

Informatique et Programmation

Cours 10

Jean-Jacques Lévy

`jean-jacques.levy@inria.fr`

`http://jeanjacqueslevy.net/prog-py`

Plan

- algorithme glouton (allocation de ressource)
- exploration exhaustive (les 8 reines — suite, sortie de labyrinthe)
- programmation dynamique (plus longue chaîne commune)

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

Devoir maison

Question 2 Quels sont les nombres de base A, C, T, G dans l'exemple **ex1** d'ADN.

Question 5 Quels sont les nombres des motifs GCA et CATCTCAGA dans l'exemple **ex1** d'ADN.

```
print (nb_base ('A', s))
print (nb_base ('C', s))
print (nb_base ('T', s))
print (nb_base ('G', s))
print ('-----')
print (nb_seq ('GCA', s))
print (nb_seq ('CATCTCAGA', s))
```

```
TATTTACCATATCAGATTCACATTCAGTCCTCAGCAAAATGAAGGGCTCCATTTTCACTCTGTTTTTATT
TATTTACCATATCAGATTCACATTCAGTCCTCAGCAAAATGAAGGGCTCCATTTTCACTCTGTTTTTATT
CTCTGTCCTATTTGCCATCTCAGAAGTGCGGAGCAAGGAGTCTGTGAGACTCTGTGGGCTAGAATACATA
CGGACAGTCATCTATATCTGTGCTAGCTCCAGGTGGAGAAGGCATCAGGAGGGGATCCCTCAAGCTCAGC
AAGCTGAGACAGGAAACTCCTTCCAGCTCCACATAAACGTGAGTTTTTCTGAGGAAAATCCAGCGCAAAA
CCTTCCGAAGGTGGATGCCTCAGGGGAAGACCGTCTTTGGGGTGGACAGATGCCCCTGAAGAGCTTTGG
AAGTCAAAGAAGCATTTCAGTGATGTCAAGACAAGATTTACAACTTTGTGTTGCACTGATGGCTGTTCCA
TGACTGATTTGAGTGCTCTTTGCTAAGACAAGAGCAAATACCCAATGGGTGGCAGAGCTTTATCACATGT
TTAATTACAGTGTTTTTACTGCCTGGTAGAACACTAATATTGTGTTATTAATAATGATGGCTTTTGGGTAGG
CAAACTTCTTTTCTAAAAGGTATAGCTGAGCGGTTGAAACCACAGTGATCTCTATTTTCTCCCTTTGCC
TAAAAATGCTATAAACCA
```



```
200
151
193
155
-----
11
1
```

Exploration

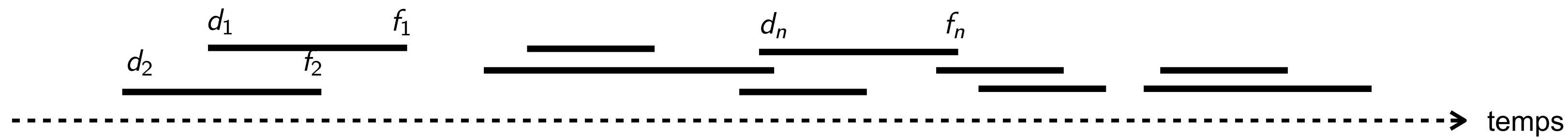
on distingue 3 méthodes d'exploration

- 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 $c_1, c_2, \dots, c_n$ rentrent aux temps $d_1, d_2, \dots, d_n$ et partent aux temps $f_1, f_2, \dots, f_n$
- on veut donner le ticket 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 :

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

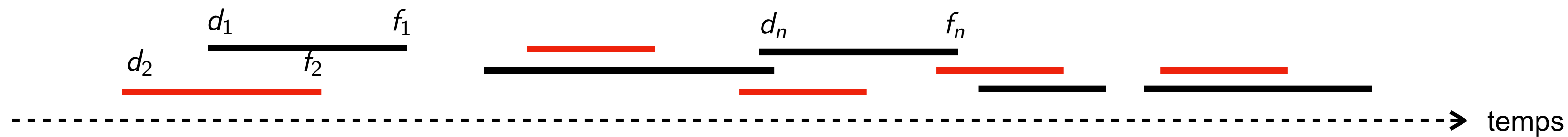
[plus compliqué si plusieurs ressources]

