Structures de données et algorithmes fondamentaux

03 – Chaînes de caractères et matrices

Anthony Labarre

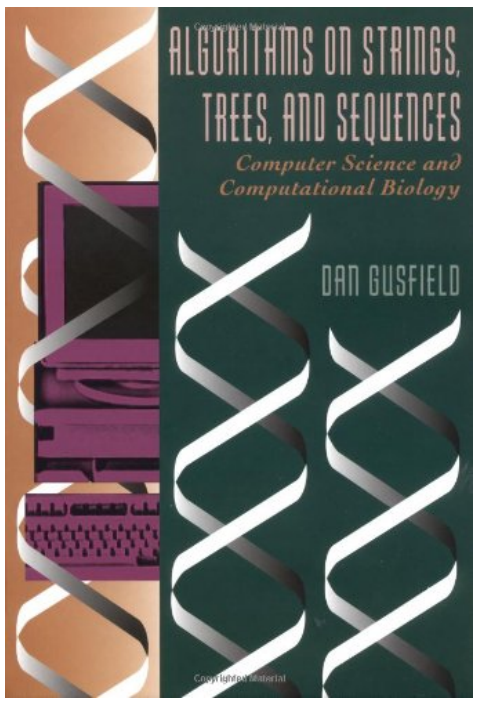
10 novembre 2015

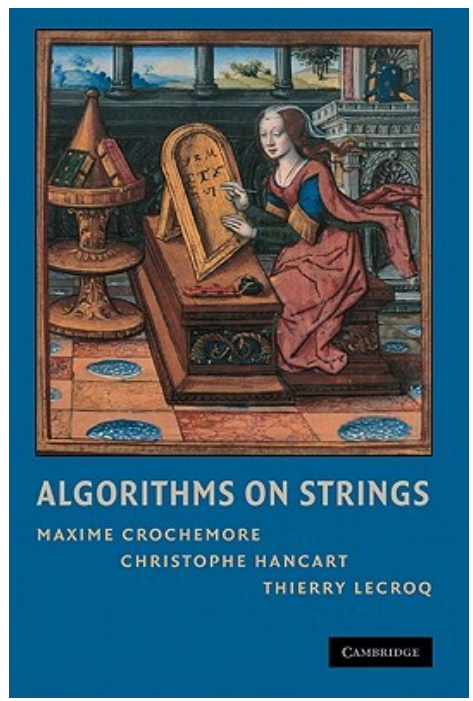
UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE

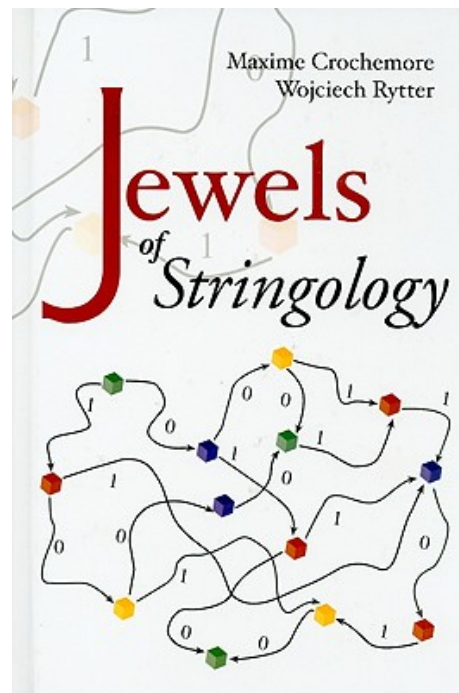
iut Marne-la-Vallée

Encodage de texte

- On peut représenter du texte de plusieurs manières :
  - par un tableau (ou une liste) de caractères ;
  - par une chaîne de caractères directement ;
- L'algorithmique du texte est un domaine riche et appliqué ;







Comparaison de chaînes

- En Python, on peut comparer directement deux chaînes `s` et `t` grâce à l'opérateur `==` ;
- Ceci n'est pas vrai dans tous les langages ;
- Comment fait-on si on ne peut comparer que des caractères ?

Comparaison de deux chaînes

```
def egalite(S, T):
    '''Renvoie True si les deux chaînes ont le même
    contenu, False sinon.'''
    if len(S) != len(T):  # même longueur?
        return False
    # chercher une différence
    for i in range(len(S)):
        if S[i] != T[i]:
            return False
    return True
```

Recherche de chaînes dans un texte

- On veut résoudre le problème suivant :

Problème (recherche d'une chaîne dans un texte)

Données : une chaîne `P` de taille `k`, une chaîne `T` de taille `n` $\geq$ `k` ;
Question : est-ce que `T` contient `P` ? (si oui, renvoyer la position d'un `P[0]` dans `T`)

