Initiation à l'algorithmique

— instructions de base —

Jacques TISSEAU

Enib–Cerv

Remarque (Notes de cours : couverture)

Ce support de cours accompagne le chapitre 2 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.

Instruction

Commande élémentaire interprétée et exécutée par le processeur.

Jeu d'instructions

Dans un processeur, ensemble des instructions que cette puce peut exécuter.

Bloc d'instructions

Dans un algorithme, séquence d'instructions pouvant être vue comme une seule instruction.

Remarque (Cycles per Instruction)

Le cœur du microprocesseur est régulé par un quartz qui oscille avec une fréquence exprimée en Hz. Le temps de cycle est l'inverse de la fréquence. Ainsi pour une fréquence de 100 MHz, on a un temps de cycle de 10 ns. L'exécution d'une instruction nécessite plusieurs temps de cycle, c'est ce que l'on appelle le cpi (Cycles per Instruction).

Classes d'instructions μP

arithmétique : `+`, `-`, `*`, `/`

logique : `not`, `and`, `or`

transferts de données : `load`, `store`, `move`

contrôle du flux d'instructions : branchements, boucles, appels de procédure

entrée-sortie : `read`, `write`

Traitement des instructions

1. `fetch` : chargement de l'instruction,
2. `decode` : décodage,
3. `load operand` : chargement des données,
4. `execute` : exécution,
5. `result write back` : mise à jour.

Remarque (Types d'architecture de micro-processeur)

- *les architectures* risc (Reduced Instruction Set Computer) *préconisent un petit nombre d'instructions élémentaires dans un format fixe ;*
- *les architectures* cisc (Complex Instruction Set Computer) *sont basées sur des jeux d'instructions très riches de taille variable offrant des instructions composées de plus haut niveau d'abstraction.*

Chaque architecture possède ses avantages et ses inconvénients : pour le risc *la complexité est reportée au niveau du compilateur, pour le* cisc *le décodage est plus pénalisant. En fait les machines* cisc *se sont orientées vers une architecture* risc *où les instructions* cisc *sont traduites en instructions* risc *traitées par le coeur du processeur.*

INSTRUCTIONS DE BASE

COMMENTAIRE, NE RIEN FAIRE, BLOC

Instructions

commentaire : aide pour l'utilisateur humain.

```
# fin de ligne ignorée ↩
```

instruction vide : ne rien faire.

```
pass
```

bloc d'instructions : regrouper plusieurs instructions en une seule.

```
instruction1
    instruction2.1
    ...
    instruction2.n
instruction3
```

noter l'indentation du bloc d'instructions 2

Instructions

affectation : changer la valeur d'une variable.

```
variable = expression
```

conditions : exécuter une instruction sous condition.

```
if condition: bloc
[elif condition: bloc]*
[else: bloc]
```

itérations : répéter plusieurs fois la même instruction.

```
while condition: bloc
```

```
for element in sequence: bloc
```

Définition

Une variable est un objet informatique qui associe un nom à une valeur qui peut éventuellement varier au cours du temps
(une variable dénote une valeur).

Nom d'une variable

Le nom d'une variable est un identificateur aussi explicite que possible (exprimer le contenu sémantique de la variable).
Exemples :

:-(	:-)
x	pression
y	angleRotation
z	altitude

:-(	:-)
t	temps
u	masse
v	vitesse

Remarque (Mots réservés en Python)

and	*del*	*for*	*is*	*raise*
assert	*elif*	*from*	*lambda*	*return*
break	*else*	*global*	*not*	*try*
class	*except*	*if*	*or*	*while*
continue	*exec*	*import*	*pass*	*with*
def	*finally*	*in*	*print*	*yield*

Règles lexicales

- Un nom de variable est une séquence de lettres (a...z , A...Z) et de chiffres (0...9), qui doit toujours commencer par une lettre.
  `a2pique`, `jeanMartin`, `ieee754`
- Pas de lettres accentuées, de cédilles, d'espaces, de caractères spéciaux tels que $, #, @, etc., à l'exception du caractère _ (souligné).
  `vitesse_angulaire`, `element`, `ca_marche`
- La casse est significative : les caractères majuscules et minuscules sont distingués.
  `python` ≠ `Python` ≠ `PYTHON`

