Informatique et Programmation

Cours 10

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-py

Plan

- algorithme glouton (marche du cavalier)
- exploration exhaustive (les 8 reines)
- programmation dynamique (plus longue chaîne commune)

dès maintenant: télécharger Python 3 en `http://www.python.org`

Exploration

on distingue 3 méthodes d'exploration

- algorithmes **gloutons**
  - un choix local permet d'obtenir la solution globale

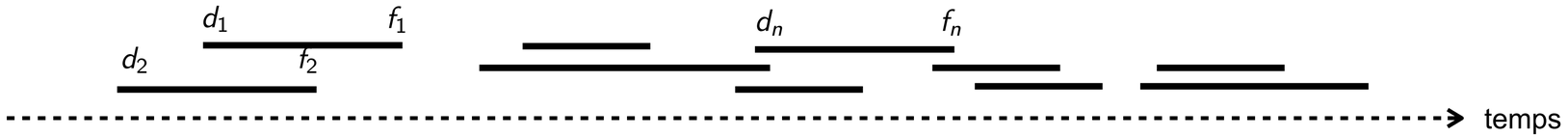
- exploration **exhaustive**
  - on parcourt les solutions globales jusqu'à trouver la bonne solution
  - retours arrière possibles (*backtracking*)

- programmation **dynamique**
  - on mémorise tous les résultats partiels pour obtenir la solution globale
  - demande de la mémoire supplémentaire

Allocation de ressource

Problème

- on gère un magasin et un seul ticket « super-bonus »
- les clients $c_1, c_2, \cdots c_n$ rentrent aux temps $d_1, d_2, \cdots d_n$ et partent aux temps $f_1, f_2, \cdots f_n$
- on veut donner le tickeet super-bonus au maximum de clients



Solution

- on trie les clients par ordre croissant de dates de fin
- et on prend les clients dans cet ordre avec la contrainte :

$$i < j \implies f_i < d_j$$

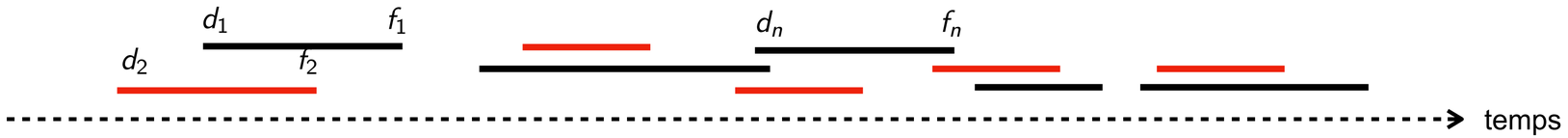
[plus compliqué si plusieurs ressources]

algorithme glouton

Allocation de ressource

Problème

- on gère un magasin et un seul ticket « super-bonus »
- les clients $c_1, c_2, \cdots c_n$ rentrent aux temps $d_1, d_2, \cdots d_n$ et partent aux temps $f_1, f_2, \cdots f_n$
- on veut donner le tickeet super-bonus au maximum de clients



Solution

- on trie les clients par ordre croissant de dates de fin
- et on prend les clients dans cet ordre avec la contrainte :

$$i < j \implies f_i < d_j$$

[plus compliqué si plusieurs ressources]

algorithme glouton

Exercice

- montrer que cet algorithme donne la solution maximale !

