PROGRAMMES ET ALGORITHMES
Partie I

B. Landelle

Table des matières

I Introduction

1 Définitions

Définition 1. *Un* algorithme *est une suite finie d'instructions permettant la résolution d'un problème.*

Le mot *algorithme* vient du nom du mathématicien persan Al Khawarizmi (783-850), précurseur de l'algèbre et diffuseur des chiffres arabes, ceux utilisés aujourd'hui.

Les premiers algorithmes référencés connus remontent à l'Antiquité : l'algorithme d'Euclide pour la détermination du pgcd, le crible d'Eratosthène pour la recherche de nombres premiers, résolution d'équations du second degré etc. ...

Nous sommes familiers, parfois sans le savoir, avec des algorithmes : recettes de cuisine, notice de montage, multiplication de deux nombres, recherche dans un dictionnaire, etc. ...

Exemple : Résolution dans $\mathbb{R}$ de l'équation $ax^2+bx+c=0$ avec a,b,c réels et a non nul.

Algorithme 1 : Équation de degré 2

Entrées : a, b, c
$\Delta \leftarrow b^2-4ac$;
si $\Delta<0$ **alors**
  retourner *False*
sinon
  retourner $\left[(-b+\sqrt{\Delta})/(2a),(-b-\sqrt{\Delta})/(2a)\right]$

Définition 2. *Un* programme *est un ensemble d'instructions écrit dans un langage de programmation donné.*

Le plus souvent, un programme est la traduction ou *implémentation* d'un algorithme dans un langage de programmation donné. Dans ce cas, l'algorithme est, en quelque sorte, le squelette abstrait du programme, indépendant d'un mode de codage particulier. Il donne une méthode universelle pour résoudre un type de problème donné.

Exemple :

```
def trinome(a,b,c):
    delta=b**2-4*a*c
    if delta<0:
        return False
    else:
        return [(-b+np.sqrt(delta))/(2*a),(-b-np.sqrt(delta))/(2*a)]
```

Remarque : Compte-tenu des limitations du format `float`, une telle implémentation n'est pas robuste en pratique.

2 Les instructions

Définition 3. *Les briques élémentaires d'un algorithme ou d'un programme sont les* instructions. *On en distingue trois types :*
- *les déclarations et affectations,*
- *les tests ou instructions conditionnelles,*
- *les boucles, conditionnelles ou inconditionnelles.*

Ainsi, aussi complexe soit-il, un programme est toujours écrit à l'aide de ces trois types d'instructions.

Définition 4. *On appelle* séquence d'instructions *une suite d'instructions consécutives.*

En pratique, on parle d'une séquence d'instructions pour des instructions qui se suivent et qui apparaissent dans un contexte précis : réponse à un test, instructions dans une boucle, etc. ...

II Instructions conditionnelles

1 Test simple

Définition 5. *Un* test simple *consiste en l'exécution d'une instruction ou d'une séquence d'instructions sous réserve qu'une condition soit réalisée.*

Syntaxe standard

```
if Condition:
    Instructions
```

On remarque l'indentation de l'action qui fait suite à la condition réalisée.

```
if␣Condition:
␣␣␣␣Instructions
```

Cette indentation est indispensable. Un bloc indenté sera réalisé dans une même séquence.

```
if␣Condition:
␣␣␣␣Instructions
␣␣␣␣Instructions
␣␣␣␣...
```

Exemple : Fonction qui calcule un maximum.

```
def maxi(a,b):
    """maxi(a:int or float,b:int or float)->int or float
    Renvoie max(a,b)"""
    if a>b:
        return a
    return b
```

Remarque : En fait, c'est une manière déguisée de faire un test avec alternative puisque le premier `return` provoque une sortie de la fonction et le second n'est donc relatif qu'à la situation `a<=b`.

Exercice : On considère une liste `L` de booléens de taille n codant un ensemble de la manière suivante : `i` est dans l'ensemble si et seulement si `L[i]` est vrai. Par exemple, l'ensemble $\{1, 3\}$ est codé par `[False,True,False,True]` ou aussi par `[False,True,False,True,False,False]` (il n'y a pas unicité du codage). Écrire une fonction `appartient(x,L)` qui détecte l'appartenance de `x` à `L`.