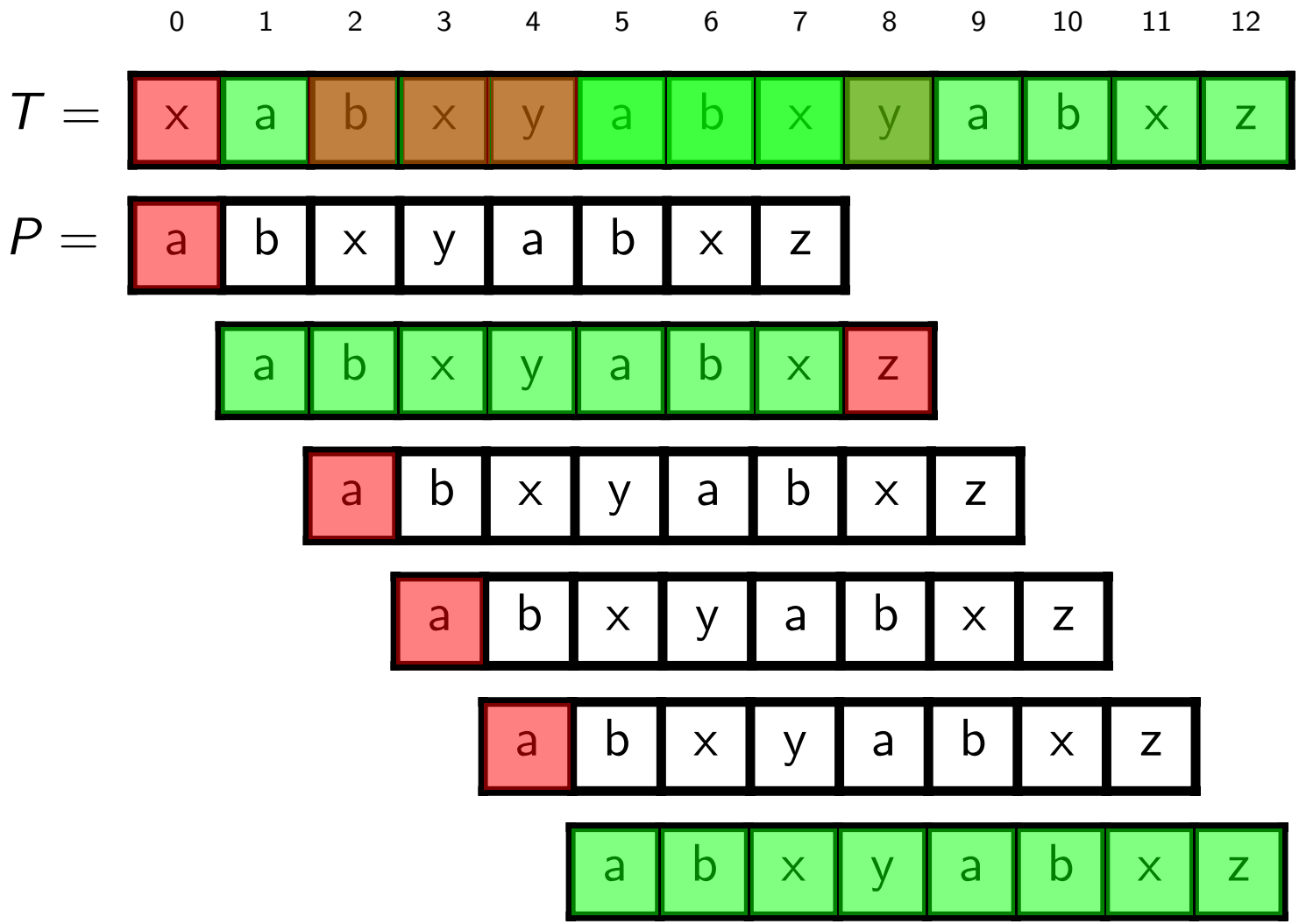
- Il s'agit d'une généralisation du problème de recherche d'un élément dans une liste ;
- Les applications devraient être assez évidentes ;

L'algorithme "naïf"

- L'idée de l'algorithme "naïf" consiste à essayer toutes les positions valides de T ;
  1. à chaque position `i`, on teste si
     $$(\texttt{T[i]}, \texttt{T[i+1]}, \ldots, \texttt{T[i+k-1]}) = (\texttt{P[0]}, \texttt{P[1]}, \ldots, \texttt{P[k-1]})$$
  2. si oui : on arrête et on renvoie `i` ;
  3. si non : on réessaie avec la position `i+1` ;
- On s'arrête quand `i>n-k`, puisqu'à partir de cette valeur il n'y a plus assez de place pour contenir `P` ;

L'algorithme "naïf" en action

- Supposons que l'on doive trouver le mot "abxyabxz" dans le texte "xabxyabxyabxz" :



L'algorithme naïf de recherche d'un mot

- L'algorithme suivant renvoie la position de la première occurrence d'un mot dans un texte, ou None si le mot n'apparaît pas ;

Recherche naïve d'un mot dans un texte

```
def recherche(mot, texte):
    k = len(mot)
    n = len(texte)
    for i in range(n - k + 1):          # parcourir le texte
        # parcourir le mot
        j = 0
        while j < k and mot[j] == texte[i + j]:
            j += 1
        if j == k:                              # on a trouvé
            return i
    return None
```

Améliorations ?