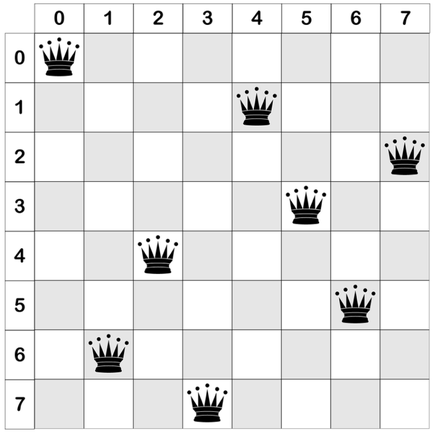
Les 8 reines

- les 8 reines (placer 8 reines sur un échiquier sans qu'elle ne soit en prise par une autre reine
  on explore les solutions avec possibles retours arrière (***backtracking***)

```
def compatible (i, j, pos) :
    for k in range (i) :
       if conflit (k, pos[k], i, j) :
          return False
    return True

def reines (n, i, pos) :
   if i >= n :
      imprimerSolution (pos);
      raise Exception
   else :
      for j in range (n) :
        if compatible (i, j, pos) :
          pos[i] = j
          reines (n, i+1, pos)
          pos[i] = None
```

ici *backtracking* si pas de solution à partir de ligne `i+1` (on passe alors à la colonne suivante)



`pos == [0, 4, 7, 5, 2, 6, 1, 3]`

```
def nReines (n) :
    try :
        pos = [None for _ in range(n)]
        pos[0] = 6
        reines (n, 1, pos)
    except :
        pass
```

on imprime une solution en partant de la reine en (0, 6)

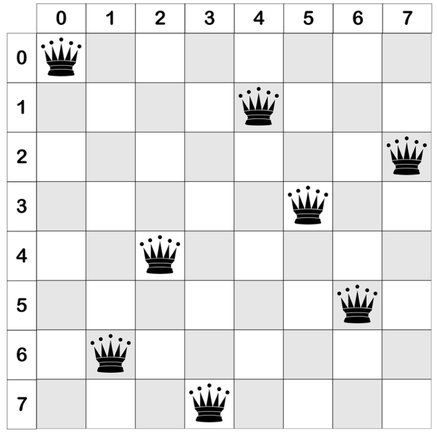
Les 8 reines

- impression de la solution

```
def print_matrix (a) :
    for line in a :
        for elt in line :
            print ( '%s ' % elt, end = ' ')
        print ()

def new_matrix (m, n, v) :
    a = [ [v for j in range (n) ] for i in range (m) ]
    return a

def imprimerSolution (pos) :
    n = len (pos)
    a = new_matrix1 (n, n, ".")
    for i in range (n) :
        if pos[i] != None:
            a[i][pos[i]] = "R"
    print_matrix1 (a)
    print ("----------------------------")
```