Corrigé : On saisit :

```
def appartient(x,L):
    """appartient(x:int,L:list)->bool
    L=[liste de booléens décrivant un ensemble]
    Renvoie le test d'appartenance de x à L"""
    if x<len(L):
        return L[x]
    return False
```

Si l'alternative concerne un bloc de plusieurs instructions, celles-ci sont toutes indentées.

```
def retire(x,L):
    """retire(x:int,L:list)->bool
    L=[liste de booléens décrivant un ensemble]
    Renvoie False si x absent, True si x présent et le supprime de L"""
    if x<len(L) and L[x]:
        L[x]=False
        return True
    return False
```

La fonction `retire(x,L)` renvoie `False` si `x` est absent de `L` et `True` sinon en le retirant de `L`, donc en basculant le booléen en indice `x` à `False`.

```
>>> a=[False,True,False,True]
>>> retire(2,a)
False
>>> retire(3,a)
True
>>> a
[False, True, False, False]
```

2 Test avec alternative

Définition 6. *Un* test avec alternative *consiste en l'exécution d'une instruction ou d'une séquence d'instructions sous réserve qu'une condition soit réalisée et d'une autre instruction ou séquence d'instructions sinon.*

Syntaxe standard

```
if Condition:
    Instructions
else:
    Instructions
```

On remarque là aussi l'indentation essentielle.

```
if␣Condition:
␣␣␣␣Instructions
else:
␣␣␣␣Instructions
```

Exemple : Soit f la fonction définie par

$$\forall x \in \mathbb{R} \qquad f(x) = \begin{cases} \dfrac{\sin x}{x} & \text{si } x \neq 0 \\ 1 & \text{sinon} \end{cases}$$

On l'implémente ainsi

```
def f(x):
    if x!=0:
        return np.sin(x)/x
    else:
        return 1
```

Exercice : Écrire une fonction avec test avec alternative (et sans boucle) qui calcule $\sum_{k=p}^{n} q^k$ où p, n et q sont saisis comme arguments.

Corrigé : On a

$$\sum_{k=p}^{n} q^k = \begin{cases} n-p+1 & \text{si } q = 1 \\ \dfrac{q^p - q^{n+1}}{1-q} & \text{si } q \neq 1 \end{cases}$$

```
def geom(p,n,q):
    """geom(p:int,n:int,q:float)->float
    Renvoie q^p+q^(p+1)+...+q^n"""
    if q==1:
        return n-p+1
    else:
        return (q**p-q**(n+1))/(1-q)
```

3 Tests imbriqués

Proposition 1. *Un algorithme peut contenir des tests imbriqués les un dans les autres pour répondre à des problèmes comportant des sous-alternatives.*

Exemple : Si n est impair, renvoyer 0, sinon, renvoyer 1 si n est multiple de 4 et -1 sinon.

```
def mod4(n):
    """mod4(n:int)->int
    Renvoie 0 si n impair, 1 si n multiple de 4 et -1 sinon"""
    if n%2==0:
        if n%4==0:
            res=1
        else:
            res=-1
    else:
        res=0
    return res
```

Exercice : Écrire une fonction `trinome` permettant la résolution d'une équation de degré 2 afin qu'une seule racine réelle soit renvoyée si le discriminant Δ est nul et deux racines distinctes si $\Delta > 0$.

Corrigé : On conditionne l'ajout d'une deuxième racine au test $\Delta > 0$.

```
def trinome(a,b,c):
    """trinome(a:float,b:float,c:float)->bool, float, tuple
    Renvoie les éventuelles racines réelles de aX^2+bX+c"""
    delta=b**2-4*a*c
    if delta<0:
        return False
    else:
        res=(-b+np.sqrt(delta))/(2*a)
        if delta>0:
            res=res,(-b-np.sqrt(delta))/(2*a)
        return res
```

4 Tests à alternatives multiples

Définition 7. *Un* test à alternatives multiples *consiste en l'exécution d'une instruction ou d'une séquence d'instructions sous réserve qu'une condition parmi d'autre soit réalisée.*

Syntaxe standard

```
if Condition 1:
    Instructions
elif Condition 2:
    Instructions
...
elif Condition n:
    Instructions
else:
    Instructions
```

Remarque : On pourrait faire usage de tests imbriqués pour distinguer différentes alternatives mais cette écriture n'est pas très adaptée. On lui préférera donc largement L'instruction `elif` dont l'écriture et l'indentation sont bien plus commodes pour traiter les différents cas de dissociation.