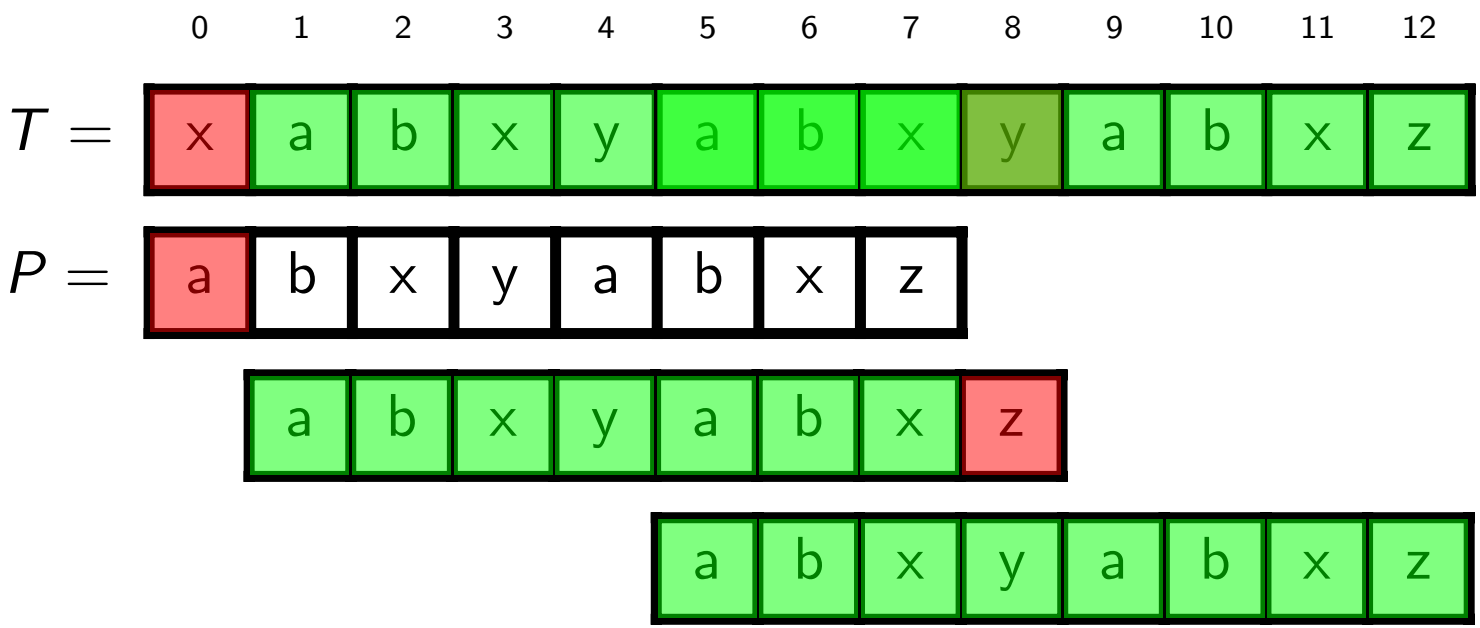
- L'algorithme naïf est en $O(k(n-k))$; peut-on faire mieux ?
- Oui !
- Une idée simple mais efficace consiste à tenter d'avancer de plus d'une position lors d'un échec ;
- On évite donc ainsi beaucoup de comparaisons inutiles ;
- Un exemple d'algorithme utilisant des décalages "améliorés" est celui de Knuth-Morris-Pratt ;

Prétraitement du motif

- Comme on ne sait rien sur le texte, on est toujours obligé d'essayer chaque position ;
- Par contre, si l'on a des informations sur la chaîne à chercher, on peut aller plus vite ;
- L'idée maîtresse est de faire des décalages plus grands ;

L'algorithme de Knuth-Morris-Pratt en action

- Supposons que l'on doive trouver le mot "abxyabxz" dans le texte "xabxyabxyabxz" :



- Il faut bien sûr savoir comment effectuer ces "décalages améliorés" ;

Chiffre de César

- Le chiffre de César consiste à décaler chaque lettre de l'alphabet de k positions ;

Exemple

On définit une correspondance entre chaque lettre et sa version chiffrée :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r

Grâce à cette correspondance, on chiffre les messages comme suit :

"message a encoder" $\longrightarrow$ "ewkksyw s wfugvwj"