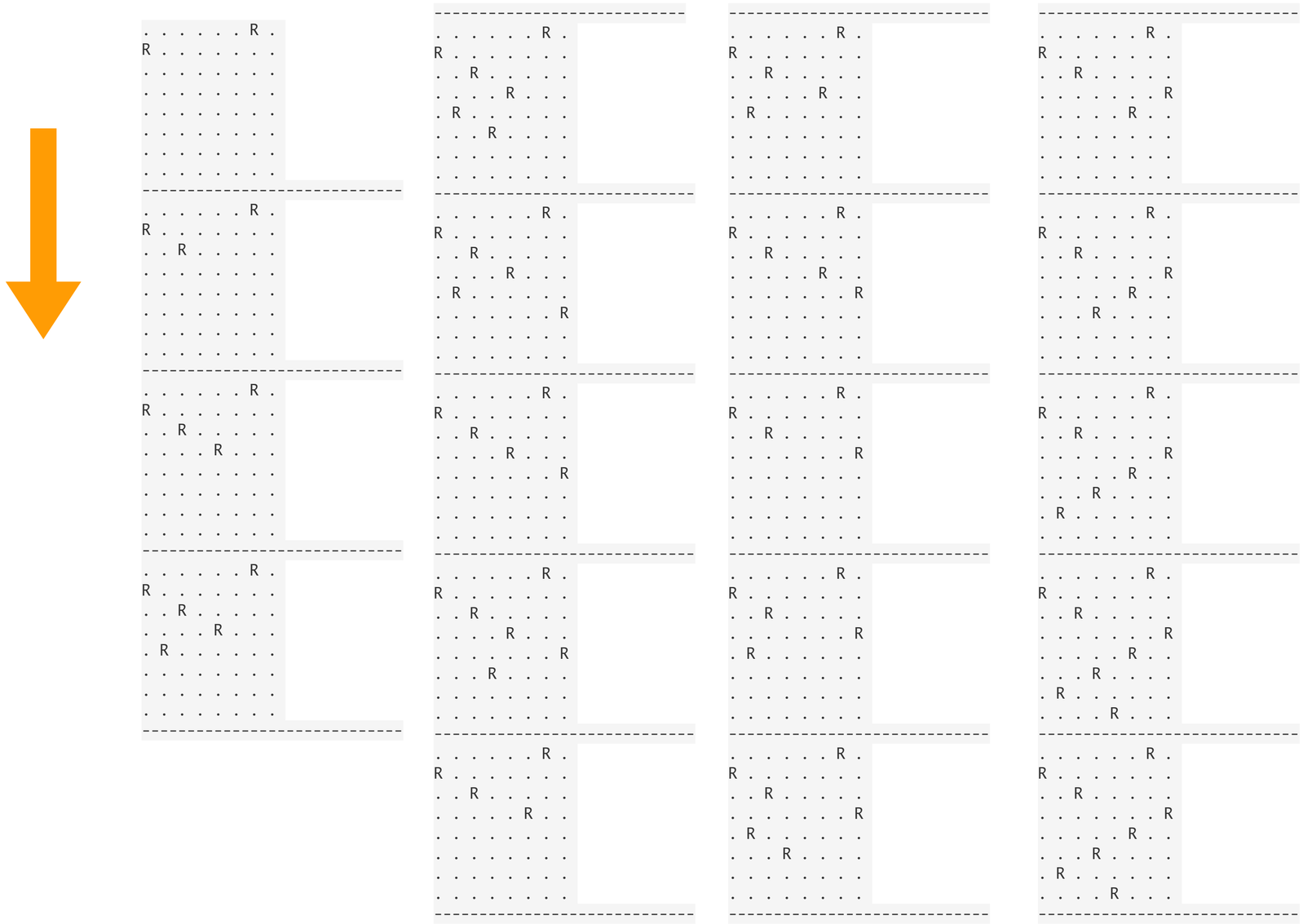


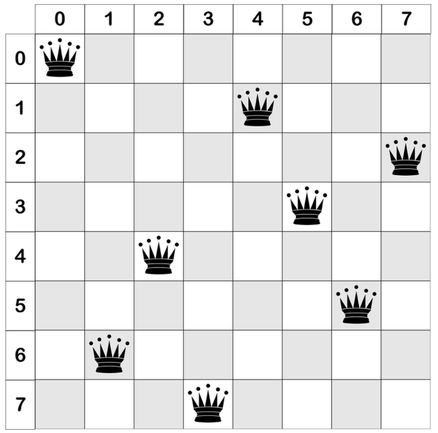

```
pos == [0, 4, 7, 5, 2, 6, 1, 3]
```

```
R . . . . . . .
. . . . R . . .
. . . . . . . R
. . . . . R . .
. . R . . . . .
. . . . . . R .
. R . . . . . .
. . . R . . . .
```