Exemple : Soit f définie sur $\mathbb{R}$ par

$$\forall x \in \mathbb{R} \qquad f(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } 0 \leqslant x \leqslant 1 \\ 1 & \text{si } x > 1 \end{cases}$$

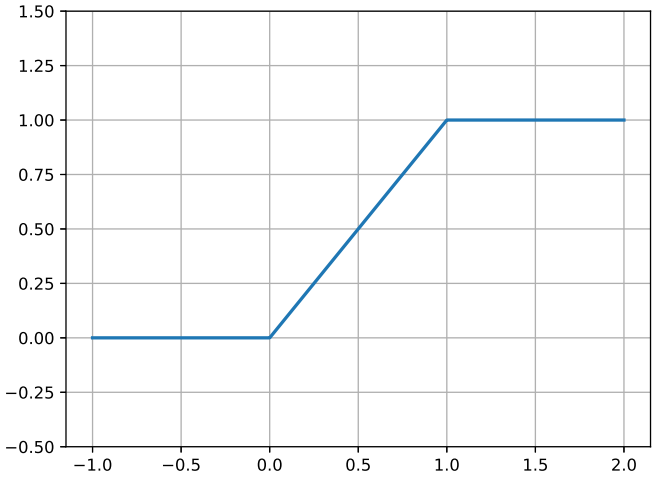

FIGURE 1 – Graphe de $y = f(x)$

```
def f(x):
    if x<0:
        return 0
    else:
        if x<=1:
            return x
        else:
            return 1
```

Le code est inutilement lourd et l'indentation suggère une hiérarchie des alternatives qui n'a pas lieu. La version avec `elif` pallie à ces défauts.

```
def f(x):
    if x<0:
        return 0
    elif x<=1:
        return x
    else:
        return 1
```

Remarque : On pourrait aussi écrire :

```
def f(x):
    if x<0:
        return 0
    if x<=1:
        return x
    else:
        return 1
```

Cette version fonctionne mais est un peu moins lisible : le deuxième `if` est en réalité une alternative au premier à cause de la présence du `return` sur le premier test.

Exercice : Calcul de l'impôt 2024

Le barème de l'impôt dû en 2024 est détaillé par le tableau qui suit :

Revenu imposable par part	Taux
Jusqu'à 11 295€	0%
de 11 295€ à 28 798€	11%
de 28 798€ à 82 342€	30%
de 82 342€ à 177 107€	41%
plus de 177 107€	45%

Par exemple, avec un revenu imposable par part égal à 15 000€, le foyer fiscal doit s'acquitter pour chaque part de $0.11\times(15\,000-11\,295)$€. Avec un revenu imposable par part égal à 30 000€, le foyer doit s'acquitter pour chaque part de $0.11\times(28\,798-11\,295)+0.30\times(30\,000-28\,798)$€, etc. ...

Écrire une fonction `impots(r)` d'argument `r` le revenu imposable par part et qui renvoie le calcul de l'impôt pour chaque part. On écrira une seule fois `return` dans le code.

Corrigé :

```
def impots(r):
    """impots(r:int or float)->int or float
    Calcul de l'impot sur le revenu"""
    S1,S2,S3,S4=11295,28798,82342,177107
    T1,T2,T3,T4=.11,.3,.41,.45

    if r<=S1:
        i=0
    elif r<=S2:
        i=T1*(r-S1)
    elif r<=S3:
        i=(S2-S1)*T1+(r-S2)*T2
    elif r<=S4:
        i=(S2-S1)*T1+(S3-S2)*T2+(r-S3)*T3
```

```
    else:
        i=(S2-S1)*T1+(S3-S2)*T2+(S4-S3)*T3+(r-S4)*T4
    return i
```

```
>>> impots(20000)
957.55
>>> impots(30000)
2285.93
>>> impots(100000)
25228.309999999998
```