← algorithme glouton

Allocation de ressource

Problème

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

Recherche exhaustive

- problème du sac à dos (ranger le maximum d'objets dans un sac)
- voyageur de commerce et tous les problèmes NP
- les 8 reines (placer 8 reines sur un échiquier sans qu'elle ne soit en prise par une autre reine)

```
def conflit (i1, j1, i2, j2) :  
    return (i1 == i2) or (j1 == j2) or \  
        (abs (i1 - i2) == abs (j1 - j2))
```



teste si la reine en (i2, j2) peut
prendre la reine en (i1, j1)

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

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 :  
        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)

	0	1	2	3	4	5	6	7
0	♔							
1					♔			
2								♔
3						♔		
4			♔					
5							♔	
6		♔						
7				♔				

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

```
def nReines (n, j0) :  
    try :  
        pos = [None for _ in range(n)]  
        pos[0] = j0  
        reines (n, 1, pos)  
    except :  
        imprimerSolution (pos)
```

Les 8 reines

- impression de la solution

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

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

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

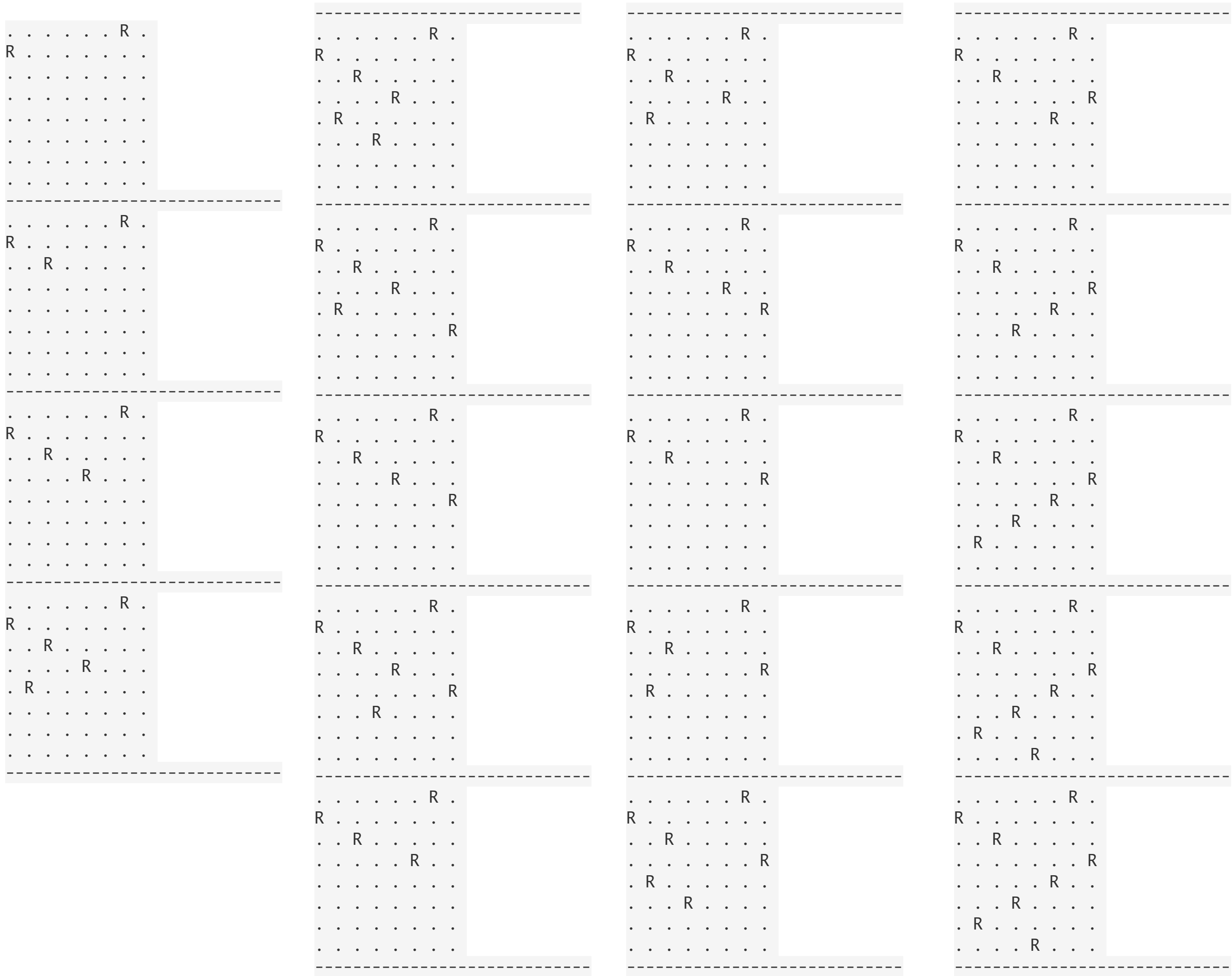
	0	1	2	3	4	5	6	7
0	♔							
1					♔			
2								♔
3						♔		
4			♔					
5							♔	
6		♔						
7				♔				