(Dé)chiffrer un message

- Pour chiffrer un caractère `c` :
  - il nous faut sa position `p` dans l'alphabet initial ;
  - sa position dans l'alphabet modifié est `(p + k) % s` (`s` étant la taille de l'alphabet) ;
- Pour déchiffrer un caractère `d` :
  - il nous faut sa position `p` dans l'alphabet modifié ;
  - le caractère d'origine est en position `(p - k) % s` dans l'alphabet modifié ;
- En pratique, on peut faire mieux et plus simple avec les bonnes structures (dictionnaires, cf. cours de programmation) ;

“Casser” le chiffre de César

- Si on ne connaît pas la table M, on ne peut en principe pas déchiffrer un message chiffré ;
- En pratique, il n’est pas très difficile de reconstituer M :
  1. on sélectionne une portion T de texte chiffré suffisamment grande ;
  2. on tente de la déchiffrer en essayant tous les décalages possibles jusqu’à ce que la portion T' déchiffrée ait du sens ;
- Il faut donc essayer tous les décalages possibles ...
- ... mais comme il n’y en a que 26, c’est vite fait ;

Amélioration : chiffrement par substitution

- Le décalage d'alphabet utilisé par César est une simple **rotation** ;
- Une rotation n'est qu'une forme particulière de **permutation**, c'est-à-dire une bijection de {'a', 'b', ..., 'z'} vers lui-même ;
- Rien ne nous empêche d'utiliser n'importe quelle permutation ;

Amélioration : chiffrement par substitution

- Les algorithmes de chiffrement et de déchiffrement ne sont pas très compliqués ;
- Par contre, pour permettre au (à la) destinataire de déchiffrer le message, il faut lui donner toute la correspondance, pas juste un décalage ;
- L'attaque consistant à passer en revue toutes les possibilités est encore applicable, mais cette fois-ci, le nombre de possibilités est :

$$\begin{aligned} 26! &= 26 \times 25 \times \cdots \times 3 \times 2 \\ &= 403\,291\,461\,126\,605\,635\,584\,000\,000 \\ &= 4,032914611 \times 10^{26} \end{aligned}$$

“Casser” le chiffrement par substitution

- Pour se faire une meilleure idée de ce que représente $4,032914611 \times 10^{26}$:
  - $1\,000 = 10^3$;
  - $1\,000\,000 = 10^6$;
  - $1\,000\,000\,000 = 10^9$;
- Supposons que notre ordinateur vérifie un million de permutations à la seconde ;
  - en une minute, on en vérifie 60×10^6
  - en une heure, on en vérifie $60 \times 60 \times 10^6 = 36 \times 10^8$
  - en une journée, on en vérifie $24 \times 36 \times 10^8 = 8,64 \times 10^{10}$
  - en une semaine, on en vérifie $7 \times 8,64 \times 10^{10} = 6,048 \times 10^{11}$
  - en un mois, on en vérifie $4 \times 6,048 \times 10^{11} = 2,4192 \times 10^{12}$

  ...
- Sachant qu’en un an, on vérifie $3,1536 \times 10^{13}$ permutations, on n’aura toujours pas fini après 15 **milliards** d’années ;

“Casser” le chiffrement par substitution

- En pratique, les choses sont plus simples que ça grâce à :
  1. l’analyse linguistique et contextuelle ;
  2. l’utilisation de machines puissantes en parallèle ;
- Des techniques de chiffrement beaucoup plus sûres existent ;

Matrices

- Les **matrices** sont des tableaux à au moins deux dimensions ;
- Pour l'instant, on se contentera de travailler sur des matrices à exactement deux dimensions ;

Exemple (matrice entière à trois lignes et quatre colonnes)

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

- A quoi ça sert ?

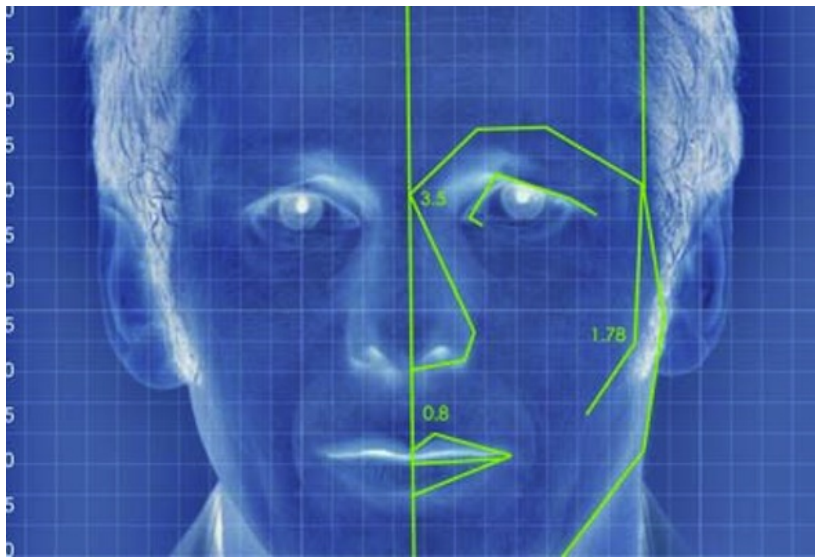
Matrices : applications

- On en utilise tous les jours pour afficher des données :

