
			longest[i][j] = 1 + longestSubsequence(a, b, i - 1, j - 1,
longest);

		else
			longest[i][j] = Math.max(longestSubsequence(a, b, i, j - 1,
longest),
					longestSubsequence(a, b, i - 1, j, longest))
		return longest[i][j];

	}
```

```java
	// plus longue sous-sequence croissante strictement
	static int find_lis(int[] x) {
		int[] b = new int[x.length + 1];
		Arrays.fill(b, Integer.MAX_VALUE);
		int best = 0, bot, top, m;
		for (int xi : x) {
			for (bot = 0, top = best; bot < top;) {
				m = (bot + top) / 2;
				if (b[m] < xi)
					bot = m + 1;
				else
					top = m;
			}
			b[bot] = xi;
			if (bot == best)
				best = bot + 1;
		}
		return best;
	}
```

```java
	static int biggestSum(int[] set) {
		int max = 0, curSum = 0;
		for (int i = 0; i < set.length; i++) {
			curSum += set[i];
			if (curSum > max)
				max = curSum;
			if (curSum <= 0)
				curSum = 0;
		}
		return max;
	}

}
```

```java
public class Knapsack {
	int n;
	int p;
	int[] weight;
	int[] price;
	int[][] best;

	Knapsack(int[] poids, int[] prix, int pp) { // pp = poidsMax
		n = poids.length;
		p = pp;
		weight = poids;
		price = prix;
		best = new int[n + 1][p + 1];
		for (int j = 0; j < p + 1; j++)
			best[0][j] = (j >= weight[0]) ? price[0] : 0;
		for (int i = 1; i < n; i++)
			for (int j = 0; j < p + 1; j++)
				best[i][j] = (j >= weight[i]) ? Math.max(best[i - 1][j], best[i - 1][j - weight[i]] + price[i]) : best[i - 1][j];
	}

	void print() {
		for (int j = 0; j < p + 1; j++)
			System.out.print(best[n - 1][j] + " ");
	}
}
```

```java
public class SubsetSum {

	// boolean savoir si on peut écrire sum à partir d'elts de set
	static boolean isSubsetSum(int set[], int sum) {
		int n = set.length;
		boolean[][] subset = new boolean[sum + 1][n + 1];
		for (int i = 0; i <= n; i++)
			subset[0][i] = true;

		for (int i = 1; i <= sum; i++)
			subset[i][0] = false;

		for (int i = 1; i <= sum; i++) {
			for (int j = 1; j <= n; j++) {
				subset[i][j] = subset[i][j - 1];
				if (i >= set[j - 1])
			subset[i][j] = subset[i][j] || subset[i - set[j - 1]][j - 1];
			}
		}

		return subset[sum][n];
	}
```

```java
	public static int kFenetre(int[] t, int k) {
		TreeSet<Integer> a = new TreeSet<Integer>();
		if (k > t.length)
			return -1;
		for (int i = 0; i < t.length; i++) {
			if (a.contains(t[i]))
				a.clear();

			a.add(t[i]);

			if (a.size() == k)
				return (i - k + 1);
		}
		return -1;
	}
}
```

```java
public class Dichotomie {
	// renvoie l'indice correspondant � l'entier e
	// l'element renvoye est strictement inferieur � e, et element + 1 superieur
	// ou egal (peut retourner -1)
	static int findIndiceInferieur(int[] t, int e) {
		return auxFind(t, 0, t.length - 1, e);
	}
	// l'element renvoye est strictement inferieur � e et element + 1 est
	// superieur ou egal
	private static int auxFind(int[] t, int bas, int haut, int e) {
		int taille = haut - bas;
		if (taille == 0) { // cas particulier
			if (t[bas] >= e)  return bas - 1;
			else return bas;
		} else if (taille == 1) // cas particulier
		{
			if (t[bas] >= e)
				return bas - 1;
			else if (t[haut] < e)  return haut;
			else return bas;
		} else {
			int milieu = (haut + bas) / 2;
			if (e <= t[milieu])  return auxFind(t, bas, milieu, e);
			else return auxFind(t, milieu, haut, e);
		}

	}
	// trouve e dans t, et renvoie l'indice si trouv�
	// renvoye -1 si non trouve
	static int find(int[] t, int e) {
		int indice = findIndiceInferieur(t, e) + 1;
		if (indice < t.length)
			if (t[indice] == e)
				return indice;
		return -1;
	}

}
```

```java
public class Sort {

	public static int[] merge(int[] a, int[] b) {
		int i = 0, j = 0, k = 0;
		int[] c = new int[a.length + b.length];
		while (k < c.length) {
			if (i == a.length)
				c[k++] = b[j++];
			else if (j == b.length || a[i] < b[j])
				c[k++] = a[i++];
			else
				c[k++] = b[j++];
		}
		return c;
	}
```

```java
// MERGE SORT a
	public static int[] mergeSort(int[] a) {
		if (a.length == 1)
			return a;
		int mid = a.length / 2;
		int[] left = new int[mid];
		int[] right = new int[a.length - mid];
		for (int i = 0; i < mid; i++) {
			left[i] = a[i];
			right[i] = a[i + mid];
		}
		right[right.length - 1] = a[a.length - 1];
		return merge(mergeSort(left), mergeSort(right));
	}