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

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

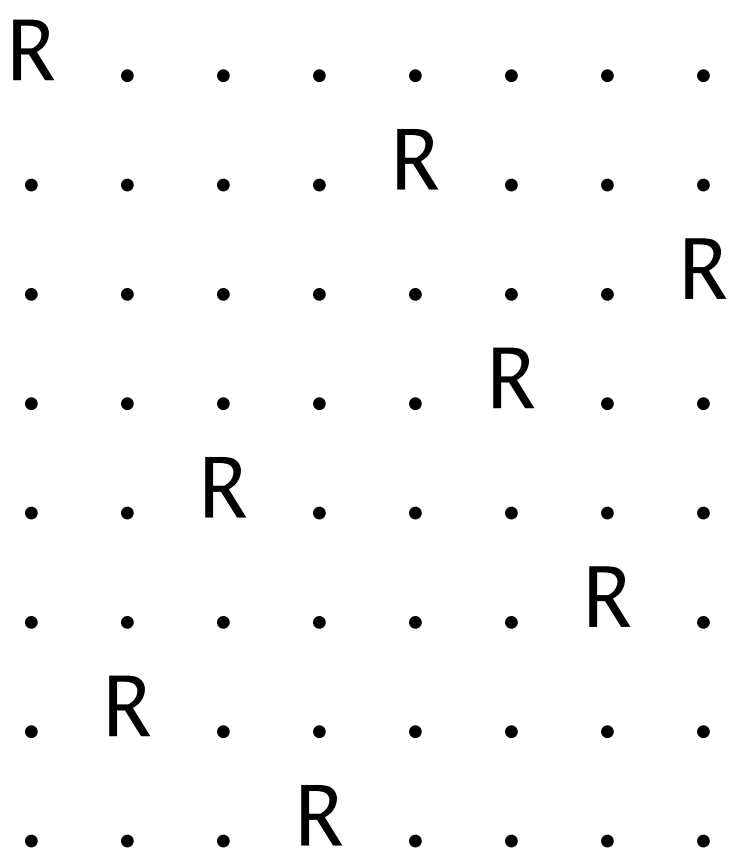