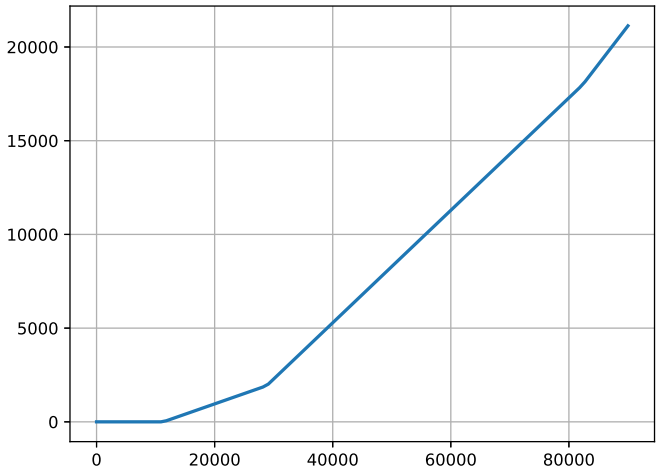

FIGURE 2 – Impôts 2024 en fonction du revenu imposable par part

III Les boucles

1 Les boucles inconditionnelles

Définition 8. *Une* boucle inconditionnelle *consiste en la répétition, par énumération des éléments d'un itérable, d'une instruction ou d'une séquence d'instructions.*

Les types composés liste, chaîne, range et tuple sont itérables.

Syntaxe standard

```
for Enumeration:
    Instructions
```

Comme pour les tests, les instructions indentées par rapport à l'instruction `for` appartiennent à une même séquence répétée tant que la condition est vraie. La fin de l'indentation correspond aux instructions réalisées à la sortie de la boucle.

```
for␣Enumeration:
␣␣␣␣Instructions
␣␣␣␣Instructions
␣␣␣␣...
```

⚠ Un `return` dans une boucle non précédé d'un `if` n'a pas de sens puisqu'il casse la boucle et rend celle-ci inopérante :

```
for Enumeration:
    Instructions
    ...
    return ...    # le return casse la boucle
```

Cette structure ne présentant aucun intérêt, elle est à proscrire.

⚠ **Quelques interdits :** On s'interdira totalement de modifier un ensemble que l'on parcourt dans une boucle `for` ou la variable qui réalise le parcours : pas de `append`, ni `pop`, ni `remove`, ni `del` ... Les exemples qui suivent sont à proscrire absolument :

```
res=[]
for k in range(10):
    k+=1
    res.append(k)
```

avec la variable `res` qui contient en sortie de boucle `[1, 2, ..., 10]` (la variable `k` est bien modifiée avant le `append` puis la modification est annulée lors du passage suivant dans la boucle) ou

```
L=[]
for x in L:
    L.append(0) # Boucle infinie!
```

ou

```
L=list(range(10))
res=[]
for x in L:
    res.append(x)
    L.pop()
```

avec la variable `res` qui contient en sortie de boucle `[0, 1, 2, 3, 4]`.

Exemples : 1. Calcul de la somme $\sum_{k=1}^{n} k$ avec n entier.

```
def arithsom(n):
    """arithsom(n:int)->int
    Renvoie 1+2+...+n"""
```

```
    res=0
    for k in range(n):
        res+=k+1
    return res
```

⚠ Pour calculer une somme, on utilise une variable locale `res` initialisée à zéro (élément neutre pour la loi $+$) qui sert « d'accumulateur » pour les termes successifs de la somme.

2. Calcul de $\binom{n}{k}$ où k et n des entier et $k \in [\![0\,;\, n]\!]$.

On a $$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \begin{cases} \dfrac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1} & \text{si } k > 0 \\ 1 & \text{si } k = 0 \end{cases}$$

On en déduit l'implémentation suivante :

```
def binom(n,k):
    """binom(n:int,k:int)->int
    Renvoie le nombre de combinaisons de k parmi n"""
    res=1
    for i in range(k):
        res=res*(n-i)//(i+1)
    return res
```

⚠ Pour calculer un produit, on utilise une variable locale `res` initialisée à un (élément neutre pour la loi $\times$) qui sert « d'accumulateur » pour les termes successifs du produit.