}
```

```java
public class PermArrays {
	// Input : (int) n
	// Outputs : int[][] all the permutations of {1..n}
	public static int[][] generatePermutations(int n) {
		if (n == 1) {
			int[][] result = new int[1][];
			result[0] = new int[] { 1 };
			return result;
		}
		int[][] previous = generatePermutations(n - 1);
		int[][] result = new int[n * previous.length][n];
		for (int i = 0; i < previous.length; i++) {
			for (int k = 0; k < n; k++) {
				for (int j = 0; j < previous[i].length; j++)
					result[i * n + k][j] = previous[i][j];
				result[i * n + k][n - 1] = n;

				swap(result[i * n + k], k, n - 1);
			}
		}
		return result;
	}
```

```java
	public static void swap(int[] t, int i, int j) {
		int tmp = t[i];
		t[i] = t[j];
		t[j] = tmp;
	}
}
```

```java
public class Binomial {

	// Input : (int) n
	// Output : (int) n!
	static int fact(int n) {
		if (n == 0) return 1;
		else return n * fact(n - 1);
	}

	// Input : k, n (int)
	// Output : (int) k parmi n
	static int binomial(int k, int n) {
		int result = 1;
		if (n - k < k)
			k = n - k;
		for (int i = k + 1; i <= n; i++) {
			result *= i;
			result /= (i - k);
		}
		return result;
	}

	// Input : (int) n
	// Output : triangle de pascal jusqu'à n
	static int[][] pascal(int n) {
		int[][] pascal = new int[++n][];
		pascal[0] = new int[] { 1 };

		for (int i = 1; i < n; i++) {
			pascal[i] = new int[i + 1];
			for (int j = 0; j < pascal[i].length; j++)
				pascal[i][j] = ((i > j) ? pascal[i - 1][j] : 0) + ((j > 0)
? pascal[i - 1][j - 1] : 0);
		}
		return pascal;
	}
}
```

```java
public class Gcd {
	// Modulo
	public static int reduce(int n, int mod) {
		int m = n % mod; // -mod < m < mod
		if (m >= 0)  return m;
		else return m + mod;
	}

// Pgcd
	public static int gcd(int m, int n) {
		int r;
		// Exchange m and n if m < n
		if (m < n) { r = n;  n = m;  m = r; }
		// It can be assumed that m >= n
		while (n > 0) { r = m % n;  m = n;  n = r; }
		return m;
	}

	// Computes the GCD and the coefficients of the Bezout equality.
	public static int[] extgcd(int m, int n) {
		// Both arrays ma and na are arrays of 3 integers such that
		// ma[0] = m ma[1] + n ma[2] and na[0] = m na[1] + n na[2]
		int[] ma = new int[] { m, 1, 0 };
		int[] na = new int[] { n, 0, 1 };
		int[] ta; // Temporary variable
		int i; // Loop index
		int q; // Quotient
		int r; // Rest

		// Exchange ma and na if m < n
		if (m < n) {
			ta = na;
			na = ma;
			ma = ta;
		}

		// It can be assumed that m >= n
		while (na[0] > 0) {
			q = ma[0] / na[0]; // Quotient
			for (i = 0; i < 3; i++) {
```

```java
                r = ma[i] - q * na[i];
                ma[i] = na[i];
                na[i] = r;
            }
        }
        return ma;
    }
```

```java
    // Computes the modular inverse
    public static int modInverse(int n, int mod) {
        int[] g = extgcd(mod, n);
        if (g[0] != 1)
            return -1; // n and mod not coprime
        else
            return reduce(g[2], mod);
    }
```

```java
    // crible d'eratosthene
    public static boolean[] crible(int limite) {

        boolean[] c = new boolean[limite];
        Arrays.fill(c, true);

        c[0] = false;
        c[1] = false;

        for (int i = 4; i < limite; i += 2)
            c[i] = false;

        int suivant = 3;
        while (suivant < Math.sqrt(limite)) {
            for (int i = 2 * suivant; i < limite; i += suivant)
                c[i] = false;

            for (suivant++; c[suivant] == false; suivant++);
        }
        return c;
    }
}
```

```java
public class Base {
    int n;
    int b;
    int length;
    int[] dec;

    Base(int nn, int bb) {
        n = nn;
        b = bb;
        length = (int) (Math.floor(Math.log(n) / Math.log(b)) + 1);
        dec = new int[length];
        for (int i = length - 1; i >= 0; i--) {
            dec[i] = n % b;
            n /= b;
        }
    }

    int toInt() {
        int n = 0;
        for (int i = 0; i < dec.length; i++)
            n = n * b + dec[i];
        return n;
    }

}
```

```java
public class MatrixOperations {
	// Input integer n
	// Output : identity matrix In
	public static double[][] identity(int n) {
		double[][] result = new double[n][n];
		for (int i = 0; i < n; i++)
			result[i][i] = 1.0;
		return result;
	}
```

```java
	// Input : 2 matrixes A, B (double[][])
	// Output : A*B (double[][])
	public static double[][] multiplyMatrix(double[][] A, double[][] B) {
		int N = A.length;
		double[][] C = new double[N][N];
		for (int i = 0; i < N; i++) {
			for (int k = 0; k < N; k++) {
				for (int j = 0; j < N; j++) {
					C[i][j] += A[i][k] * B[k][j];
				}
			}
		}
		return C;
	}
```

```java
	// Input : matrix A (double[][]), n (int)
	// Output : A^n (double[][])
	public double[][] matrixPower(double[][] A, int n) {
		if (n == 1)
			return A;
		double[][] temp = matrixPower(A, n / 2);
		if (n % 2 == 0)
			return multiplyMatrix(temp, temp);
		else
			return multiplyMatrix(temp, multiplyMatrix(temp, A));