Conventions lexicales

- *a priori*, n'utiliser que des lettres minuscules

:-(	:-)
`Variable`	`variable`

- n'utiliser les majuscules qu'à l'intérieur du nom pour augmenter la lisibilité

:-(	:-)
`programmepython`	`programmePython`

- nom de constante tout en majuscule

:-(	:-)
`rouge`	`ROUGE`

AFFECTATION

DÉFINITION

Définition

Opération qui attribue une valeur à une variable.

```
... = ...
```

Valeur d'une constante

```
variable = constante
```

Valeur d'une expression

```
variable = expression
```

AFFECTATION

`variable = constante`

Valeur d'une constante

```
variable = constante
```

Exemple : initialisations

```
booleen = False
entier = 3
reel = 0.0
chaine = "salut"
autreChaine = 'bonjour, comment ça va?'
tableau = [5,2,9,3]
matrice = [[1,2],[6,7],[9,1]]
```


Remarque (Types de base en Python)

type	*nom*	*exemples*
booléens	`bool`	`False, True`
entiers	`int`	`3, -7`
réels	`float`	`3.14, 7.43e-3`
chaînes	`str`	`'salut', "l'eau"`
n-uplets	`tuple`	`1,2,3`
listes	`list`	`[1,2,3]`
dictionnaires	`dict`	`{'a':4, 'r':8}`

AFFECTATION

`variable = expression`

Valeur d'une expression

```
variable = expression
```

On évalue d'abord l'expression puis on affecte sa valeur à la variable.

Exemple : calculs

```
somme = n*(n+1)/2
delta = b*b - 4*a*c
```

Exemple : échange de valeurs entre 2 variables

```
tmp = x
x = y
y = tmp
```


Remarque (Principales affectations en Python)

`a = b`		
`a += b`	≡	`a = a + b`
`a -= b`	≡	`a = a - b`
`a *= b`	≡	`a = a * b`
`a /= b`	≡	`a = a / b`
`a %= b`	≡	`a = a % b`
`a **= b`	≡	`a = a ** b`

AFFECTATION

`variable = expression`

Exemple : modification

```
i = i + 1      # incrémentation
i = i - 1      # décrémentation
q = q/b
```

Attention !

L'affectation est une opération typiquement informatique qui se distingue de l'égalité mathématique.
En mathématique une expression du type `i = i+1` se réduit en `0 = 1` !
En informatique, l'expression `i = i+1` conduit à ajouter `1` à la valeur de `i` (évaluation de l'expression `i+1`), puis à donner cette nouvelle valeur à `i` (affectation).

TD (Permutation circulaire)

Effectuer une permutation circulaire droite entre les valeurs de 4 entiers x, y, z et t.

TD (Séquences d'affectations)

Quelles sont les valeurs des variables a, b, q et r après les séquences d'affectations suivantes ?

1.
```
a = 19
b = 6
q = 0
r = a
r = r - b
q = q + 1
r = r - b
q = q + 1
r = r - b
q = q + 1
```

2.
```
a = 12
b = 18
r = a%b
a = b
b = r
r = a%b
a = b
b = r
r = a%b
a = b
b = r
```

Définition

Exécuter une instruction sous condition.

```
if condition: bloc
[elif condition: bloc]*
[else: bloc]
```

Les instructions entre crochets ([...]) sont optionnelles.
[...]* signifie que les instructions entre crochets peuvent être répétées 0 ou plusieurs fois.

Structure de contrôle effectuant un test et permettant un choix entre diverses parties du programme. On sort ainsi de l'exécution purement séquentielle des instructions.

