Initiation à l'algorithmique

— structures linéaires —

Jacques TISSEAU

Enib–Cerv

Remarque (Notes de cours : couverture)

Ce support de cours accompagne le chapitre 4 des notes de cours « Initiation à l'algorithmique ».

— Cours d'Informatique S1 —

Initiation à l'algorithmique

Jacques TISSEAU

Ecole nationale d'ingénieurs de Brest
Centre européen de réalité virtuelle
tisseau@enib.fr

Ces notes de cours accompagnent les enseignements d'informatique du 1er semestre (S1) de l'Ecole Nationale d'Ingénieurs de Brest (Enib : www.enib.fr). Leur lecture ne dispense en aucun cas d'une présence attentive aux cours ni d'une participation active aux travaux dirigés.

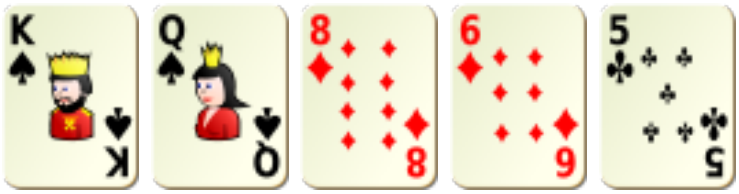
version du 21 octobre 2014

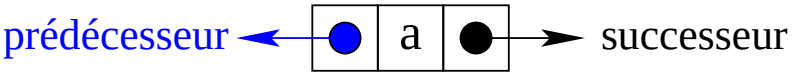
Avec la participation de Romain Bénard, Stéphane Bonneaud, Cédric Buche, Gireg Desmeulles, Céline Jost, Sébastien Kubicki, Eric Maisel, Alexis Nédélec, Marc Parenthoën et Cyril Septseault.

SÉQUENCES
DÉFINITION

Séquence : suite ordonnée d'éléments,éventuellement vide,accessibles par leur rang dans la séquence

Exemple de séquence : main au poker





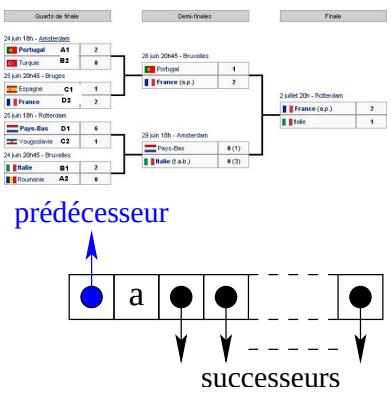
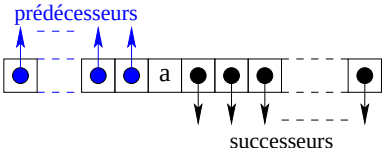
Définitions

collection regroupement fini de données dont le nombre n'est pas fixé *a priori*.

séquence suite ordonnée d'éléments, éventuellement vide, accessibles par leur rang dans la séquence.

arbre collection d'éléments, appelés « nœuds », organisés de façon hiérarchique à partir d'un nœud particulier, appelé la « racine » de l'arbre.

graphe collection d'éléments, appelés « sommets », et de relations entre ces sommets.

collection	arbre	graphe
		 

SÉQUENCES

N-UPLET, CHAÎNE, LISTE

n-uplet

```
>>> s = 1,7,2,4
>>> type(s)
<type 'tuple'>
>>> len(s)
4
>>> 3 in s
False
>>> s[1]
7
>>> s + (5,3)
(1, 7, 2, 4, 5, 3)
```

chaîne

```
>>> s = '1724'
>>> type(s)
<type 'str'>
>>> len(s)
4
>>> '3' in s
False
>>> s[1]
'7'
>>> s + '53'
'172453'
```

liste

```
>>> s = [1,7,2,4]
>>> type(s)
<type 'list'>
>>>len(s)
4
>>> 3 in s
False
>>> s[1]
7
>>> s + [5,3]
[1, 7, 2, 4, 5, 3]
```