	}
```

```java
public static double[][] transpose(double[][] A) {
		int n = A.length;
		double[][] result = new double[n][n];
		for (int i = 0; i < n; i++)
			for (int j = 0; j < n; j++)
				result[i][j] = A[j][i];
		return result;
	}
```

```java
	// Input : matrix A (double[][]), (int) i j
	// Swaps lines i and j of A
	public static void swap(double[][] A, int i, int j) {
		double[] temp = A[i];
		A[i] = A[j];
		A[j] = temp;
	}
```

```java
	// Input : 2 matrixes a, b
	// Output : X = A^-1 *B. Leaves a, b unchanged
	public static double[][] solve(double[][] a, double[][] b) {
		int n = a.length;
		int rowsB = b.length;
		int colB = b[0].length;

		if (rowsB != n && n != a[0].length)
			throw new RuntimeException("Illegal matrix
dimensions.");

		double[][] A = new double[n][n], B = new double[rowsB][colB]
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < n; j++)
				A[i][j] = a[i][j];
			for (int j = 0; j < colB; j++)
				B[i][j] = b[i][j];
		}

		for (int i = 0; i < n; i++) {
			int max = i;
			for (int j = i + 1; j < n; j++)
				if (Math.abs(A[j][i]) > Math.abs(A[max][i]))
```

```java
                    max = j;

            swap(A, i, max);
            swap(B, i, max);

            if (A[i][i] == 0.0)
                throw new RuntimeException("Matrix is singular.");

            for (int j = i + 1; j < n; j++)
                for (int k = 0; k < colB; k++)
                    B[j][k] -= B[i][k] * A[j][i] / A[i][i];

            for (int j = i + 1; j < n; j++) {
                double m = A[j][i] / A[i][i];
                for (int k = i + 1; k < n; k++)
                    A[j][k] -= A[i][k] * m;
                A[j][i] = 0.0;
            }
        }

        double t;
        double[][] x = new double[n][colB];
        for (int j = 0; j < colB; j++) {
            for (int i = n - 1; i >= 0; i--) {
                t = 0.0;
                for (int k = i + 1; k < n; k++)
                    t += A[i][k] * x[k][j];
                x[i][j] = (B[i][j] - t) / A[i][i];
            }
        }
        return x;
    }

}
```

```java
public class Edge {
    int from;
    int to;
    int distance;
    int capacity;
    int flow;
    Edge oppose;

    public Edge(Edge e) ;

    public Edge(int from, int to, int distance, int capacity, int flow) ;

    public Edge(int from, int to, int distance, int capacity, int flow, Edge oppose) ;

    public Edge(int from, int to, int distance) ; // capacity, flow = 0

    public Edge(int from, int to, int distance, Edge oppose) ; // same

    public Edge(int from, int to) ; // same & distance = 1
}
```

```java
public class Graph2 {
	LinkedList<Edge>[] noeuds;
	int n; // nbr de noeuds
	int longueur; // Pour Prim
	// Pour dinic
	int[] l; // niveau
	Graph2 gr; // graph residu
	// Pour Mincut
	LinkedList<Edge> aCouper;
	// Dijkstra
	int[] distances;

	Graph2(int n) {
		this.n = n;
		noeuds = new LinkedList[n];
		for (int i = 0; i < n; i++)
			noeuds[i] = new LinkedList<>();
		l = new int[n]; // Pour dinic
	}

	boolean existe(int i, int j) // renvoie true si l'arrete existe
	{
		for (Edge k : noeuds[i])
			if (k.to == j)
				return true;
		return false;
	}

	void addEdge(Edge e) { // ajoute une arrete
		noeuds[e.from].add(e);
	}

	void addDoubleEdge(Edge ee) { // ajoute une arrete et son opposee
		Edge e = new Edge(ee);
		Edge eOppose = new Edge(e.to, e.from, e.distance);
		eOppose.oppose = e;
		e.oppose = eOppose;
		noeuds[e.from].add(e);
		noeuds[e.to].add(eOppose);
	}
```

```java
int shortestPath01(int entree, int sortie) { // Labyrinthe avec porte

			boolean flag[] = new boolean[n]; // -- dÃ©jÃ  visitÃ© ?
			Deque<Integer> Q = new LinkedList<Integer>(); // -- sommets Ã 
visiter
			Q.offer(entree); // -- commencer chez nemo

			int d[] = new int[n]; // c -- distance
			Arrays.fill(d, Integer.MAX_VALUE);
			d[entree] = 0;
			while (!Q.isEmpty()) {
				int p = Q.pollFirst();
				if (p == sortie)
					return d[p];
				if (flag[p]) // -- ne pas traiter deux fois
					continue;
				flag[p] = true;
				// -- parcourir les 4 voisins

				for (Edge e : noeuds[p]) {// i=0; i<4; i++) {
					if (flag[e.to])
						continue; // un point deja visite ou un
mur? ignorer
					if (e.distance == 0) {
						d[e.to] = Math.min(d[e.to], d[p]);
						Q.offerFirst(e.to);
					} else { // PORTE
						d[e.to] = Math.min(d[e.to], d[p] + 1);
						Q.offerLast(e.to);
					}
				}
			}
			return -1; // nemo ne peut pas quitter le labyrinthe
		}
```