Les 8 reines

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



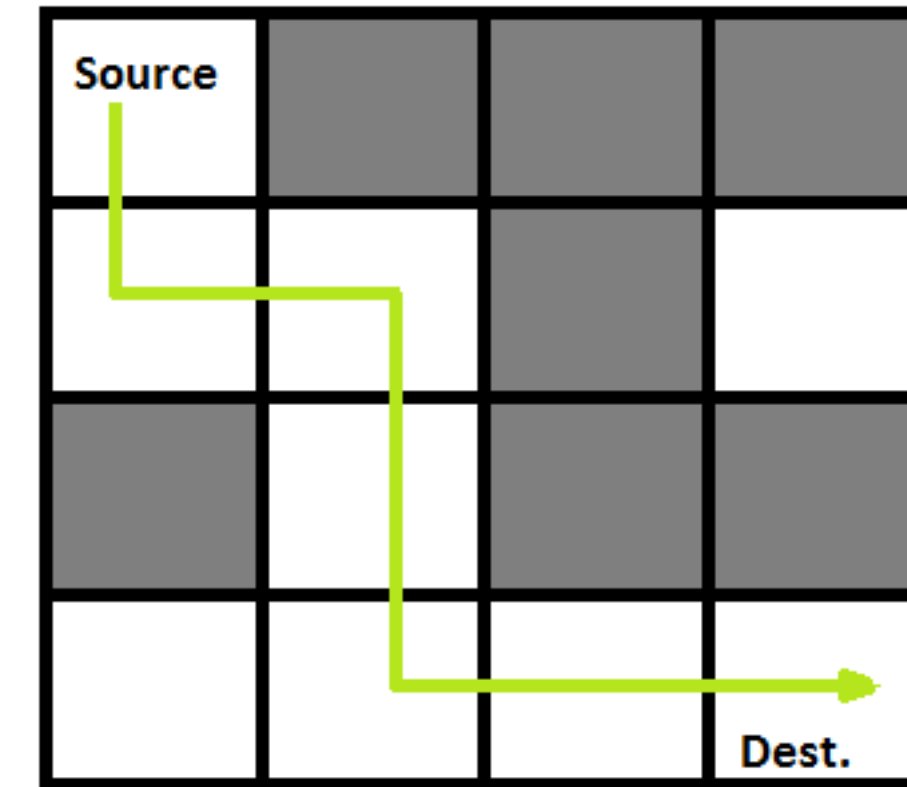