Remarque (Instructions conditionnelles)

test simple	`if condition : blocIf`
alternative simple	`if condition : blocIf` `else: blocElse`

alternative multiple	`if condition : blocIf`
	`elif condition1: blocElif1`
	`elif condition2: blocElif2`
	`...`
	`else: blocElse`

TD (Opérateurs booléens dérivés)

En utilisant les opérateurs booléens de base (`not`, `and` *et* `or`*), ecrire un algorithme qui affecte successivement à une variable* `s` *le résultat des opérations booléennes suivantes : ou exclusif (*xor, $a \oplus b$*), non ou (*nor, $\overline{a + b}$*), non et (*nand, $\overline{a \cdot b}$*), implication ($a \Rightarrow b$) et équivalence ($a \Leftrightarrow b$).*

ALTERNATIVES

TEST SIMPLE

```
if condition: bloc
```

Condition : comparaison

```
x == y
x != y
x <  y
x <= y
x >  y
x >= y
```

```
if x < 0: y = -x
if x != y: y = x
```

Condition : calcul booléen

```
    not a
a   and b
a   or  b
```

```
if (x > 0) and (x < 2):
    y = 3*x
if (x <= 0) or (x >= 2):
    y = 4*x
```

```
if condition: bloc
else: bloc
```

Exemple : valeur absolue

```
if x < 0:
    valeurAbsolue = -x
else:
    valeurAbsolue = x
```

Exemple : maximum

```
if x > y:
    maximum = x
else:
    maximum = y
```


Définitions

test simple instruction de contrôle du flux d'instructions qui permet d'exécuter une instruction sous condition préalable.

alternative simple instruction de contrôle du flux d'instructions qui permet de choisir entre deux instructions selon qu'une condition est vérifiée ou non.

TD (Alternative simple et test simple)

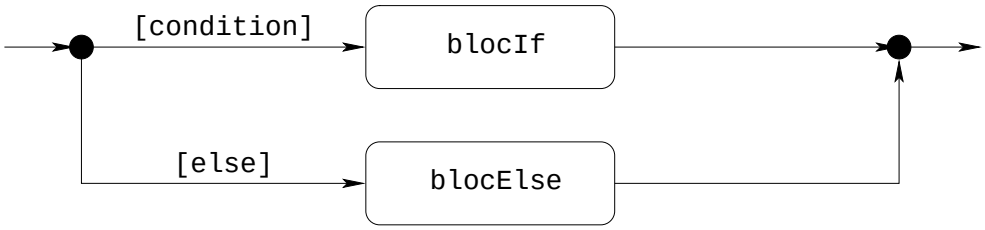
Montrer à l'aide d'un contre-exemple que l'alternative simple :

```
if condition : blocIf
else : blocElse
```

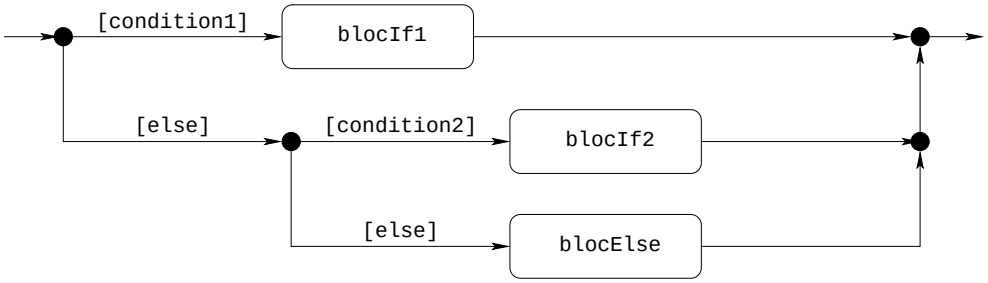
n'est pas équivalente à la séquence de tests simples suivante :

```
if condition : blocIf
if not condition : blocElse
```

Alternative simple



Alternatives simples en cascade



TD (Alternative simple)

Quelle est la valeur de la variable y après la suite d'instructions suivante ?

```
p = 1
d = 0
r = 0
h = 1
z = 0
f = p and (d or r)
g = not r
m = not p and not z
g = g and (d or h or m)
if f or g : y = 1
else : y = 0
```