```java
// Renvoie toute les distances de tout les noeuds
		public int[][] floydWarshall() {
			int[][] distTo = new int[n][n];
			Edge[][] edgeTo = new Edge[n][n];
```

```java
		// initialize distances to infinity
		for (int v = 0; v < n; v++) {
			for (int w = 0; w < n; w++) {
				distTo[v][w] = Integer.MAX_VALUE / 2;
			}
		}

		// initialize distances using edge-weighted digraph's
		for (int i = 0; i < n; i++) {
			for (Edge e : noeuds[i]) {
				distTo[e.from][e.to] = e.distance;
				edgeTo[e.from][e.to] = e;
			}
			// in case of self-loops
			if (distTo[i][i] >= 0) {
				distTo[i][i] = 0;
				edgeTo[i][i] = null;
			}
		}

		// Floyd-Warshall updates
		for (int i = 0; i < n; i++) {
		// compute shortest paths using only 0, 1, ..., i as intermediate
			// vertices
			for (int v = 0; v < n; v++) {
				if (edgeTo[v][i] == null)
					continue; // optimization
				for (int w = 0; w < n; w++) {
					if (distTo[v][w] > distTo[v][i] +
distTo[i][w]) {
						distTo[v][w] = distTo[v][i] +
distTo[i][w];
						edgeTo[v][w] = edgeTo[i][w];
					}
				}
				// check for negative cycle
				if (distTo[v][v] < 0) {
					System.out.println("cycle negatif");
					return null;
```

```java
					}
				}
			}

			return distTo;
		}
```

```java
		// Peut renvoyer le chemin (en decommentant la fin)
		int shortestPath(int begin, int end) { // Dijkstra : toutes les distances

		// dans distances
			if (n == 0)
				return Integer.MAX_VALUE;

			List<Edge> shortest = new LinkedList<Edge>(); // shortest
			boolean[] visited = new boolean[n];
			distances = new int[n];
			for (int i = 0; i < distances.length; i++)
				distances[i] = Integer.MAX_VALUE;
			distances[begin] = 0;
			PriorityQueue<Edge> pq = new PriorityQueue<>(n, new
DijkstraComparator());

			int node = begin;
			pq.add(new Edge(-1, begin, 0, 0, 0));

			while (!pq.isEmpty()) {
				node = pq.poll().to;
				if (visited[node])
					continue;
				visited[node] = true;
				List<Edge> edges = noeuds[node];
				for (Edge e : edges) {
					if (!visited[e.to]) {
						distances[e.to] =
Math.min(distances[e.from] + e.distance, distances[e.to]);
						pq.add(e);
						shortest.add(e); // shortest
					}
				}
```

```java
		}

		// ******** shortest
		Edge current = new Edge(0, 0, 0, 0, 0);
		for (Edge e : shortest) {
			if (e.to == end)
				current = e;
		}
		System.out.println(current.from + "=>" + current.to + "(" +
current.distance + ")");

		while (current.from != begin) {
			for (Edge e : shortest) {
				if (e.to == current.from)
					current = e;
			}
			System.out.println(current.from + "=>" + current.to + "("
+ current.distance + ")");
		}
		// ***********
		return distances[end];
	}
```

```java
	// Si cycle n�gatifs
	int BellmanFord(int begin, int end) {
		if (n == 0)
			return Integer.MAX_VALUE;
		distances = new int[n];
		for (int i = 0; i < distances.length; i++)
			distances[i] = Integer.MAX_VALUE / 2;
		distances[begin] = 0;

		for (int i = 0; i < n; i++)
			for (int j = 0; j < n; j++)
				for (Edge e : noeuds[j])
					distances[e.to] =
Math.min(distances[e.from] + e.distance, distances[e.to]);
		return distances[end];
	}
```

```java
	// Arbre couvrant Kruskal (en O(MlnM)) utilise racine
	Graph2 arbreCouvrantKruskal() {
		PriorityQueue<Edge> q = new PriorityQueue<>(n, new
EdgeComparator());
		Graph2 arbre = new Graph2(n);
		int somme = 0;

		for (int i = 0; i < noeuds.length; i++)
			for (Edge a : noeuds[i])
				q.add(a);

		int[] succ = new int[n];
		Arrays.fill(succ, -1);

		while (!q.isEmpty()) {
			Edge e = q.poll();
			int racine1 = racine(e.to, succ);
			int l1 = longueur;
			int racine2 = racine(e.from, succ);
			int l2 = longueur;

			if (racine1 != racine2) {
				if (l1 > l2)
					succ[racine2] = racine1;
				else
					succ[racine1] = racine2;
				arbre.addDoubleEdge(e);
				somme += e.distance;
			}
		}
		System.out.println(somme);
		return arbre;

	}
```

```java
	// renvoie la racine, utilisee par kruskal
	int racine(int sommet, int[] succ) {
		int i = sommet;
		longueur = 0;
```

```java
		while (succ[i] != -1) {
			i = succ[i];
			longueur++;
		}
		return i;
	}
```

```java
	// Arbre couvrant Prim (O(N^2))
	Graph2 arbreCouvrantPrim(int source) {
		Graph2 arbre = new Graph2(n);
		int somme = 0;

		int[] distances = new int[n];
		int[] prox = new int[n];