- De même :
  - les images peuvent être vues comme des matrices à deux dimensions ;
  - les animations peuvent être vues comme des matrices à trois dimensions ;
- Autre exemple : bases de données ;

Matrices : applications

- Le problème de recherche de mot dans un texte peut se généraliser aux matrices ;
- On recherche alors une matrice dans une autre matrice ;
- Application : reconnaissance faciale :



- En pratique, les choses sont beaucoup plus compliquées ;

Construction des matrices en Python

- On a vu comment construire des listes ;
- Une matrice est simplement une liste ... de listes ;

Exemple

```
>>> ligne1 = [3, 6, 7, 1]
>>> ligne2 = [2, 5, 6, 2]
>>> ligne3 = [3, 1, 4, 2]
>>> M = [ligne1, ligne2, ligne3]
>>> M
[[3, 6, 7, 1], [2, 5, 6, 2], [3, 1, 4, 2]]
>>> M[0] = [0, 0, 0, 0]
>>> M
[[0, 0, 0, 0], [2, 5, 6, 2], [3, 1, 4, 2]]
>>> ligne1
[3, 6, 7, 1]
```

Visualisation sur PythonTutor

Construction des matrices en Python

- On peut également créer une liste de listes vides, qu'on remplira plus tard ;
- Attention à ne pas se faire avoir :

Exemple

```
>>> M = [ [] ] * 10
>>> M[0].append(1)
>>> M
[[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
>>> M = [ list() ] * 10
>>> M[0].append(1)
>>> M
[[1], [1], [1], [1], [1], [1], [1], [1], [1], [1]]
```

Visualisation sur PythonTutor

Construction des matrices en Python

- Solutions :
  1. une boucle ;
  2. ou la **compréhension de listes** (prochain cours de programmation) ;

Création de matrice vide (1)

```
>>> M = []
>>> for i in range(10):
...         M.append([])
```

Création de matrice vide (2)

```
>>> M = [ list() for i\
...         in range(10) ]
```

- Dans les deux cas, on peut vérifier que tout marche “correctement” ensuite :

Exemple

```
>>> M
[[], [], [], [], [], [], [], [], [], []]
>>> M[0].append(1)
>>> M
[[1], [], [], [], [], [], [], [], [], []]  # ouf !
```

Matrices et listes de listes

- Différence de terminologie ;
- En mathématiques :
  - chaque ligne d'une matrice contient le même nombre de colonnes ;
  - on a exactement deux dimensions ;
- En programmation, il est possible :
  - d'avoir des lignes de longueurs différentes ;
  - de traiter plus de deux dimensions ;
- En algorithmique, on suppose en général que toutes les lignes sont de même longueur ;

Matrices : taille et dimensions

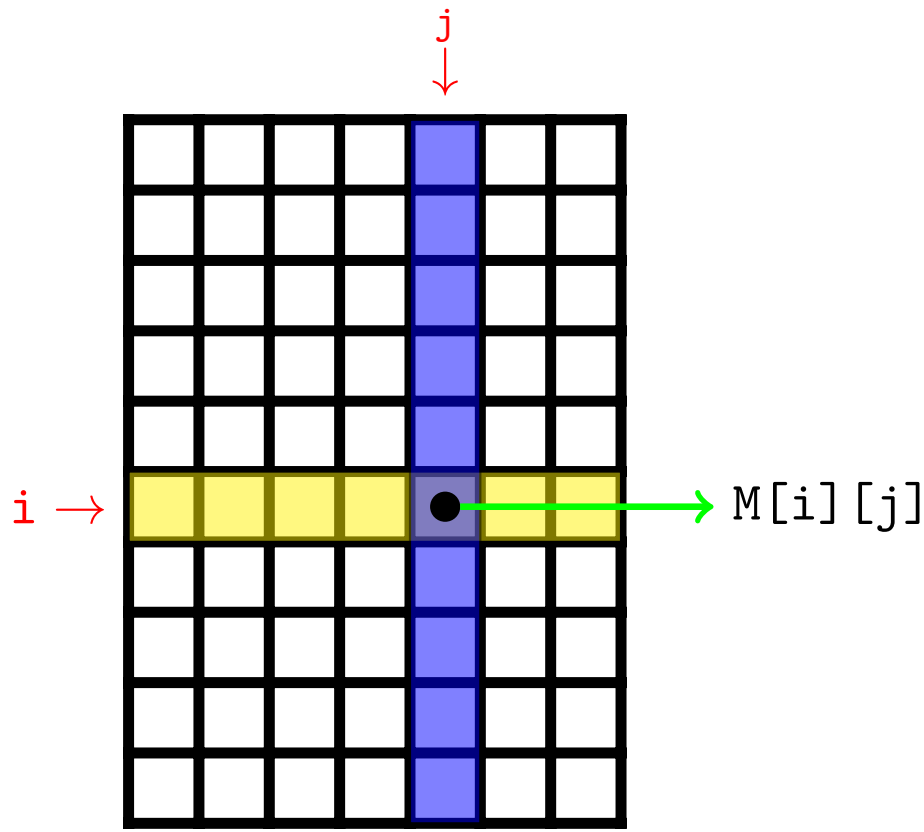
- Comme avant, on peut utiliser la fonction `len(M)` ;
- ... mais attention, cette fonction vous donne le nombre de lignes de `M` !
- Pour connaître le nombre d'éléments, il faut donc additionner la longueur de chaque ligne !