TD (Alternatives simples en cascade)

Quelle est la valeur de la variable ok après la suite d'instructions suivante ?

```
x = 2
y = 3
d = 5
h = 4
if x > 0 and x < d :
  if y > 0 and y < h : ok = 1
  else : ok = 0
else : ok = 0
```

```
if condition: bloc
elif condition: bloc
...
else: bloc
```

Exemple : mentions du bac

```
if note < 10: mention = "ajourné"
elif note < 12: mention = "passable"
elif note < 14: mention = "assez bien"
elif note < 16: mention = "bien"
else: mention = "très bien"
```

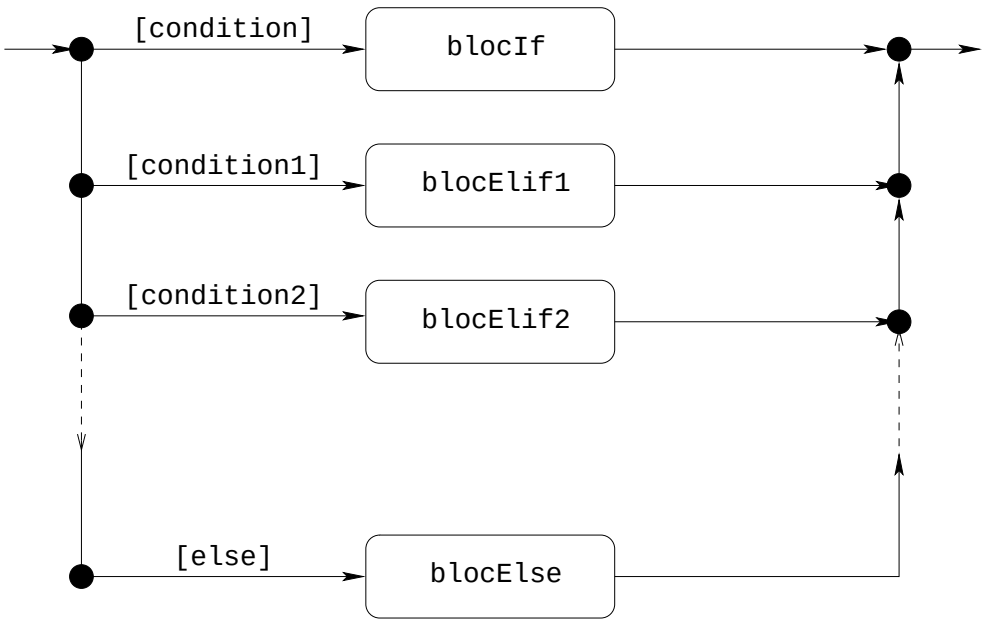

Définitions

alternative multiple instruction de contrôle du flux d'instructions qui permet de choisir entre plusieurs instructions en cascadant des alternatives simples.

TD (Alternatives multiples)

Quelle est la valeur de la variable y après la suite d'instructions suivante ?

```
x = 3
y = -2
if x < y : y = y - x
elif x == y : y = 0
else : y = x - y
```



TD (Prix d'une photocopie)

Ecrire un algorithme qui affiche le prix de n photocopies sachant que le reprographe facture 0,10 E les dix premières photocopies, 0,09 E les vingt suivantes et 0,08 E au-delà.

TD (Calcul des impôts)

Ecrire un algorithme qui affiche si un contribuable d'un pays imaginaire est imposable ou non sachant que :

- *les hommes de plus de 18 ans paient l'impôt,*
- *les femmes paient l'impôt si elles ont entre 18 et 35 ans,*
- *les autres ne paient pas d'impôt.*

Définition

Répétition d'un bloc d'instructions 0 ou plusieurs fois.

```
while condition: bloc
```

```
for element in sequence: bloc
```

Structures de contrôle destinées à être exécutées plusieurs fois (la structure de contrôle relançant l'exécution du bloc tant qu'une condition est remplie).