		Arrays.fill(prox, -1);
		Arrays.fill(distances, Integer.MAX_VALUE);

		for (Edge e : noeuds[source]) {
			distances[e.to] = e.distance;
			prox[e.to] = source;
		}
		prox[source] = -2;

		for (int i = 0; i < n - 1; i++) {
			float min = Integer.MAX_VALUE;
			int aAjouter = -1;
			for (int j = 0; j < n; j++)
				if (distances[j] < min && prox[j] != -2) {
					min = distances[j];
					aAjouter = j;
				}
			if (aAjouter == -1) {
				System.out.println("Graph pas connexe");
				return null;
			}
			arbre.addDoubleEdge(new Edge(aAjouter,
prox[aAjouter], distances[aAjouter]));
			somme += distances[aAjouter];
			prox[aAjouter] = -2;
			for (Edge e : noeuds[aAjouter])
				if (prox[e.to] != -2 && distances[e.to] >
e.distance) {
					prox[e.to] = aAjouter;
					distances[e.to] = e.distance;
				}
		}
		System.out.println(somme);
		return arbre;

	}
```

```java
	// Renvoie le mincut, les arrete � couper sont dans aCouper
	int minCut(int s, int t) {
		int cut = dinic(s, t);
		aCouper = new LinkedList<Edge>();
		Queue<Integer> q = new LinkedList<Integer>();
		q.add(s);
		boolean[] flag = new boolean[n];
		flag[s] = true;
		while (!q.isEmpty()) {
			int sommet = q.poll();
			for (Edge e : gr.noeuds[sommet])
				if (!flag[e.to] && e.capacity != e.flow) {
					flag[e.to] = true;
					q.add(e.to);
				}
		}

		for (int i = 0; i < n; i++)
			for (Edge e : gr.noeuds[i])
				if (flag[e.from] == true && flag[e.to] == false)
					aCouper.add(e);

		return cut;

	}
```

```java
// Dinic avec optimisation de memeoire
int dinic(int s, int t) {
	Queue<Integer> q = new LinkedList<Integer>();
	gr = new Graph2(n);
	for (int i = 0; i < n; i++)
		for (Edge e : noeuds[i])
			gr.addDoubleEdge(e);
	int total = 0;
	while (true) {
		q.offer(s);
		Arrays.fill(l, -1);
		l[s] = 0;
		while (!q.isEmpty()) {
			int u = q.remove();
			for (Edge e : gr.noeuds[u]) {
				if (l[e.to] == -1 && e.capacity > e.flow) {
					l[e.to] = l[u] + 1;
					q.offer(e.to);
				}

			}
		}
		if (l[t] == -1)
			return total;
		total += auxDinic(s, t, Integer.MAX_VALUE);
	}
}
```

```java
int auxDinic(int u, int t, int val) {
	if (val <= 0)
		return 0;
	if (u == t)
		return val;
	int a = 0, av;
	for (Edge e : gr.noeuds[u])
		if (l[e.to] == l[u] + 1 && e.capacity > e.flow) {
			av = auxDinic(e.to, t, Math.min(val - a, e.capacity - e.flow));
			e.flow += av;
			e.oppose.flow -= av;
			a += av;
		}
	if (a == 0)
		l[u] = -1;
	return a;
}
```

```java
// utilisee par Shortestpath
class DijkstraComparator implements Comparator<Edge> {

	@Override
	public int compare(Edge o1, Edge o2) {
		// return distances[o1] - distances[o2];
		return (distances[o1.from] + o1.distance) - (distances[o2.from] + o2.distance);
	}

}
```

```java
// utiliser par Kruskal
class EdgeComparator implements Comparator<Edge> {

	@Override
	public int compare(Edge o1, Edge o2) {
		return (o1.distance - o2.distance);

	}

}
```

```java
public class Graph {
	LinkedList<Integer>[] noeuds;
	int n; // nbr de noeuds
	// Pour le calcul de low, arbre stocke l'arbre couvrant calcul�
	int[] low;
	int temps;
	int ordre;
	int[] du;
	int[] fu;
	int ordreDesSommets[];
	int numeroComposante[];
	boolean[] flag;
	Graph arbre;

	// Pour dinic
	int[] l; // niveau
	int[][] F; // flows
	int[][] C; // capacite
	Graph gr;

	// Pour mincut
	LinkedList<Integer> aCouper; // arretes � couper
	// Pour 2sat
	int[] val;

	// Pour Kruskal
	int longueur;
```

```java
	Graph(int n) // modifier comme il faut
	{
		this.n = n;
		noeuds = new LinkedList[n];
		for (int i = 0; i < n; i++)
			noeuds[i] = new LinkedList<Integer>();
		C = new int[n][n]; // pour flow
		F = new int[n][n]; // Pour flow
		l = new int[n]; // Pour dinic

	}
```

```java
	Graph(int n, int[][] C) // modifier comme il faut
	{
		this.n = n;
		noeuds = new LinkedList[n];
		for (int i = 0; i < n; i++)
			noeuds[i] = new LinkedList<Integer>();
		this.C = C; // pour flow
		F = new int[n][n]; // Pour flow
		l = new int[n]; // Pour dinic

	}
```

```java
	void add(int i, int j) {
		noeuds[i].add(j);
	}
```

```java
	void addNonOriente(int i, int j) {
		add(i, j);
		add(j, i);
	}
```

```java
	// utilisee dans calcul du low
	boolean existe(int i, int j) {
		for (int k : noeuds[i])
			if (k == j) return true;
		return false;
	}
```