Exemple

```
>>> M
[[3, 6, 7, 1], [2, 5, 6, 2], [3, 1, 4, 2]]
>>> len(M)
3
>>> len(M[0])
4
>>> len(M[0]) + len(M[1]) + len(M[2])
12
```

Accès aux éléments

- Comme avant, on utilise l'opérateur `[]` pour accéder aux éléments d'une matrice ;
- Mais comme on a plusieurs dimensions, il faut spécifier une position par dimension :



Exemple

- Autrement dit : si `M` est une matrice, `M[i]` est une liste, pas un simple élément ;
- Pour accéder à un élément, il faut donc à la fois spécifier la ligne ET la colonne ;
- A titre d'exemple, regardons comment créer une matrice dont la ligne `i` contient les multiples de `i+1` ;

Exemple

```
M = []
for i in range(5):
    M.append(list(range(1, 11)))
    for j in range(10):
        M[i][j] *= (i + 1)
```

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50

Recherche d'un élément

- Au passage, remarquons que certaines méthodes de Python **ne marchent plus pour les matrices** ;

Exemple

```
>>> M = [[3, 6, 7, 1], [2, 5, 6, 2], [3, 1, 4, 2]]
>>> 3 in M
False
>>> M.index(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 2 is not in list
```

- Python ne "fouille" pas les sous-listes !
- Il faut donc être capable de se débrouiller sans ça ;

Recherche d'un élément

- Adapter les algorithmes vus précédemment aux matrices requiert de prendre en compte les dimensions supplémentaires ;
- A titre d'exemple, la recherche d'un élément dans une matrice à deux dimensions s'effectue comme suit :

Recherche d'un élément dans une matrice à deux dimensions

```
1  def rechercheMatrice(M, x):
2      nb_lignes, nb_colonnes = len(M), len(M[0])
3      for i in range(nb_lignes):
4          for j in range(nb_colonnes):
5              if M[i][j] == x:
6                  return i, j
7      return None
```