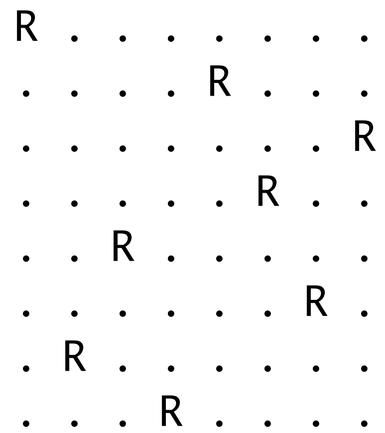
Les 8 reines

- avancées et retours en arrière de `reines (n, i, pos)`



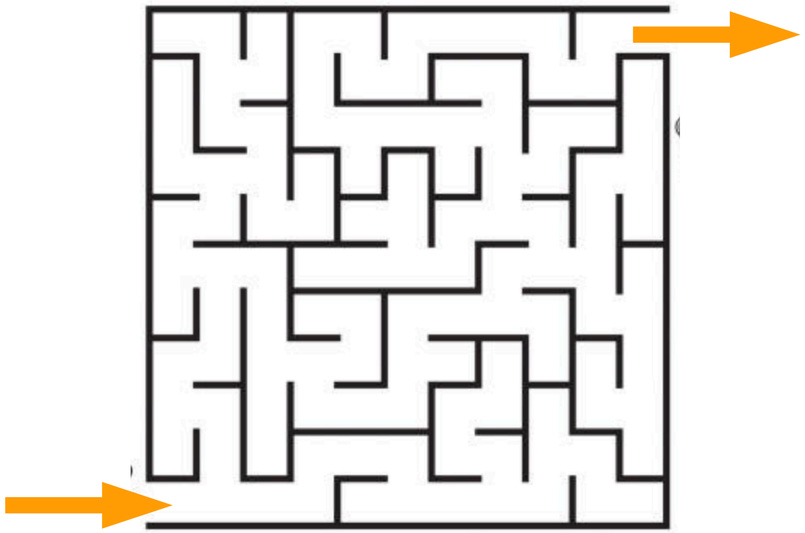


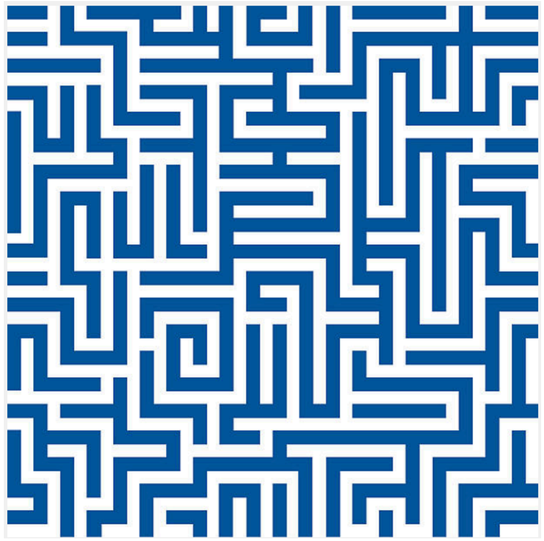
`pos == [0, 4, 7, 5, 2, 6, 1, 3]`

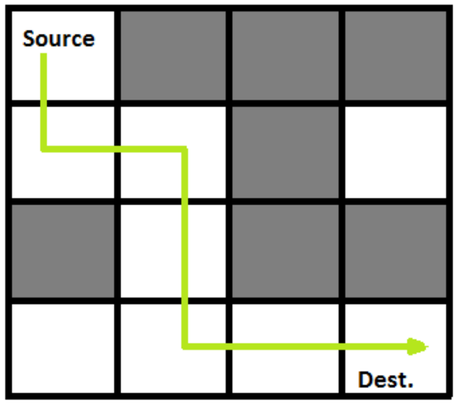


Sortie de labyrinthe

- trouver le chemin vers la sortie !







- le chemin vers la sortie commence par une case libre et un chemin de cette case vers la sortie

- il faut laisser une marque pour ne pas passer 2 fois par la même case

[algorithme du fil d’Ariane]

Sortie de labyrinthe

- représentation du labyrinthe par une matrice de 0 (blanc) et 1 (noir)

```
def chemin (a, p, q, dejaVu) :
    dejaVu [p[0]][p[1]] = True
    if p == q :
        return [p]
    for r in voisins (a, p) :
        if not dejaVu [r[0]][r[1]] :
            ch = chemin (a, r, q, dejaVu)
            if ch != [] :
               return [r] + ch
    return []
```