Operation on sequences (`list`, `tuple`, `str`)	**Result**
`x in s` `x not in s`	`True` if an item of `s` is equal to `x`, else `False` `False` if an item of `s` is equal to `x`, else `True`
`s1 + s2` `s * n, n*s`	the concatenation of `s1` and `s2` `n` copies of `s` concatenated
`s[i]` `s[i:j[:step]]`	`i`'th item of `s`, origin 0 Slice of `s` from `i` (included) to `j`(excluded). Optional `step` value, possibly negative (default : 1)

`len(s)`
`min(s)`
`max(s)`
	Length of `s`
	Smallest item of `s`
	Largest item of `s`

Remarque (item assignment)

```
>>> s = 1,7,2,4
>>> type(s)
<type 'tuple'>
>>> s[1] = 3
Traceback ...
TypeError: 'tuple'
object does not
support item
assignment
>>> s[1]
7
```

```
>>> s = '1724'
>>> type(s)
<type 'str'>
>>> s[1] = '3'
Traceback ...
TypeError: 'str'
object does not
support item
assignment
>>> s[1]
7
```

```
>>> s = [1,7,2,4]
>>> type(s)
<type 'list'>
>>> s[1] = 3
>>> s
[1, 3, 2, 4]
>>> s[1]
3
```

N-UPLETS

SÉQUENCE NON MODIFIABLE D'ÉLÉMENTS

singleton	: (a)	paire	: (a,b)
triplet	: (a,b,c)	quadruplet	: (a,b,c,d)
...	:		
n-uplet	: (a,b,c,d,e,f,g,h,i,j,...)		

```
>>> s = ()
>>> type(s)
<type 'tuple'>
>>> s = 1,7,2,4
>>> type(s)
<type 'tuple'>
>>> s
(1, 7, 2, 4)
```

```
>>> s = (5)
>>> type(s)
<type 'int'>
>>> s = (5,)
>>> type(s)
<type 'tuple'>
>>> s =
(5,)+()+(6,7,9)
>>> s
(5, 6, 7, 9)
```

```
>>> s = (5,6,7,9)
>>> s[1:3]
(6, 7)
>>> s[1:]
(6, 7, 9)
>>> s[:2]
(5, 6)
>>> s[-2:]
(7, 9)
```


TD (Division entière)

Définir une fonction qui retourne le quotient q ET le reste r de la division entière $a \div b$ ($a = bq + r$).
On n'utilisera pas les opérateurs prédéfinis / et %.

CHAÎNE DE CARACTÈRES

SÉQUENCE NON MODIFIABLE DE CARACTÈRES

```
>>> s = 'une chaîne'
>>> s
'une chaîne'
>>> s = "une autre chaîne"
>>> s
'une autre chaîne'
>>> s = 'chaîne entrée sur \
... plusieurs lignes'
>>> s
'chaîne entrée sur plusieurs
lignes'
>>> s = 'chaîne entrée \n sur 1
ligne'
>>> s
chaîne entrée
 sur 1 ligne
```

```
>>> s = 'c\'est ça \"peuchère\"'
>>> s
'c\'est ça "peuchère"'
>>> print(s)
c'est ça "peuchère"
>>> s = ''' a ' \ " \n z '''
>>> s
' a \' \\ " \n z '
>>> s = 'des caractères'
>>> s[9]
't'
>>> for c in s: print(c,end=' ')
...
d e s  c a r a c t è r e s
>>> s[4:9]
'carac'
>>> s[len(s)-1]
's'
>>> s[:4] + 'mo' + s[9] + s[-1]
'des mots'
```


Operation on `str`	Result
`s.find(sub[,start[, end]])`	Returns the lowest index in `s` where substring `sub` is found. Returns `-1` if `sub` is not found
`s.partition(separ)`	Searches for the separator `separ` in `s`, and returns a tuple `(head, sep, tail)` containing the part before it, the separator itself, and the part after it. If the separator is not found, returns `s` and two empty strings
`s.replace(old,new[, maxCount=-1])`	Returns a copy of `s` with the first `maxCount` (`-1` : unlimited) occurrences of substring `old` replaced by `new`
`s.split([separator[, maxsplit]])`	Returns a list of the words in `s`, using `separator` as the delimiter string