Rappel : produit de deux vecteurs

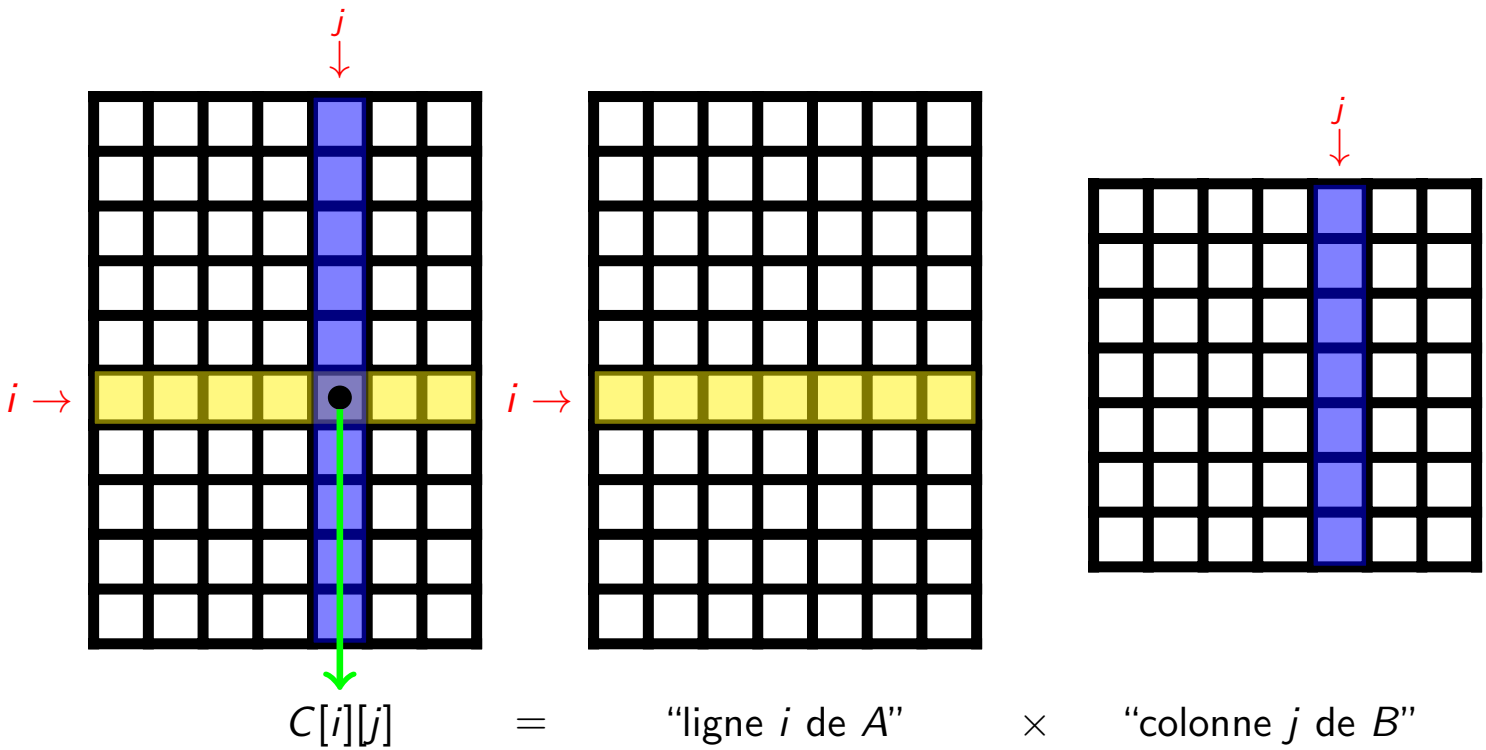
- Soit U et V deux listes de taille n ;
- Leur produit est la quantité $\sum_{i=0}^{n-1} U[i] * V[i]$;
- Il se calcule de la façon suivante :

Produit de deux vecteurs

```
def produit(U, V):
    somme = 0
    for i in range(len(U)):
        somme += U[i] * V[i]
    return somme
```

Produit matriciel

- Soit A une matrice $m \times n$ et B une matrice $n \times p$;
- Le **produit matriciel** $C = AB$ est défini comme suit :



- ... ce qui explique pourquoi le nombre de colonnes de A doit être égal au nombre de lignes de B ;

Algorithme de multiplication matricielle

- Plus formellement :
  - notons $A_{i,\rightarrow}$ la $i^{\text{ème}}$ ligne de A ;
  - notons $B_{\downarrow,j}$ la $j^{\text{ème}}$ colonne de B ;
- Alors la composante en ligne i et colonne j de AB est donnée par

$$C[i][j] = A_{i,\rightarrow} * B_{\downarrow,j}$$

- Pour calculer $A_{i,\rightarrow} * B_{\downarrow,j}$, on doit effectuer la somme des produits de leurs éléments ;
- Autrement dit :

$$A_{i,\rightarrow} * B_{\downarrow,j} = \sum_{k=0}^{n-1} A_{i,k} * B_{k,j}$$

Algorithme de multiplication matricielle

- L'algorithme peut donc s'écrire comme suit :

Produit de deux matrices

```
def produitMatriciel(A, B):
    m, n, p = len(A), len(A[0]), len(B[0])
    C = []
    for i in range(m):
        C.append([])
        for j in range(p):
            # produit de la ligne i de A
            # et de la colonne j de B
            somme = 0
            for k in range(n):
                somme += A[i][k] * B[k][j]
            C[i].append(somme)
    return C
```

Complexité du produit matriciel

- Supposons que A est une matrice $m \times n$ et B une matrice $n \times p$;
- On a trois boucles imbriquées :
  1. la première comporte m itérations ;
  2. la deuxième comporte p itérations ;
  3. la troisième comporte n itérations ;
- En conclusion, on a un algorithme en $O(m * n * p)$;
- Si les matrices sont carrées ($m = n = p$), on a donc un algorithme en $O(n^3)$;