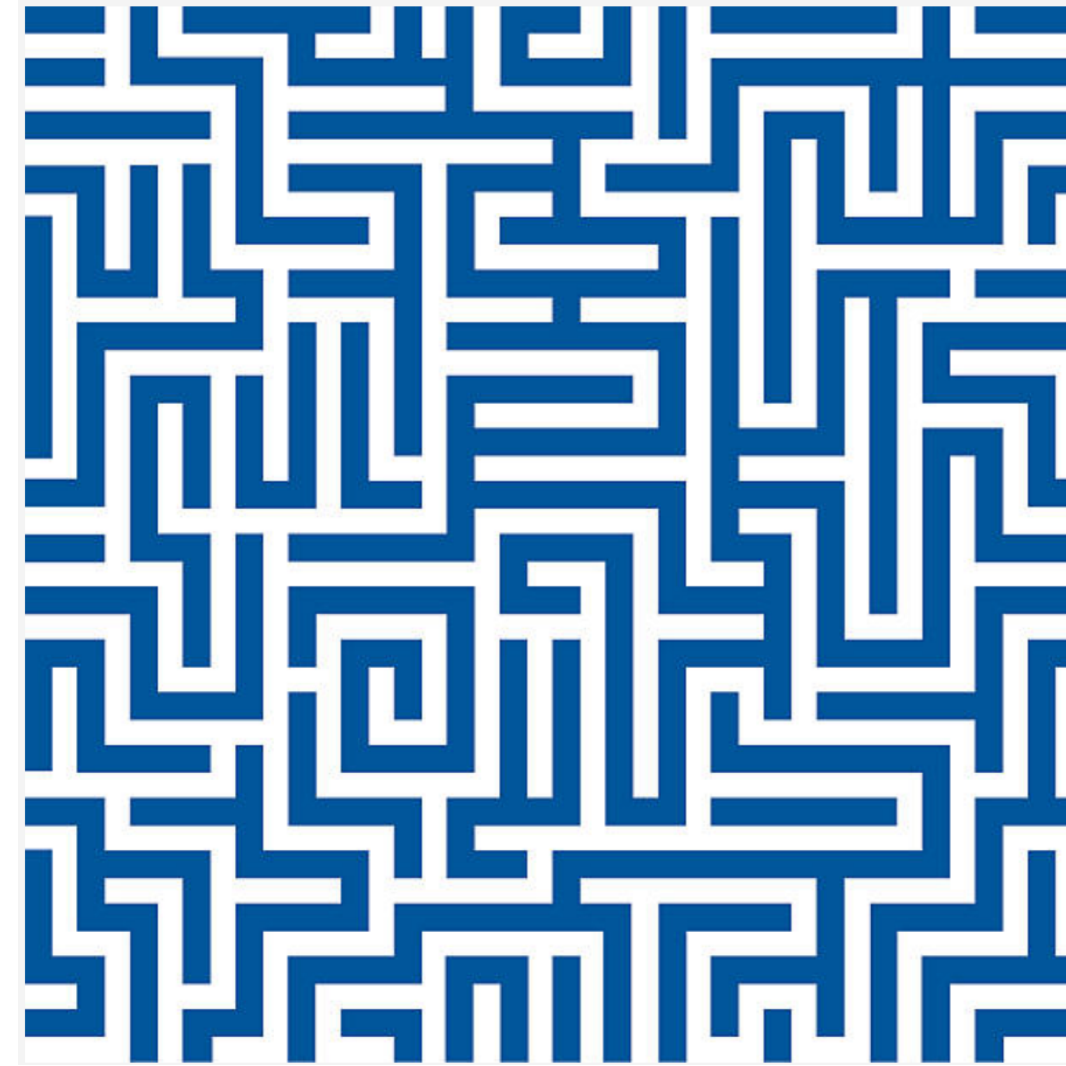
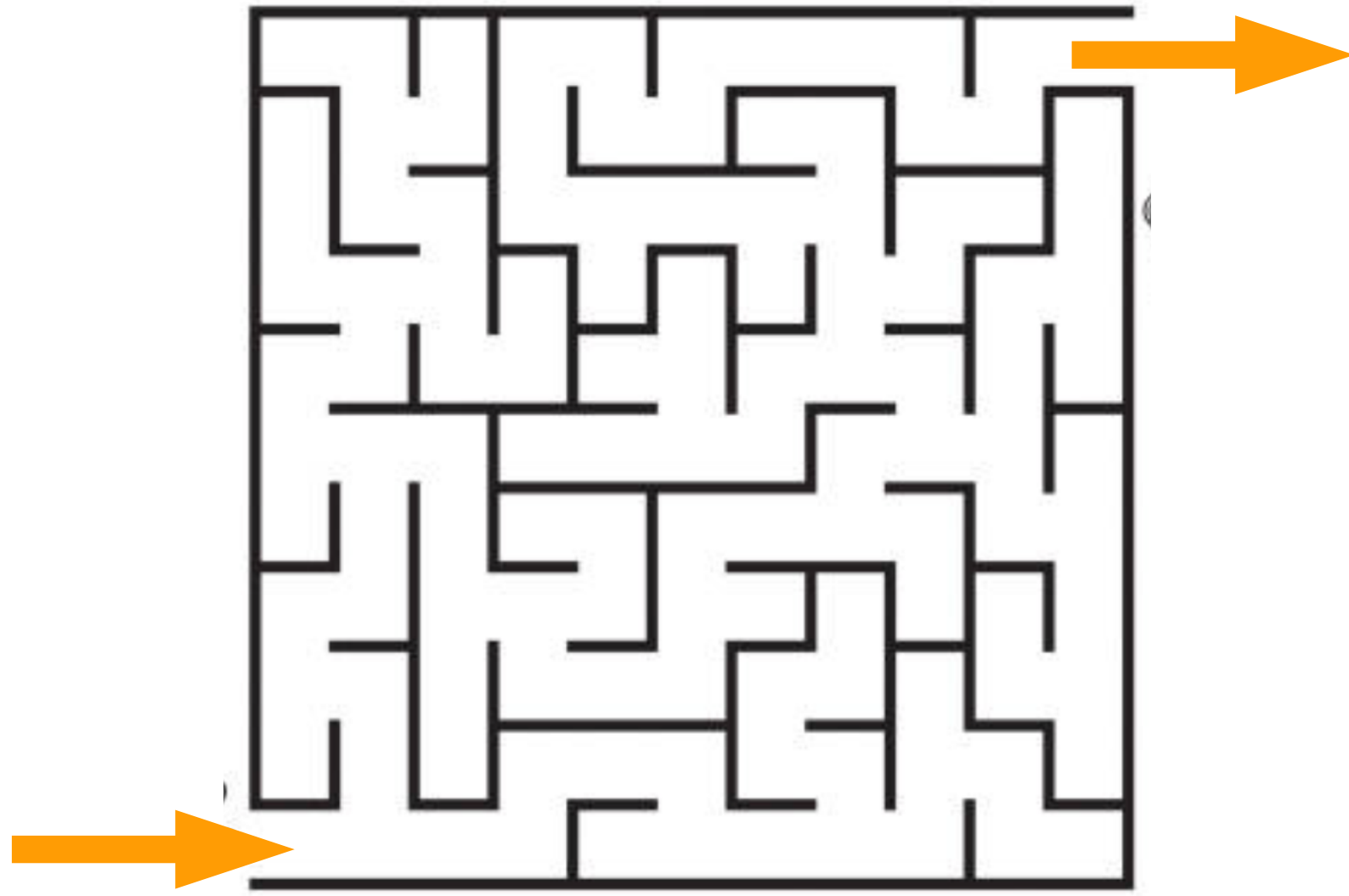
	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