TD (Inverser une chaîne)

1. *Définir une fonction qui crée une copie d'une chaîne en inversant l'ordre des caractères.*

```
>>> inverser('inverser')
'resrevni'
```

2. *Définir une fonction qui teste si une chaîne est un palindrome (qui se lit de la même manière de gauche à droite ou de droite à gauche : non, kayak, laval...).*

```
>>> palindrome('kayak')
True
```

LISTES

SÉQUENCE MODIFIABLE D'ÉLÉMENTS

```
>>> s = [1,3,5,7]
>>> s[2]
5
>>> s[len(s)-1]
7
>>> for c in s: print(c,end=' ')
...
1 3 5 7
>>> s[1:3]
[3,5]
>>> s[1:3] = [2,4]
>>> s
[1, 2, 4, 7]
>>> s[len(s):len(s)] = [8,9]
>>> s
[1, 2, 4, 7, 8, 9]
```

```
>>> s = [1,2,3]
>>> t = s
>>> t[0] = 9
>>> t
[9, 2, 3]
>>> s
[9, 2, 3]
>>> s = [1,2,3]
>>> t = []
>>> t[:0] = s[0:]
>>> t
[1, 2, 3]
>>> t[0] = 9
>>> t
[9, 2, 3]
>>> s
[1, 2, 3]
```


Operation on `list`	Result

`s[i] = x`
`s[i:j [:step]] = t`
`del s[i:j[:step]]`
	item `i` of `s` is replaced by `x`
	slice of `s` from `i` to `j` is replaced by `t`
	same as `s[i:j] = []`

`s.count(x)` `s.index(x[,start[,stop]])`	returns number of `i`'s for which `s[i] == x` returns smallest `i` such that `s[i] == x`. `start` and `stop` limit search to only part of the list

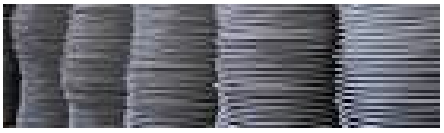
`s.append(x)`
`s.extend(x)`
`s.insert(i, x)`
 	
`s.remove(x)`
`s.pop([i])`
	same as `s[len(s) : len(s)] = [x]`
	same as `s[len(s):len(s)]= x`
	same as `s[i:i] = [x] if i>= 0. i == -1` inserts before the last element
	same as `del s[s.index(x)]`
	same as `x = s[i]; del s[i]; return x`

`s.reverse()` `s.sort([cmp ])`	reverses the items of `s` in place sorts the items of `s` in place

TD (Opérations sur les listes)

Donner un exemple d'utilisation de chacune des opérations sur les listes décrites dans le tableau ci-dessus.

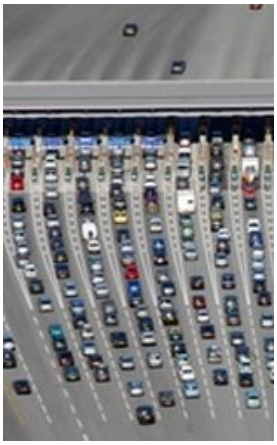
Piles



- tester si la pile est vide,
- accéder au sommet de la pile,
- empiler un élément au sommet de la pile,
- dépiler l'élément qui se trouve au sommet de la pile.

LIFO : Last In, First Out

Files



FIFO : First In, First Out

TD (Opérations sur les piles)

Définir les 4 opérations sur les piles : `emptyStack`, `topStack`, `pushStack` *et* `popStack`.
On empilera et dépilera à la fin de la liste qui sert à stocker les éléments de la pile.