Complexité du produit matriciel

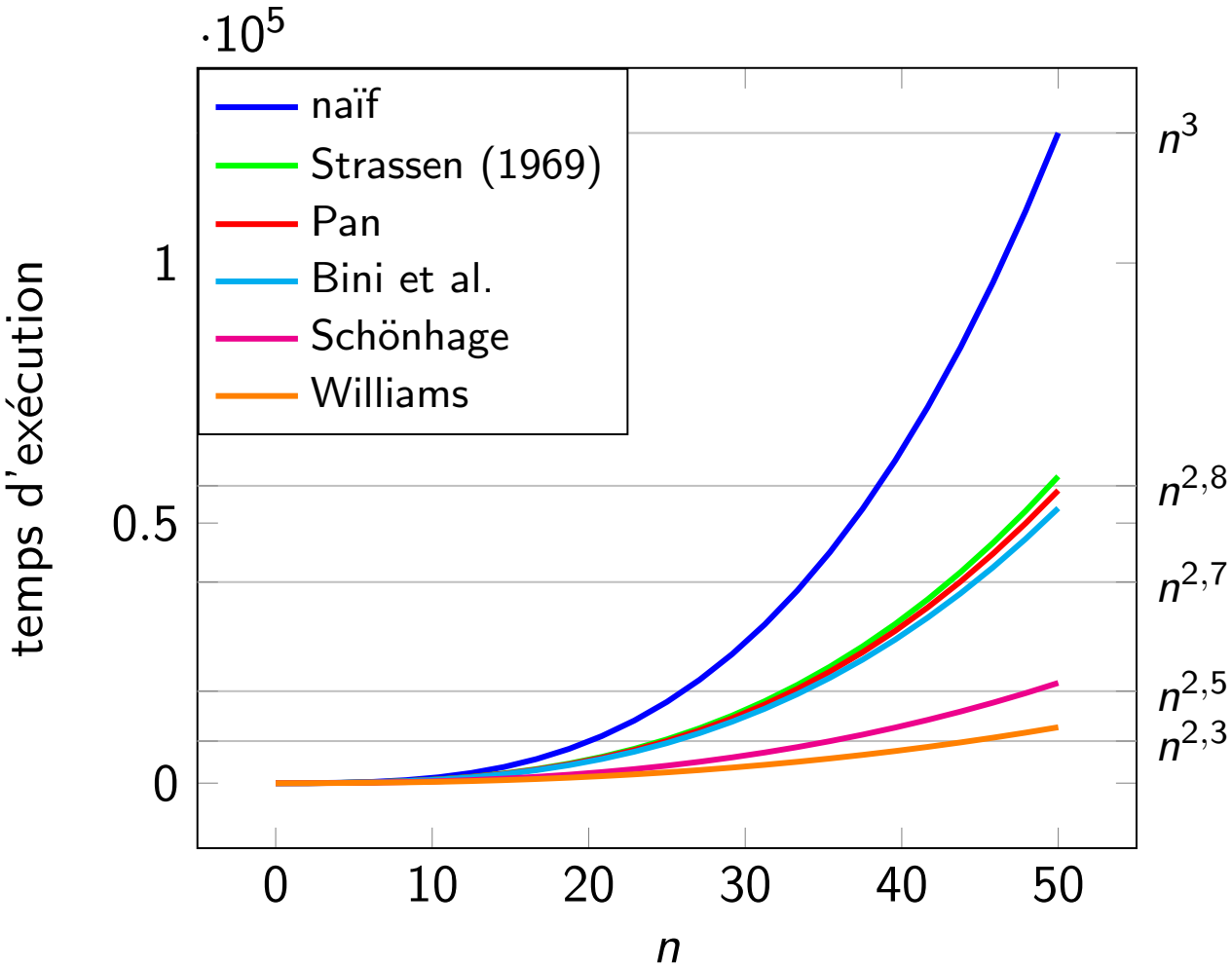
- Peut-on faire mieux que $O(n^3)$ pour deux matrices carrées ?
- Très étrangement, oui !

1969	Strassen :	$O(n^{2.808})$
1978	Pan :	$O(n^{2.796})$
1979	Bini, Capovani, Romani, Lotti :	$O(n^{2.78})$
1981	Schönhage :	$O(n^{2.522})$
1982	Romani :	$O(n^{2.517})$
1982	Coppersmith et Winograd :	$O(n^{2.496})$
1986	Strassen :	$O(n^{2.479})$
1989	Coppersmith et Winograd :	$O(n^{2.376})$
2010	Stothers :	$O(n^{2.3737})$
2012	Williams :	$O(n^{2.3727})$

- **Peut-on multiplier deux matrices $n \times n$ en temps $O(n^2)$?**

Comparaison des complexités

- Ces progrès théoriques peuvent ne pas sembler impressionnants, mais on remarque quand même la différence en pratique :



Intérêts de la multiplication matricielle

- Il s'agit d'une opération fréquente sur les matrices ;
- Beaucoup de problèmes peuvent s'y ramener :
  - factorisation LU (voir prochain cours) ;
  - inversion de matrice ;
  - transformée de Fourier et DCT (utilisées entre autres pour la compression JPEG) ;
  - plus court**s** chemin**s** dans un graphe ;

    ⋮
- Autrement dit, une solution plus efficace pour le produit matriciel donne automatiquement une solution plus efficace pour les problèmes qui lui sont équivalents (cf. cours sur la complexité) ;