Remarque (Instructions itératives)

itération conditionnelle	`while condition : blocWhile`
parcours de séquence	`for element in sequence : blocFor`

Boucle `while`

```
while condition: bloc
```

- Le bloc d'instructions d'une boucle `while` peut ne jamais être exécuté (condition non vérifiée la première fois).
  Exemple : `i = 0`
  `while i > 0: bloc`
- On peut ne jamais sortir d'une boucle `while` (condition toujours vérifiée).
  Exemple : `while True: bloc`

Définitions

itération conditionnelle instruction de contrôle du flux d'instructions qui permet sous condition préalable de répéter zéro ou plusieurs fois la même instruction.

TD (Dessin d'étoiles)

Ecrire un algorithme itératif qui affiche les n lignes suivantes (l'exemple est donné ici pour n = 6) :

```
******
*****
****
***
**
*
```

Rappel Python :

```
>> 5*'r'
'rrrrr'
>> 2*'to'
'toto'
```

ITÉRATION CONDITIONNELLE

ELÉVATION À LA PUISSANCE

$$p = x^n$$

```
x = 2
n = 3
i = 0
p = 1
print(x, n, p, i)
while i < n:
  p = p * x
  i = i + 1
  print(x, n, p, i)
print(x, n, p, i)
```

x	n	p	i
2	3	1	0
2	3	2	1
2	3	4	2
2	3	8	3
2	3	8	3

$$p = 8 = 2^3 = x^n$$

TD (Fonction factorielle)

Ecrire un algorithme qui calcule $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$.

ITÉRATION CONDITIONNELLE

DIVISION ENTIÈRE

$$a = bq + r$$

```
a = 8
b = 3
q = 0
r = a
print(a, b, r, q)
while r >= b:
  r = r - b
  q = q + 1
  print(a, b, r, q)
print(a, b, r, q)
```

a	b	r	q
8	3	8	0
8	3	5	1
8	3	2	2
8	3	2	2

$$a = bq + r = 3 \cdot 2 + 2 = 8$$

TD (Pgcd de 2 entiers)

Ecrire un algorithme qui calcule le plus grand commun diviseur de 2 entiers a et b sachant que

$$\text{pgcd}(a, b) = \text{pgcd}(b, a\%b) = \cdots = \text{pgcd}(d, 0) = d$$

ITÉRATION CONDITIONNELLE

RACINE CARRÉE ENTIÈRE

$$r = \sqrt{n}$$

```
n = 17
r = 0
print(n, r)
while (r+1)**2 <= n:
  r = r + 1
  print(n, r)
print(n, r)
```

n	r
17	0
17	1
17	2
17	3
17	4
17	4

$$r^2 = 4^2 = 16 \leq 17 = n$$

$$n = 17 < (r+1)^2 = 5^2 = 25$$

$$r^2 \leq n < (r+1)^2$$

TD (Itérations conditionnelles)

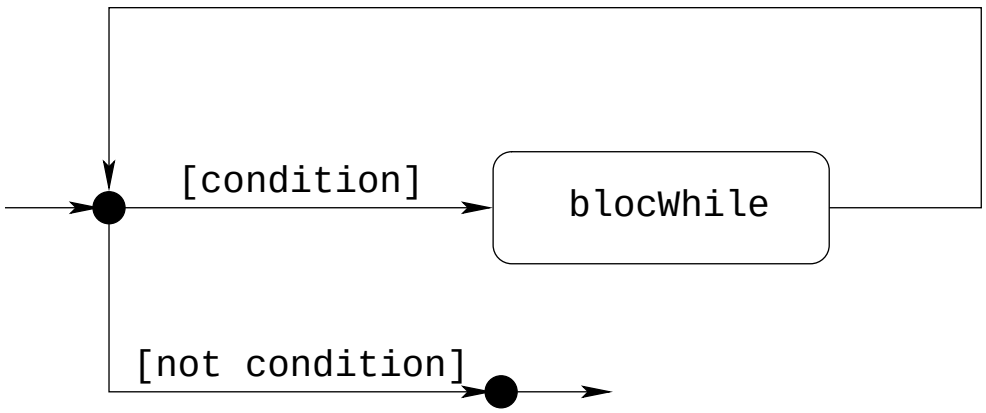
1. *Que fait cette suite d'instructions ?*

```
x = 0
while x <= 0 or x > 5 :
  x = input('entrer un nombre : ')
```