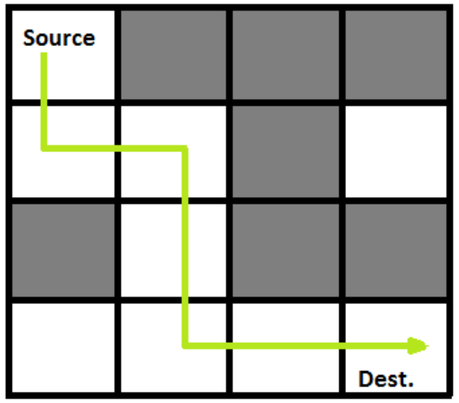
concaténation des listes (*append*)

```
def uneSolution (a, p, q) :
    m = len (a); n = len (a[0])
    dejaVu = new_matrix (m, n, False)
    ch = chemin (a, p, q, dejaVu)
    if ch != [] :
        return [p] + ch
    return []
```

concaténation des listes (*append*)

```
print (uneSolution (maze, (0, 0), (3, 3)))
```

```
maze = [[0, 1, 1, 1],
        [0, 0, 1, 0],
        [1, 0, 1, 1],
        [0, 0, 0, 0]]
```

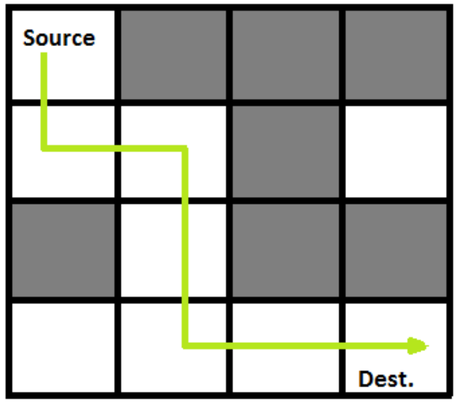


[algorithme du fil d'Ariane]

Sortie de labyrinthe

- représentation du labyrinthe par une matrice de 0 (blanc) et 1 (noir)

```
maze = [[0, 1, 1, 1],
        [0, 0, 1, 0],
        [1, 0, 1, 1],
        [0, 0, 0, 0]]
```



[algorithme du fil d'Ariane]

```
def voisins (a, p) :
    m = len(a); n = len(a[0])
    res = []
    i = p[0]; j = p[1]
    for r in [(i-1, j), (i, j-1), (i, j+1), (i+1, j)] :
        i1 = r[0]; j1 = r[1]
        if 0 <= i1 < m and 0 <= j1 < n and a[i1][j1] == 0 :
            res = res + [r]
    return res
```

concaténation des listes (*append*)

Sortie de labyrinthe

- représentation du labyrinthe par une matrice de 0 (blanc) et 1 (noir)

```
def chemin (a, p, q, dejaVu) :
    dejaVu [p[0]][p[1]] = True
    if p == q :
        return [p]
    for r in voisins (a, p) :
        if not dejaVu [r[0]][r[1]] :
            ch = chemin (a, r, q, dejaVu)
            if ch != [] :
               return [r] + ch
    return []
```

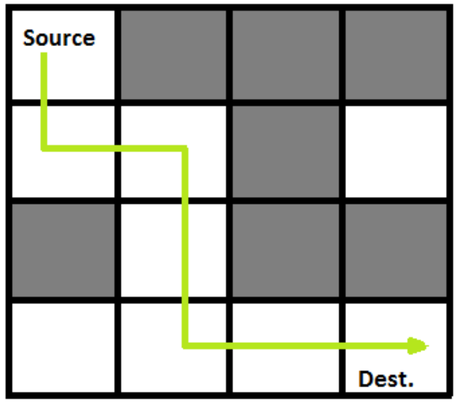
concaténation des listes (*append*)

```
def uneSolution (a, p, q) :
    m = len (a); n = len (a[0])
    dejaVu = new_matrix (m, n, False)
    ch = chemin (a, p, q, dejaVu)
    if ch != [] :
        return [p] + ch
    return []
```

concaténation des listes (*append*)

```
print (uneSolution (maze, (0, 0), (3, 3)))
```

```
maze = [[0, 1, 1, 1],
        [0, 0, 1, 0],
        [1, 0, 1, 1],
        [0, 0, 0, 0]]
```