Exercice : Écrire une fonction `fact(n)` d'argument `n` entier et qui renvoie $n!$.

```
def fact(n):
    """fact(n:int)->int
    Renvoie la factorielle n!"""
    res=1
    for k in range(2,n+1):
        res*=k
    return res
```

Exemples importants : ⚠ à connaître

• Somme des éléments d'une liste

La fonction `somme(L)` d'argument `L` une liste de nombres renvoie la somme de ceux-ci.

```
def somme(L):
    """somme(L:list)->int or float
    Renvoie la somme des termes de L"""
    res=0
    for x in L:
```

```
        res+=x
    return res
```

On peut parcourir directement la liste `L` par une boucle `for`.

• Recherche d'un élément dans une liste

La fonction `detect(elt,L)` d'argument `elt` un objet et `L` une liste renvoie `True` si `elt` est présent dans `L` et `False` sinon. On remarque que l'on peut casser une boucle `for` avec un `return`.

```
def detect(elt,L):
    """detect(elt:any,L:list)->bool
    Test d'appartenance de elt à L"""
    for x in L:
        if x==elt:
            return True
    return False
```

Si on trouve l'élément dans `L`, l'instruction `return` casse la boucle `for`. Si celle-ci est exécutée complètement, c'est qu'on n'a pas trouvé l'élément d'où le dernier `return False`.

Si on souhaite que la fonction `detect` renvoie une position de l'élément quand celui-ci est présent dans la liste et `False` sinon, on saisit :

```
def detect(elt,L):
    """detect(elt:any,L:list)->bool
    Test d'appartenance de elt à L"""
    n=len(L)
    for k in range(n):
        if L[k]==elt:
            return k
    return False
```

L'utilisation du `range` permet de parcourir les indices de la liste. L'élément d'indice `k` de la liste `L` est obtenu par `L[k]`.

Exercice : Écrire une fonction `somme_sq(L)` d'arguments `L` une liste de nombres et qui renvoie la somme des carrés de ceux-ci.

Corrigé : On saisit :

```
def somme_sq(L):
    """somme_sq(L:list)->int or float
    Renvoie la somme des carrés des termes de L"""
    res=0
    for x in L:
        res+=x**2
    return res
```

Exercice : Écrire une fonction `prod(L)` d'arguments `L` une liste de nombres et qui renvoie le produit de ceux-ci.

Corrigé : On saisit :

```
def prod(L):
    """prod(L:list)->int or float
    Renvoie le produit des termes de L"""
    res=1
    for x in L:
        res*=x
    return res
```

2 Les listes par compréhension

Les *listes par compréhension* sont une application excessivement utile des boucles inconditionnelles.

Syntaxe standard

```
[f(x) for x in iterable]
```

Cette instruction construit la liste des valeurs de `f(x)` pour `x` parcourant un itérable (liste, chaîne, range).

Exemple :

```
>>> [k**2 for k in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [str(k) for k in range(10)]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

On peut utiliser des listes par compréhension pour construire des listes de sous-listes indépendantes mais identiques en contenu :

```
>>> a=[[] for k in range(5)]
>>> b=[[]]*5
>>> a
[[], [], [], [], []]
>>> b
[[], [], [], [], []]
>>> a[0].append(1)
>>> a
[[1], [], [], [], []]
>>> b[0].append(1)
>>> b
[[1], [1], [1], [1], [1]]
```

En affichant les contenus des variables `a` et `b`, ceux-ci semblent identiques. Mais en réalité, la variable `b` contient cinq étiquettes sur la même sous-liste. En modifiant le contenu de l'une, on les modifie toutes ! C'est donc le procédé utilisé pour la variable `a` qu'il faut utiliser.

3 Les boucles conditionnelles

Définition 9. *Une* boucle conditionnelle *consiste en la répétition d'une instruction ou d'une séquence d'instructions tant qu'une condition est réalisée.*

Syntaxe standard

```
while Condition:
    Instructions
```

Là encore, on souligne l'indentation des instructions exécutées sous conditions.