2. *Quelle est la valeur de la variable s à la fin des instructions suivantes ?*

```
b = 2
k = 8
n = 23
s = 0
i = k - 1
q = n
while q != 0 and i >= 0 :
  s = s + (q%b)*b**(k-1-i)
  print(q%b,end=' ')
  q = q/b
  i = i - 1
```

Boucle `while`



TD (Fonction exponentielle)

Ecrire un algorithme qui calcule $\exp(x)$ *en fonction de son développement en série entière.*

$$y = \exp(x) \approx \sum_{k=0}^{n} u_k = \sum_{k=0}^{n} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \ldots + \frac{x^n}{n!}$$

Les calculs seront arrêtés lorsque la valeur absolue du terme u_k *sera inférieure à un certain seuil* s $(0 < s < 1)$*. On n'utilisera ni la fonction* puissance (x^n) *ni la fonction* factorielle $(n!)$ *pour effectuer le calcul de* $\exp(x)$*.*

Boucle `for`

```
for element in sequence: bloc
```

La séquence peut être

- une séquence explicite
  Exemples : `[5,6,7]`,
- une séquence calculée (`range(min,max,pas)`)
  Exemples : `range(0,5,2)` → `[0,2,4]`
  `range(0,3,1)` → `[0,1,2]`
  `range(0,3)` → `[0,1,2]`
  `range(3)` → `[0,1,2]`

Définitions

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

Remarque (Principales opérations sur les séquences en Python)

Operation	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]`
`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

PARCOURS DE SÉQUENCE

AFFICHAGE ÉLÉMENT PAR ÉLÉMENT

```
s = [6,7,8,9,10]
print(s)
for e in s:
  print(e)
print(s)
```

s	e
$[6, 7, 8, 9, 10]$	
	6
	7
	8
	9
	10
$[6, 7, 8, 9, 10]$	

TD (Affichage inverse)

Ecrire un algorithme qui affiche les caractères d'une séquence `s`*, un par ligne en partant de la fin de la séquence.*

PARCOURS DE SÉQUENCE

SOMME ARITHMÉTIQUE

$$s = \sum_{i=1}^{i=n} i$$

```
n = 4
s = 0
print(n, i, s)
for i in range(1,n+1):
  s = s + i
  print(n, i, s)
print(n, i, s)
```

n	i	s
4	?	0
4	1	1
4	2	3
4	3	6
4	4	10
4	4	10

$$s = 1 + 2 + 3 + 4 = 10 = \sum_{i=1}^{i=4} i$$

PARCOURS DE SÉQUENCE

FACTORIELLE

$$f = n! = \prod_{i=1}^{i=n} i$$

```
n = 4
f = 1
print(n, i, f)
for i in range(1,n+1):
  f = f * i
  print(n, i, f)
print(n, i, f)
```