TD (Opérations sur les files)

Définir les 4 opérations sur les files : `emptyQueue`, `topQueue`, `pushQueue` *et* `popQueue`.
On enfilera en début de liste et on défilera à la fin de la liste qui sert à stocker les éléments de la file.

liste multidimensionnelle liste dont les éléments sont des listes.

```
>>> s = [[4,5],[1,2,3],[6,7,8,9]]
>>> type(s)
<type 'list'>
>>> len(s)
3
>>> type(s[2])
<type 'list'>
>>> s[2]
[6, 7, 8, 9]
>>> s[2][1]
7
>>> s[1][2]
3
```

```
>>> s = [[4,5],[1,2,3],[6,7,8,9]]
>>> for c in s: print(c)
...
[4, 5]
[1, 2, 3]
[6, 7, 8, 9]
>>> s[2]
[6, 7, 8, 9]
>>> for c in s:
...     for e in c: print(e,end=' ')
...     print()
...
4 5
1 2 3
6 7 8 9
```


TD (listes multidimensionnelles)

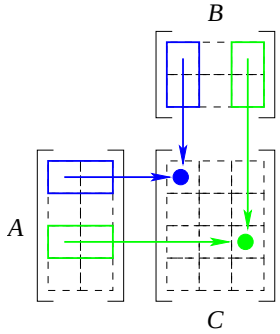
Afficher un par un les éléments de la liste suivante :
`s = [[[4,5],[1,2,3]], [[0],[10,11],[6,7,8,9]]]`.

TD (Addition de matrices)

Définir la fonction qui calcule la matrice C, addition de 2 matrices A et B de mêmes dimensions (n, m) telle que $c_{ij} = a_{ij} \cdot b_{ij}$.

TD (Produit de matrices)

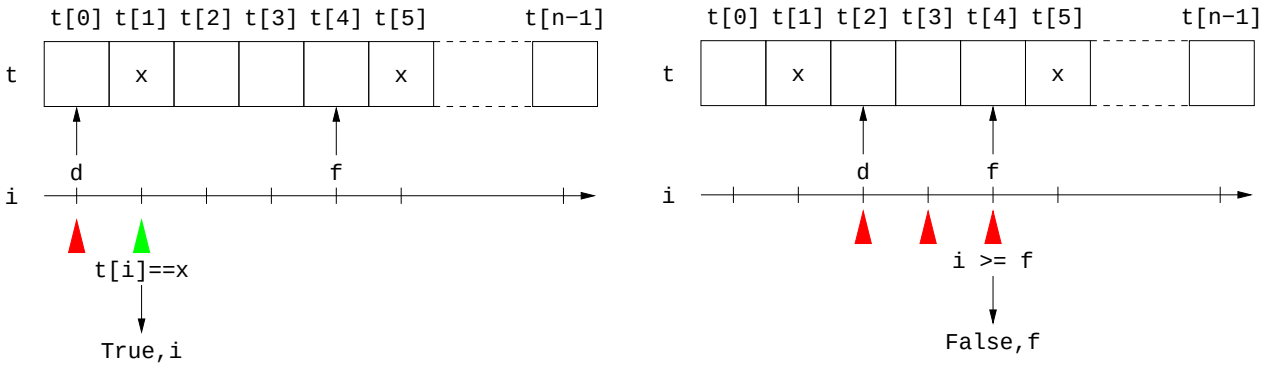
Définir la fonction qui calcule la matrice C, produit de 2 matrices A et B respectivement de dimensions (n, r) et (r, m).



$$c_{ij} = \sum_{k=0}^{r-1} a_{ik} \cdot b_{kj}$$

recherche retrouver une information stockée en mémoire vive, sur un disque dur ou sur le réseau.