```
while␣Condition:
␣␣␣␣Instructions
␣␣␣␣Instructions
```

Comme pour les boucles inconditionnelles, les instructions indentées par rapport à l'instruction `while` appartiennent à une même séquence répétée tant que la condition est vraie. La fin de l'indentation correspond aux instructions réalisées à la sortie de la boucle.

Remarque : Une boucle conditionnelle peut être infinie et syntaxiquement correcte ! Il faut donc s'assurer, avant exécution d'un code, que la boucle va s'arrêter.

```
while True:
    print("boucle infinie...")
```

On observe :

```
boucle infinie...
boucle infinie...
boucle infinie...
boucle infinie...
boucle infinie...
boucle infinie...
...
```

qu'il faut casser avec la séquence de touches `ctrl+C`.

Exemple : Pour tout entier x, il existe un unique entier n et un unique entier impair d tel que

$$x = 2^n \times d$$

L'entier n est appelé 2-valuation de x. Par exemple

$$5 = 2^0 \times 5, \quad 8 = 2^3 \times 1, \quad 18 = 2^1 \times 9, \quad 28 = 2^2 \times 7$$

```
def val2(x):
    """val2(x:int)->int
    Renvoie la 2-valuation de x"""
    a,k=x,0
    while a%2==0:
        k+=1
        a//=2
    return k
```

Exercice : On considère la suite des sommes $\sum_{k=1}^{n} \frac{1}{\sqrt{k}}$ avec n entier non nul. On a

$$\forall n \in \mathbb{N}^* \qquad \sum_{k=1}^{n} \frac{1}{\sqrt{k}} \geqslant \frac{n}{\sqrt{n}} = \sqrt{n} \xrightarrow[n\to+\infty]{} +\infty$$

Écrire une fonction `seuil(p)` qui, pour un p entier non nul donné, détermine le plus petit entier n tel que

$$\sum_{k=1}^{n} \frac{1}{\sqrt{k}} \geqslant p$$

et qui renvoie également la somme.

Corrigé : On saisit :

```
def seuil(p):
    s,k=0,0
    while s<p:
        k+=1
        s+=1/np.sqrt(k)
    return k,s
```

Remarque : On a $$\sum_{k=1}^{n} \frac{1}{\sqrt{k}} \geqslant \sqrt{n}$$

D'où $$\sqrt{n} \geqslant p \Longrightarrow \sum_{k=1}^{n} \frac{1}{\sqrt{k}} \geqslant p$$

Ainsi, le nombre p^2 serait un candidat possible pour être le seuil de dépassement mais l'expérimentation montre qu'il dépasse très largement le seuil.

```
>>> seuil(54)
(769, 54.019372493901983)
>>> 54**2
2916
```

On pose $$\forall p \in \mathbb{N} \qquad \varphi(p) = \min \left\{ n \in \mathbb{N} \mid \sum_{k=1}^{n} \frac{1}{\sqrt{k}} \geqslant p \right\}$$

La fonction `seuil` est donc l'implémentation de la fonction φ. On observe :

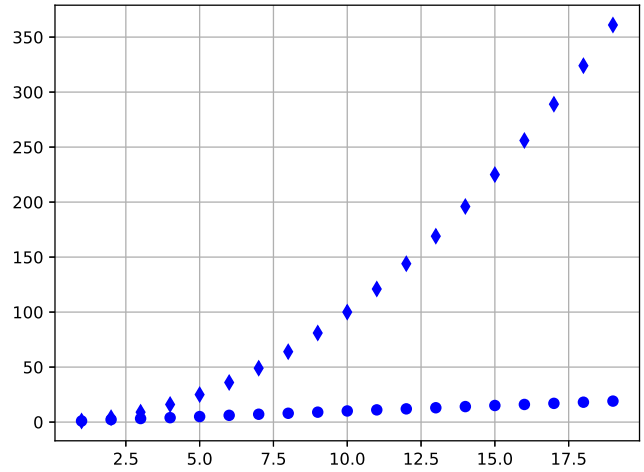

FIGURE 3 – Tracé des suites $(\varphi(p))_p$ et $(p^2)_p$