n	i	f
4	?	1
4	1	1
4	2	2
4	3	6
4	4	24
4	4	24

$$s = 1 \cdot 2 \cdot 3 \cdot 4 = 24 = \prod_{i=1}^{i=4} i$$

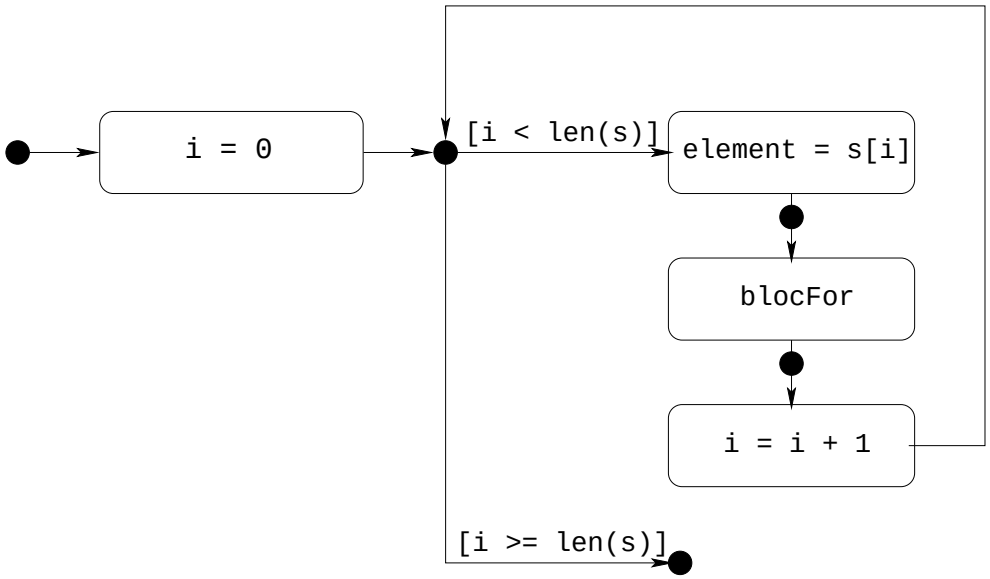
TD (Itérations imbriquées)

Qu'affichent les itérations suivantes ?

1.
```
for i in range(1,10):
  for j in range(0,11) :
    print(i, 'x', j, ' = ', i*j)
  print()
```

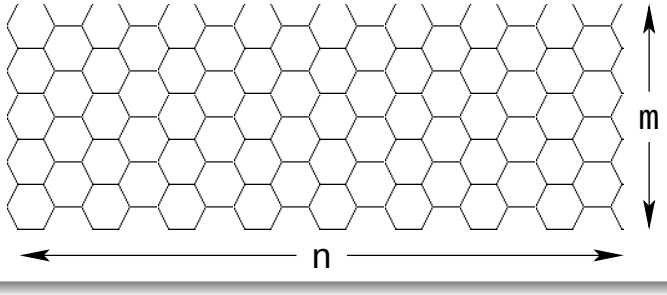
2.
```
for n in range(10) :
  for p in range(n+1) :
    num = 1
    den = 1
    for i in range(1,p+1) :
      num = num*(n-i+1)
      den = den*i
    c = num/den
    print(c,end=' ')
  print()
```

Boucle `for`



TD (Nid d'abeilles)

Ecrire un algorithmequi dessine un nid d'abeilles formé de $n \times m$ hexagones en quinconce comme sur la figure ci-dessous.



```
for i in range(min,max,pas):
  bloc
```

```
# élévation à la puissance

p = 1
for i in range(n):
  p = p * x
```

```
i = min
while i < max:
  bloc
  i = i + pas
```

```
p = 1
i = 0
while i < n:
  p = p * x
  i = i + 1
```


TD (Boucles for et while imbriquées)

Qu'affichent les itérations suivantes ?

1. ```
   for i in range(0,10) :
     j = 10 - i
     while j > 0 :
       print('*',end=' ')
       j = j - 1
     print()
   ```
2. ```
   n = 0
   while n < 10 :
     for p in range(n+1) :
       num = 1
       den = 1
       for i in range(1,p+1) :
         num = num*(n-i+1)
         den = den*i
       c = num/den
       print(c,end=' ')
     print()
     n = n + 1
   ```

```
for element in sequence:
  bloc
```

```
# affichage élément
# par élément

s = [6,7,8,9,10]
print(s)
for e in s:
  print(e)
print(s)
```

```
i = 0
while i < len(sequence):
  element = sequence[i]
  bloc
  i = i + 1
```

```
s = [6,7,8,9,10]
print(s)
i = 0
while i < len(s):
  e = s[i]
  print(e)
  i = i + 1
print(s)
```