Recherche dans une séquence



Complexité linéaire : $O(n)$

TD (Recherche séquentielle)

Définir une fonction de recherche séquentielle de la première occurence d'un élément `x` *dans une liste* `t` *entre les rangs* `debut` *(inclus) et* `fin` *(exclu).*
On retournera un couple de valeurs `(ok,r)` *:* `ok` *est un booléen qui teste si la recherche est fructueuse ou non et* `r` *est le rang de la première occurence de l'élément recherché s'il a effectivement été trouvé.*

```
>>> s = [1,3,5,6,5,2]
>>> recherche(s,5,0,len(s)-1)
(True, 2)
>>> recherche(s,5,3,len(s)-1)
(True, 4)
>>> recherche(s,4,0,len(s)-1)
(False, 6)
```

TD (Annuaire téléphonique)

On considère un annuaire téléphonique stocké sous la forme d'une liste de couples `(nom,téléphone)`.
Exemple : `[('jean','0607080910'),('paul','0298000102')]`

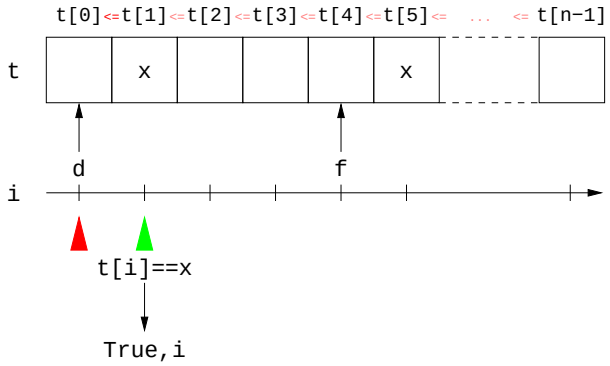
1. *Définir une fonction qui retrouve dans un annuaire téléphonique le numéro de téléphone à partir du nom.*
2. *Définir la fonction inverse qui retrouve le nom à partir du numéro de téléphone.*

RECHERCHE D'UN ÉLÉMENT

RECHERCHE DICHOTOMIQUE

Recherche dans une séquence triée

`m = (d+f)/2` : milieu de la plage de recherche



- si `x == t[m]`, on a trouvé une solution et la recherche s'arrête ;
- si `x < t[m]`, poursuivre la recherche dans la moitié gauche de la liste ;
- si `x > t[m]`, poursuivre la recherche dans la moitié droite de la liste.

Complexité logarithmique : $O(\log(n))$

TD (Recherche dichotomique)

1. *Définir une fonction de recherche dichotomique d'un élément* `x` *dans une liste triée* `t` *entre les rangs* `debut` *(inclus) et* `fin` *(exclu).*
   On retournera un couple de valeurs `(ok,r)` *:* `ok` *est un booléen qui teste si la recherche est fructueuse ou non et* `r` *est le rang de la première occurence de l'élément recherché s'il a effectivement été trouvé.*

```
>>> s = [1,3,5,6,6,9]
>>> dichotomie(s,6,0,len(s)-1)
(True, 4)
>>> dichotomie(s,6,2,len(s)-1)
(True, 3)
>>> dichotomie(s,4,0,len(s)-1)
(False, 1)
```

2. *A priori, la recherche dichotomique obtenue n'assure pas de trouver la première occurence d'un élément* `x` *dans une liste* `t` *triée.*
   Modifier l'algorithme de recherche dichotomique proposé pour rechercher la première occurence d'un élément `x` *dans une liste* `t` *triée.*

TRI D'UNE SÉQUENCE

TRI PAR SÉLECTION

relation d'ordre total (notée $\leq$)

1. réflexivité : $x \leq x$
2. antisymétrie : $(x \leq y)$ and $(y \leq x) \Rightarrow x = y$
3. transitivité : $(x \leq y)$ and $(y \leq z) \Rightarrow (x \leq z)$

6	4	1	3	5	2
1	4	**6**	3	5	2
1	**2**	6	3	5	**4**
1	2	**3**	**6**	5	4
1	2	3	**4**	5	**6**
1	2	3	4	**5**	6
1	2	3	4	5	**6**

Complexité quadratique : $O(n^2)$

TD (Liste ordonnée)

Définir une fonction qui teste si une liste est triée ou non entre les rangs `debut` *(inclus) et* `fin` *(exclu).*

```
>>> s = [3,1,2,3,1]
>>> enOrdre(s,0,len(s)-1)
False
>>> enOrdre(s,1,3)
True
>>> enOrdre(s,1,4)
False
```