```java
	int composantesConnexes() // utilise calculelow et dfsInv {
		calculeLow();
		Arrays.fill(flag, false);
		int composante = 0;
		numeroComposante = new int[n];
		for (int i = 0; i < n; i++) {
			int u = ordreDesSommets[i];
			if (flag[u] == false)
				dfsInv(u, composante++);
		}
		return composante++; }
```

```java
void calculeLow() // utilise dfs
	{
		low = new int[n]; // uniquement pour low
		du = new int[n];
		fu = new int[n];
		ordreDesSommets = new int[n];
		flag = new boolean[n];
		temps = 0;
		ordre = 0;
		Arrays.fill(du, 0);
		Arrays.fill(fu, 0);
		Arrays.fill(flag, false);
		Arrays.fill(ordreDesSommets, 0);
		arbre = new Graph(noeuds.length);
		for (int i = 0; i < n; i++)
			if (flag[i] == false)
				dfs(i);
	}
```

```java
	void dfsInv(int i, int composante) {
		flag[i] = true;
		numeroComposante[i] = composante;
		for (int j : noeuds[i])
			if (flag[j] == false)
				dfsInv(j, composante);
	}
```

```java
	private void dfs(int i) // dfs utilisee par low, composantes connexes
	{

		flag[i] = true;
		du[i] = temps;
		temps++;
		low[i] = du[i]; // uniquement pour low
		for (int j : noeuds[i]) {
			if (flag[j] == false) {
				arbre.add(i, j);
				dfs(j);
				if (low[j] < low[i]) // uniquement pour low
					low[i] = low[j]; // uniquement pour low
			} else // uniquement pour low
			{
				if (du[j] < du[i] & !arbre.existe(j, i)) {
					// Si besoin de stocker les arretes retour :
					// arreteRetour.add(i, j);
					if (low[i] > du[j])
						low[i] = du[j];
				}
			}
		}

		ordreDesSommets[ordre] = i; // uniquement pour composante
connexe
		ordre++; // uniquement pour composante connexe
		fu[i] = temps;
		temps++;
	}
```

```java
	int plusCourtChemin(int source, int cible) // simple dfs
	{
		LinkedList<Integer> l = new LinkedList<Integer>();
		l.add(source);
		boolean[] b = new boolean[noeuds.length];
		int[] d = new int[noeuds.length];
		Arrays.fill(b, false);
		d[source] = 0;
		b[source] = true;
		while (!l.isEmpty()) {
			int s = l.poll();
			for (int j : noeuds[s])
				if (!b[j]) {
					d[j] = d[s] + 1;
					l.addLast(j);
					b[j] = true;
				}
			if (s == cible)
				return d[s];
		}
		return -1;
	}
```

```
int dinic(int s, int t) // renvoie le maxflow, tout est stock� dans gr
                        // utilise auxDinic
	{
		Queue<Integer> q = new LinkedList<Integer>();
		gr = new Graph(n, C);
		for (int i = 0; i < n; i++)
			for (int j : noeuds[i])
				gr.addNonOriente(j, i);
		int total = 0;
		while (true) {
			q.offer(s);
			Arrays.fill(l, -1);
			l[s] = 0;
			while (!q.isEmpty()) {
				int u = q.remove();
				for (int v : gr.noeuds[u]) {
					if (l[v] == -1 && C[u][v] > F[u][v]) {
						l[v] = l[u] + 1;
						q.offer(v);
					}
				}
			}
			if (l[t] == -1)
				return total;
			total += auxDinic(s, t, Integer.MAX_VALUE);
		}
	}
```

```
	int auxDinic(int u, int t, int val) // renvoie le maxflow possible inferieur
                                        //
� val entre u et t
	{
		if (val <= 0)
			return 0;
		if (u == t)
			return val;
		int a = 0, av;
		for (int v : gr.noeuds[u])
			if (l[v] == l[u] + 1 && C[u][v] > F[u][v]) {
				av = auxDinic(v, t, Math.min(val - a, C[u][v] -
F[u][v]));
				F[u][v] += av;
				F[v][u] -= av;
				a += av;
			}
		if (a == 0)
			l[u] = -1;
		return a;
	}
```

```
	// Returns tne maximum flow from s to t in the given graph
	int fordFulkerson(int s, int t) {
		int u, v;
		int rGraph[][] = new int[n][n];

		for (u = 0; u < n; u++)
			for (v = 0; v < n; v++)
				rGraph[u][v] = C[u][v];

		int parent[] = new int[n]; // This array is filled by BFS and to stor
		                                                                  // path

		int max_flow = 0; // There is no flow initially

		// Augment the flow while there is path from source to sink
		while (bfs(rGraph, s, t, parent)) {
			int path_flow = Integer.MAX_VALUE;
			for (v = t; v != s; v = parent[v]) {
				u = parent[v];
				path_flow = Math.min(path_flow, rGraph[u][v]);
			}

			for (v = t; v != s; v = parent[v]) {
				u = parent[v];
				rGraph[u][v] -= path_flow;
				rGraph[v][u] += path_flow;
			}
```

```java
				// Add path flow to overall flow
				max_flow += path_flow;
			}

			// Return the overall flow
			return max_flow;
		}
```

```java
		private boolean bfs(int rGraph[][], int s, int t, int parent[])
	// trouve un chemin  augmentant
		{
			boolean visited[] = new boolean[n];
			for (int i = 0; i < n; ++i)
				visited[i] = false;

			LinkedList<Integer> queue = new LinkedList<Integer>();
			queue.add(s);
			visited[s] = true;
			parent[s] = -1;