[algorithme du fil d'Ariane]

```
def voisins (a, p) :
    m = len(a); n = len(a[0])
    res = []
    i = p[0]; j = p[1]
    for r in [(i-1, j), (i, j-1), (i, j+1), (i+1, j)] :
        i1 = r[0]; j1 = r[1]
        if 0 <= i1 < m and 0 <= j1 < n and a[i1][j1] == 0 :
            res = res + [r]
    return res
```

concaténation des listes (*append*)

Sortie de labyrinthe

- mauvaise programmation avec des variables globales

```
def chemin (p, q) :
    dejaVu [p[0]][p[1]] = True
    if p == q :
        return [p]
    for r in voisins (p) :
        if not dejaVu [r[0]][r[1]] :
            ch = chemin (r, q)
            if ch != [] :
               return [r] + ch
    return []
```

concaténation des listes (*append*)

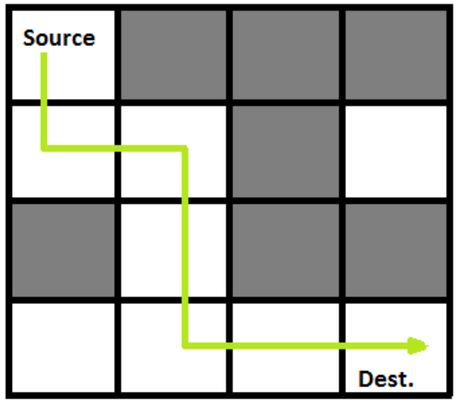
```
def uneSolution (p, q) :
    for i in range (len (maze)); :
       for j in range (len (maze[0])) :
          dejaVu[i][j] = False
    ch = chemin (p, q)
    if ch != [] :
        return [p] + ch
    return []
```

concaténation des listes (*append*)

```
print (uneSolution ((0, 0), (3, 3)))
```

```
maze = [[0, 1, 1, 1],
        [0, 0, 1, 0],
        [1, 0, 1, 1],
        [0, 0, 0, 0]

dejaVu = new_matrix (4, 4, False)
```



[algorithme du fil d'Ariane]

```
def voisins (p) :
    m = len(a); n = len(a[0])
    res = []
    i = p[0]; j = p[1]
    for r in [(i-1, j), (i, j-1), (i, j+1), (i+1, j)] :
        i1 = r[0]; j1 = r[1]
        if 0 <= i1 < m and 0 <= j1 < n and a[i1][j1] == 0 :
            res = res + [r]
    return res
```

concaténation des listes (*append*)

Sortie de labyrinthe

- génération de labyrinthe

```
import random

def generate_maze (m, n) :
    a = new_matrix3 (m, n, 0)
    for i in range (m) :
        for j in range (n) :
            a[i][j] = random.choice ([0, 1])
    print_maze (a)
    return a

maze1 = generate_maze (10, 10)
```

- avec impression

```
def print_maze (a) :
    m = len (a); n = len (a[0])
    b = new_matrix3 (m, n, ".")
    for i in range (m) :
        for j in range (n) :
            b[i][j] = "*" if a[i][j] == 1 else "."
    print_matrix1 (b)
    print ("----------------------------")
```

```
. .   *   *   *   .   *   *   *   *
*     .   .   .   *   .   .   .   .
*   *   *   .   *   *   *   *   .   *
*   .   .   .   *   *   .   .   *   *
*   .   *   .   *   *   .   .   .   *
*   .   .   .   .   .   .   .   .   *
*   .   .   *   *   .   *   *   .   .
.   *   .   .   *   .   *   *   *   *
.   .   *   .   *   *   *   *   *   *
.   .   *   *   .   .   *   *   .   .
----------------------------
```

```
m1 = [[0, 0, 1, 1, 1, 0, 1, 1, 1, 1],
      [1, 0, 0, 0, 1, 0, 0, 0, 0, 0],
      [1, 1, 1, 0, 1, 1, 1, 1, 0, 1],
      [1, 0, 0, 0, 1, 1, 0, 0, 1, 1],
      [1, 0, 1, 0, 1, 1, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
      [1, 0, 0, 1, 1, 0, 1, 1, 0, 0],
      [0, 1, 0, 0, 1, 0, 1, 1, 1, 1],
      [0, 0, 1, 0, 1, 1, 1, 1, 1, 1],
      [0, 0, 1, 1, 0, 0, 1, 1, 0, 0]]
```