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)

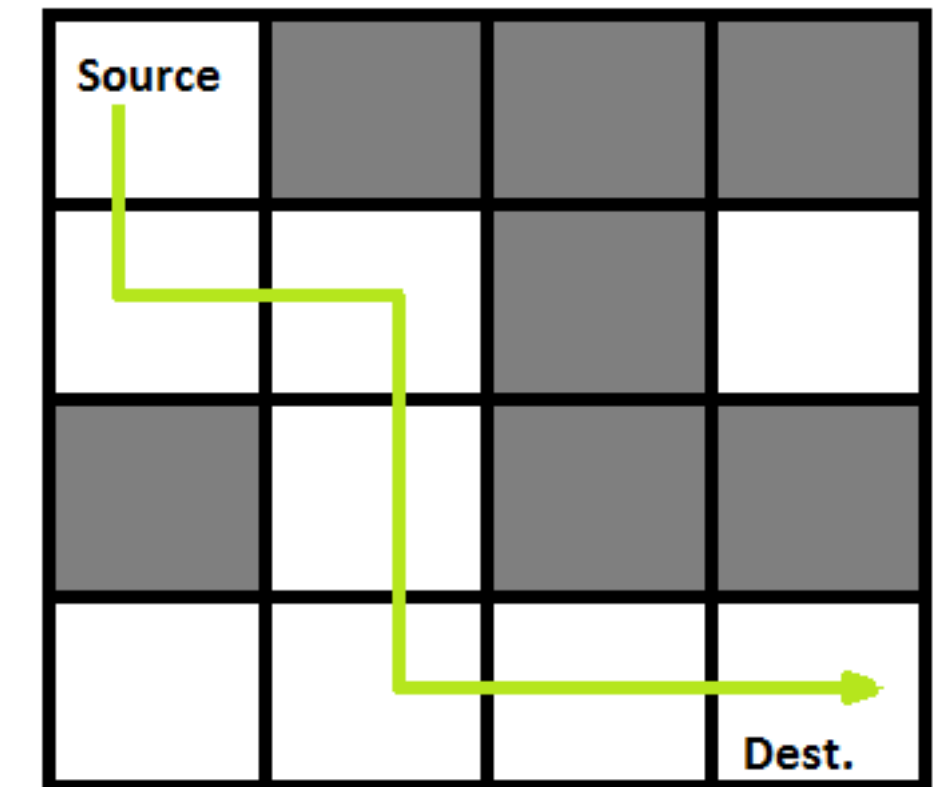
```
def chemin (a, p, q, dejaVu) :  
    dejaVu [p[0]][p[1]] = True  
    if p == q :  
        return [p]  
    for r in voisins (a, p) :  
        if not dejaVu [r[0]][r[1]] :  
            ch = chemin (a, r, q, dejaVu)  
            if ch != [ ] :  
                return [p] + ch  
    return [ ]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

ici *backtracking* si pas de chemin, on recommence avec un autre voisin

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

```
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

- génération de labyrinthe

```
import random

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

maze1 = generate_maze (10, 10)

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

print_maze (maze1)
```

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

```
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 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)  
    (lg, p) = longueurSSC (u, v)  
    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'

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

v = 'fbcexyg'

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 lexicales et syntaxiques
- 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 !