			// Standard BFS Loop
			while (queue.size() != 0) {
				int u = queue.poll();
				for (int v = 0; v < n; v++) {
					if (visited[v] == false && rGraph[u][v] > 0) {
						queue.add(v);
						parent[v] = u;
						visited[v] = true;
					}
				}
			}
			return (visited[t] == true);
		}
```

```java
		// Litteraux reprensente de 1 � n, negatif si ils sont negation
		static int[] deuxSat(int[][] formule) {
			int nombreLitteraux = 0;
			for (int i = 0; i < formule.length; i++) {
				if (Math.abs(formule[i][0]) > nombreLitteraux)
					nombreLitteraux = Math.abs(formule[i][0]);
				if (Math.abs(formule[i][1]) > nombreLitteraux)
					nombreLitteraux = Math.abs(formule[i][1]);
			}

			Graph g = new Graph(2 * nombreLitteraux + 1);

			for (int i = 0; i < formule.length; i++) {
				g.add(-formule[i][0] + nombreLitteraux, formule[i][1] +
nombreLitteraux);
				g.add(-formule[i][1] + nombreLitteraux, formule[i][0] +
nombreLitteraux);
			}

			int nombreComposantes = g.composantesConnexes();
			int[] neg = new int[nombreComposantes];
			int[] valbis = new int[nombreComposantes];

			for (int x = 1; x < nombreLitteraux + 1; x++) {
				if (g.numeroComposante[x + nombreLitteraux] ==
g.numeroComposante[-x + nombreLitteraux])
					return null;
				else {
					neg[g.numeroComposante[x + nombreLitteraux]]
= g.numeroComposante[-x + nombreLitteraux];
					neg[g.numeroComposante[-x + nombreLitteraux]]
= g.numeroComposante[x + nombreLitteraux];
				}
			}

			for (int j = 0; j < nombreComposantes; j++)
				if (valbis[j] == 0 && j !=
g.numeroComposante[nombreLitteraux]) {
					valbis[j] = 1;
					valbis[neg[j]] = -1;
				}

			g.val = new int[nombreLitteraux + 1];

			for (int i = 1; i < nombreLitteraux + 1; i++)
```

```
				g.val[i] = valbis[g.numeroComposante[i +
nombreLitteraux]];

			return g.val;
		}
```

```
// Prend en argumant un arbre
		int plusLongChemin() {
			chemin = new int[n];
			maxprofondeur = new int[n];
			flag = new boolean[n];
			int source = 3;
			auxChemin(source);
			return Math.max(maxprofondeur[source], chemin[source]);
		}
```

```
		private void auxChemin(int i) {
			flag[i] = true;
			if (noeuds[i] == null || noeuds[i].isEmpty()) {
				maxprofondeur[i] = 0;
				chemin[i] = 0;
			} else {
				int max1 = -1, max2 = -1;
				chemin[i] = 0;
				for (int j : noeuds[i])
					if (flag[j] == false) {
						auxChemin(j);
						if (maxprofondeur[j] > max1) {
							max2 = max1;
							max1 = maxprofondeur[j];
						} else if (maxprofondeur[j] > max2)
							max2 = maxprofondeur[j];

						if (chemin[j] > chemin[i])
							chemin[i] = chemin[j];
					}
				maxprofondeur[i] = max1 + 1;
				chemin[i] = Math.max(max1 + max2 + 2, chemin[i]);
			} }
}
```

```
//complexit� O(nm), mieux que solution par couplage
public class CouplageMaxBiparti {
	// nb sommets a gauche et a droite
	static int n0, n1;
	// liste d'adjacence
	static int[][] e0, e1;
	// couplage, m0[u] est le sommet a droite auquel u est couple'
	static int[] m0, m1;
	static final int LIBRE = -1;
	// marquage des sommets deja visites pour le dfs
	static boolean[] deja;
	// dit si un chemin aug est trouv�, modifie le match en cons�quence
	static boolean dfs(int u) {
		deja[u] = true;
		for (int v : e0[u])
			if (m1[v] == LIBRE || !deja[m1[v]] && dfs(m1[v])) {
				m0[u] = v; // coupler u avec v
				m1[v] = u;
				return true;
			}
		return false;
	}
```

```
	static boolean cheminAugmentant(int u) {
		deja = new boolean[n0];
		return dfs(u);
	}
```

```
	static int maxBipartiteMatching() {
		int val = 0;
		m0 = new int[n0];
		m1 = new int[n1];
		Arrays.fill(m0, LIBRE);
		Arrays.fill(m1, LIBRE);
		for (int u = 0; u < n0; u++)
			if (cheminAugmentant(u))
				val++;
		return val;
	}}
```

```
public class ClosestPointANDConvexe {

	static class Point {
		double x;
		double y;

		Point(double a, double b) {
			x = a;
			y = b;
		}
// INUTILES POUR CONVEX

		double distance(Point a) {
			return Math.sqrt((a.x - x) * (a.x - x) + (a.y - y) * (a.y - y));
		}

		public static class ComparatorP implements Comparator<Point> {

			public int compare(Point a, Point b) {
				int dx = (int) Math.signum(a.x - b.x);
				int dy = (int) Math.signum(a.y - b.y);
				return dx != 0 ? dx : dy;
			}
		}
	}

	// Prints pair of closest points
	static void pair(Point[] tab) {
		Arrays.sort(tab, new Point.ComparatorP());
		double dist = tab[0].distance(tab[1]);
		Point sol0 = tab[0], sol1 = tab[1];