Exercice

- faire une sortie graphique

Programmation dynamique

- plus longue sous-séquence commune entre 2 chaînes de caractères (commande Unix `diff`)

  [on mémorise les solutions partielles — $m \times n$ opérations]

```
GAUCHE = 1
HAUT = 2
DIAG = 3
```

```
def new_matrix (m, n, v) :
    a = [ [v for _ in range (n) ] for _ in range (m) ]
    return a
```

```
def longueurSSC (u, v) :
  m = len(u); n = len(v)
  lg = new_matrix (m+1, n+1, 0)
  p = new_matrix (m+1, n+1, 0)
  for i in range(1, m+1):
    for j in range(1, n+1):
      if u[i-1] == v[j-1]:
        lg[i][j] = 1 + lg[i-1][j-1]
        p[i][j] = DIAG;
      elif lg[i][j-1] > lg[i-1][j]:
        lg[i][j] = lg[i][j-1]
        p[i][j] = GAUCHE
      else :
        lg[i][j] = lg[i-1][j]
        p[i][j] = HAUT
  return (lg[m][n], p)
```

u = ‘abcadefg’ v = ‘fbcexyg’

lg

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 3 3 3 3
0 1 1 2 3 3 3 3
0 1 1 2 3 3 3 4
```

p

```
0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2
0 2 3 1 1 1 1 1
0 2 2 3 1 1 1 1
0 2 2 2 2 2 2 2
0 2 2 2 2 2 2 2
0 2 2 2 3 1 1 1
0 3 2 2 2 2 2 2
0 2 2 2 2 2 2 3
```

Programmation dynamique

- plus longue sous-séquence commune entre 2 chaînes de caractères (commande Unix `diff`)

  [on mémorise les solutions partielles — $m \times n$ opérations]

```
def ssc (u, v) :
    m = len(u); n = len(v)
    lgp = longueurSSC (u, v)
    lg = lgp[0]; p = lgp[1]
    r = ''; i = m; j = n;
    while lg > 0 :
        if p[i][j] == DIAG :
            r = u[i-1] + r
            i = i - 1; j = j - 1;
            lg = lg - 1
        elif p[i][j] == GAUCHE :
            j = j - 1
        else :
            i = i - 1
    return r


print (ssc ('abcadefg', 'fbcexyg'))
```

u = ‘abcadefg’ v = ‘fbcexyg’

lg

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 2 2 2 2
0 0 1 2 3 3 3 3
0 1 1 2 3 3 3 3
0 1 1 2 3 3 3 4
```

p

```
0 0 0 0 0 0 0 0
0 2 2 2 2 2 2 2
0 2 3 1 1 1 1 1
0 2 2 3 1 1 1 1
0 2 2 2 2 2 2 2
0 2 2 2 2 2 2 2
0 2 2 2 3 1 1 1
0 3 2 2 2 2 2 2
0 2 2 2 2 2 2 3
```

à faire

- retour sur les objets et les arbres
- analyses lexicale et syntaxique
- modularité et programmation objet
- programmation graphique
- algorithmes géométriques
- calculs flottants et méthodes numériques
- programmation de plusieurs fils de calcul
- assertions et logique des programmes
- introduction à l'informatique théorique
- etc

vive l’informatique

et

la programmation !