TD (Tri par sélection)

Définir une fonction qui trie une liste `t` *par la méthode du tri par sélection entre les rangs* `debut` *(inclus) et* `fin` *(exclu).*

```
>>> s = [5,4,3,2,1,0]
>>> triSelection(s,0,len(s)-1)
>>> s
[0, 1, 2, 3, 4, 5]
```

TD (Tri d'un annuaire téléphonique)

On considère un annuaire téléphonique stocké sous la forme d'une liste de couples `(nom,téléphone)`.
Exemple : `[('paul','0607080910'),('jean','0298000102')]`
Définir une fonction qui trie un annuaire téléphonique par ordre alphabétique des noms.

TRI D'UNE SÉQUENCE

TRI PAR INSERTION

tri par insertion : trier successivement les premiers éléments de la liste
A la $i^{ème}$ étape, on insère le $i^{ème}$ élément à son rang parmi les $i-1$ éléments précédents qui sont déjà triés entre eux.

6	4	1	3	5	2
4	6	1	3	5	2
1	4	6	3	5	2
1	**3**	4	6	5	2
1	3	4	**5**	6	2
1	**2**	3	4	5	6

Complexité quadratique : $O(n^2)$

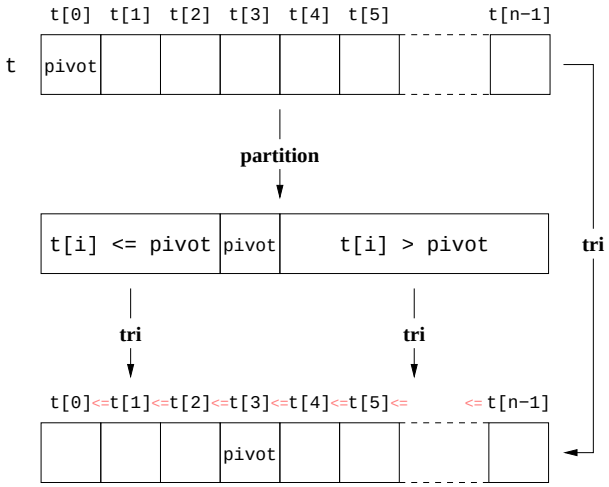
TD (Tri par insertion)

Définir une fonction qui trie une liste `t` *par la méthode du tri par insertion entre les rangs* `debut` *(inclus) et* `fin` *(exclu).*

```
>>> s = [9,8,7,6,5,4]
>>> triInsertion(s,0,len(s)-1)
>>> s
[4, 5, 6, 7, 8, 9]
>>> s = [9,8,7,6,5,4]
>>> triInsertion(s,1,4)
>>> s
[9, 5, 6, 7, 8, 4]
```

TRI D'UNE SÉQUENCE

TRI RAPIDE



6	4	1	3	5	2
2	4	1	3	5	**6**
1	**2**	4	3	5	6
1	2	3	**4**	5	6
1	2	3	4	5	6

Complexité quasi-linéaire : $O(n \log(n))$

TD (Partition d'une liste)

Définir une fonction qui partitionne une liste en deux telle que tous les éléments à gauche du pivot soient inférieurs à tous les éléments à droite du pivot.
On retournera le rang du pivot.

```
>>> s = [3,5,2,6,1,4]
>>> partition(s,0,len(s)-1,s[0]), s
(2, [1, 2, 3, 6, 5, 4])
```

TD (Tri rapide)

Définir une version récursive du tri rapide d'une liste `t` *entre les rangs* `debut` *(inclus) et* `fin` *(exclu).*

```
>>> s = [9,8,7,6,5,4]
>>> triRapide(s,0,len(s)-1)
>>> s
[4, 5, 6, 7, 8, 9]
>>> s = [9,8,7,6,5,4]
>>> triRapide(s,1,4)
>>> s
[9, 5, 6, 7, 8, 4]
```