		TreeSet<Point> B = new TreeSet<Point>(new Point.ComparatorP());

		for (Point p : tab) {
			Point low = new Point(p.x, p.y - dist);
			Point up = new Point(p.x, p.y + dist);
			for (Point q : B.subSet(low, up)) {
```

```
				if (q.y < p.y - dist)
					B.remove(q);
				else {
					double d = p.distance(q);
					if (d < dist) {
						dist = d;
						sol0 = q;
						sol1 = p;
					}
				}
			}
			B.add(p);
		}
		System.out.println(sol0.x + " " + sol0.y);
		System.out.println(sol1.x + " " + sol1.y);
	}

	// sert pour convexe
	static double cross(Point a, Point b, Point c) {
		return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
	}

	// enveloppe convexe de t
	static LinkedList<Point> convexe(Point[] t) {
		Arrays.sort(t, new Comparator<Point>() {
			public int compare(Point a, Point b) {
				int dx = (int) Math.signum(a.x - b.x);
				int dy = (int) Math.signum(a.y - b.y);
				return dx != 0 ? dx : dy;
			}
		});
		Point[] top = new Point[t.length];
		Point[] bot = new Point[t.length];
		int ntop = 0, nbot = 0;
		for (Point p : t) {
			while (ntop >= 2 && cross(top[ntop - 2], top[ntop - 1], p) >= 0)

				ntop--;
			top[ntop++] = p;
```

```java
			while (nbot >= 2 && cross(bot[nbot - 2], bot[nbot - 1], p)
<= 0)
				nbot--;
			bot[nbot++] = p;
		}
		LinkedList<Point> ret = new LinkedList<Point>();
		for (int i = 0; i < nbot - 1; i++)
			ret.add(bot[i]);
		for (int i = ntop - 1; i > 0; i--)
			ret.add(top[i]);
		return ret;
	}
}
```

```java
public class RectangleSousHisto {
	// aire du plus grand rectangle
	static int largestRect(int[] x) {
		int n = x.length;
		int a[] = new int[n];
		int h[] = new int[n];
		int best = 0, s = 0;
		for (int i = 0; i < n; i++) {
			a[s] = i;
			while (s > 0 && h[s - 1] > x[i]) {
				s--; int area = (i - a[s]) * h[s];
				if (area > best) best = area;
			}
			h[s] = x[i]; s++;
		}
		return best;
	}
}
```

```java
public class SurfaceRectangle {
	// input : matrix with 0 and 1
	// output : area of the largest rect
	static int bestRect(int[][] x) {
		int n = x.length;
		int m = x[0].length;

		for (int i = 0; i < n; i++) {
			int k = 0;
			for (int j = 0; j < m; j++) {
				if (x[i][j] != 0)
					x[i][j] = ++k;
				else {
					x[i][j] = 0;
					k = 0;
				}
			}
		}
		for (int i = 0; i < n; i++) {
			for (int j = 0; j < m; j++) {
				System.out.print(x[i][j]);
			}
			System.out.println();
		}
		int best = 0;
		for (int j = 0; j < m; j++)
			best = Math.max(best,
RectangleSousHisto.largestRect(x[j]));
		return best;
	}

}
```

```
class FindingNemo_Sol {

   final static int L=201, MUR=-1, VIDE=0, PORTE=+1;
   static int sep[][][];

   // codage case (x,y) dans un entier p
   static int cx(int p)       {return p/L;}
   static int cy(int p)       {return p%L;}
   static int cp(int x, int y) {return x*L+y;}

   static boolean inBound(int x, int y) {
        return 1<=x && x<L-1 && 0<=y && y<L-1;
   }

   static int shortestPath01(int nx, int ny) {
        if (!inBound(nx,ny))
           return 0;
        boolean deja[] = new boolean[L*L]; // -- dÃ©jÃ  visitÃ© ?
        Deque<Integer> Q = new LinkedList<Integer>(); // -- sommets Ã  visiter
        int nemo = cp(nx,ny);
        Q.offer(nemo); //              -- commencer chez nemo
        int d[] = new int[L*L]; //  c     -- distance
        Arrays.fill(d,Integer.MAX_VALUE);
        d[nemo] = 0;
        while (!Q.isEmpty()) {
           int p = Q.pollFirst();
           int x = cx(p), y=cy(p);
           if (!inBound(x,y))
                return d[p]; //         -- trouvÃ© sortie
           if (deja[p])     //          -- ne pas traiter deux fois
                continue;
           deja[p] = true;
           //                           -- parcourir les 4 voisins
           int q[] = {cp(x-1,y),   cp(x,y-1),   cp(x,y+1),     cp(x+1,y)};
           int s[] = {sep[x][y][1], sep[x][y][0], sep[x][y+1][0], sep[x+1][y][1]};
           for (int i=0; i<4; i++) {
                if (s[i]==MUR || deja[q[i]])
                   continue;    // un point deja visite ou un mur? ignorer
                if (s[i]==VIDE) {
                   d[q[i]] = Math.min(d[q[i]], d[p]);
                   Q.offerFirst(q[i]);
                }
                else { // PORTE
                   d[q[i]] = Math.min(d[q[i]], d[p]+1);
                   Q.offerLast(q[i]);
                }
           }
        }
        return -1; // nemo ne peut pas quitter le labyrinthe
   }
}
```