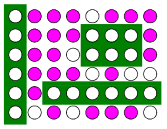
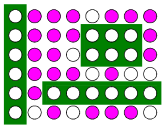


# *Tous ensemble au spectacle*

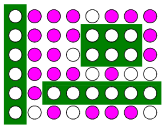


# Inspiration

**The Maximal Rectangle Problem**

David Vandevoorde, April 01, 1998 (Hewlett-Packard)

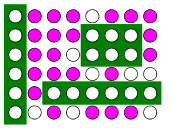
http://www.drdobbs.com/database/the-maximal-rectangle-problem/184410529



# Solution simple et naïve

- Pour chaque position possible du coin supérieur gauche (nbR x nbS possibilités)
  - Envisager chaque position possible du coin inférieur droit (nbR x nbS possibilités)
    - Et pour chaque rectangle ainsi défini, parcourir le rectangle pour vérifier qu'il ne contient que des 1 (nbR x nbS cases à vérifier).

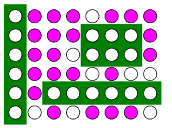
Bilan: $O(nbR^3 \times nbS^3)$



# Amélioration

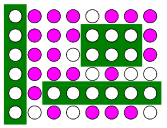
- Pour chaque position possible du coin supérieur gauche (nbR x nbS possibilités)
  - Considérer des rectangles de plus en plus grands, tant qu'ils ne contiennent que des 1.

Bilan: $O(nbR^2 \times nbS^2)$



# Solution de Geoffroy

```
public static void main(String[] args) {
  int[][] salle = readInput();
  int result= compute(salle);
  System.out.print(result);
}
public static int[][] readInput() {
  Scanner scan = new Scanner(System.in);
  int nbR = scan.nextInt();
  int nbS = scan.nextInt();
  int[][] salle = new int[nbR][nbS];
  for(int r = 0; r < nbR; ++r) {
    int val = 0;
    for(int s = 0; s < nbS; ++s) {
      if(scan.nextInt() == 0)
        val = 0;
      else
        val += 1;
      salle[r][s] = val;
    }
  }
  return salle;
}
```

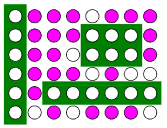


# Solution de Geoffroy

**Idée: On ne mémorise pas les 0 et les 1 mais, pour chaque siège, le nombre de sièges libres à sa gauche (lui compris)**

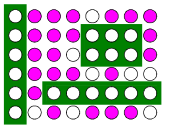
**Ainsi il ne faudra plus parcourir la rangée pour savoir s'il y a des sièges libres!**

```
1 0 1 1 1          1 0 1 2 3
0 1 1 1 1          0 1 2 3 4
1 1 1 1 1    ➔     1 2 3 4 5
0 0 1 1 1          0 0 1 2 3
1 1 0 0 1          1 2 0 0 1
```



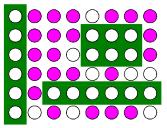
# Solution de Geoffroy

```
public static int compute(int[][] salle) {
  int nbS = salle[0].length;
  int rectY = salle.length;
  int nbR= 0;
  for(int y = nbR - 1; y >= 0; --y) {
    for(int x = nbS - 1; x >= 0; --x){
      int temp = bestRectangle(salle, x, y);
      if(temp > result)
          result = temp;
    }
  }
  return result;
}
```



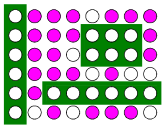
# Solution de Geoffroy

```
public static int bestRectangle(int[][] rectangle, int xPos, int yPos) {
  int hauteur = 1;
  int largeur = rectangle[yPos][xPos];
  int result  = 0;
  for(int y = yPos; y >= 0 ; --y) {
    int largeurRangee = rectangle[y][xPos];
    if(largeurRangee == 0) //rectangle sans largeur !
      break;
    if(largeurRangee < largeur)
      largeur = largeurRangee;
    int surface = hauteur * largeur;
    if(surface > result)
      result = surface;
    ++hauteur;
  }
  return result;
}
```



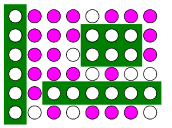
# Solution de Geoffroy

Bilan: $O(nbR^2 \times nbS)$



# Solution optimale

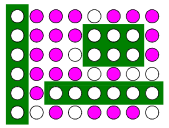
O(nbR x nbS)



# Solution optimale

```
int maxFreeSeats(int nbR, int nbS) {
  int[] heights = new int[nbS + 1];
  int nbSeats = 0;
  for(int r = 0; r < nbR; ++r) {
    loadNextRow(heights); // Lecture ligne par ligne
    nbSeats = Math.max(nbSeats, maxInHisto(heights));
  }
  return nbSeats;
}

// L'algorithme est linéaire. Il n'est pas nécessaire de mémoriser toute la
   salle.
void loadNextRow(int[] heights) {
  for(int s = 0; s < heights.length - 1; ++s)
    if(scan.nextInt() == 1) ++heights[s];
    else heights[s] = 0;
}
```



# Solution optimale

```
// Pré: height[height.length - 1] == 0
static int maxInHisto(int[] heights) {
  int len = heights.length;
  // Une pile de sommet top (java.util.Stack très innefficace)
  int[] lefts = new int[len + 1];
  lefts[0] = heights.length - 1;
  int top = 1;
  int maxArea = 0;
  int right = 0;
  while(right < len)
    if(top == 1 || heights[right] >= heights[lefts[top-1]])
      lefts[top++] = right++;
    else {
      int h = heights[lefts[--top]];
      int w = (top == 1 ? right : right - lefts[top - 1] - 1);
      maxArea = Math.max(maxArea, h * w);
    }
  return maxArea;
}
```