Le sujet comporte un total de 3 exercices indépendants
qui peuvent être traités dans l'ordre de votre choix.

Exercice N°1 – A la découverte de la notation polonaise inversée

Introduction

La notation polonaise inversée, encore appelée « post-fixée » (l'écriture classique des calculs est dite « infixée »), que nous noterons désormais NPI, est une méthode d'écriture des calculs permettant de s'affranchir des parenthèses. Elle est dérivée des travaux (1920) dans le domaine de la logique du mathématicien Jan WKASIEWICZ. Elle fut implémentée dans de nombreuses calculatrices de la société Hewlett-Packard (dont la célèbre HP41). Celle-ci fut malheureusement marginalisée par les sociétés Texas Instruments et Casio, notamment du fait du prix élevé de ses machines. Aujourd'hui les calculatrices HP n'incorporent plus systématiquement ce seul mode de saisie et lorsqu'il existe, il est également possible de saisir les calculs selon l'approche infixée.

Un exemple et quelques définitions

Supposons que nous souhaitions effectuer le calcul suivant : $5\times(13+34)$.

En NPI, un tel calcul pourra être noté : $34\,13+5\times$.

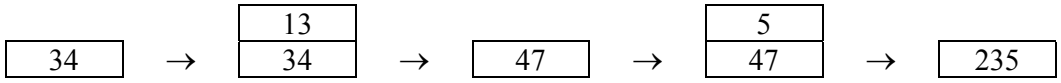
Le calcul est traité comme une liste de symboles. Un groupe de symboles sans blanc (comme « 34 » ou « 13 » ou « + ») est appelé « mot » et les mots sont séparés par des blancs (un seul blanc entre deux mots consécutifs). Le calcul ci-dessus comporte donc 5 mots. Il y a deux types de mot : les « opérateurs » (« + » et « × ») et les « opérandes » (« 34 », « 13 » et « 5 »). Ainsi, on comprend mieux l'appellation « post-fixée » : dans le calcul, l'opérateur est placé **après** les opérandes (à l'origine, la notation de WKASIEWICZ était une notation préfixée).

Principe général et implémentation

Un interpréteur balaie la liste de symboles correspondant au calcul de la gauche vers la droite et gère une pile de la façon suivante :

- Si le mot rencontré est un opérande, il est empilé.
- Si le mot rencontré est un opérateur binaire (tels « + » et « × ». Nous ne considèrerons que des opérateurs binaires ici), on dépile deux éléments de la pile et on leur applique l'opérateur. Le résultat est ensuite empilé.

Avec notre exemple, la pile associée évoluera comme suit (nous ne représentons pas la pile initialement créée, vide à ce moment-là) :



On notera que dans notre exemple, la notation $5\ 34\ 13\ +\ \times$ est aussi une NPI pour le calcul $5\times(13+34)$.

Questions

Dans ce qui suit, on suppose que les opérandes sont des entiers naturels et on ne considère que les deux opérateurs « + » (addition) et « × » (multiplication).

1. Ecrire le calcul suivant en NPI :

$$(7+12)\times(5+11\times(2+35+19))$$

Diverses écritures sont possibles. Si on souhaite retarder le plus possible les opérations, on pourra écrire :

$$7\ 12\ +\ 5\ 11\ 2\ 35\ 19\ +\ +\ \times\ +\ \times$$

Note : ce calcul vaut 11 799. Par ailleurs, la notation NPI comporte 6 symboles de moins que la notation infixée …

2. Quelle valeur fournira le calcul suivant écrit en NPI ?

$$21\ 5\ 8\ \times\ 7\ 13\ \times\ +\ \times$$

On effectue l'analyse suivante :

$$\underbrace{21\ \underbrace{\underbrace{5\ 8\ \times}_{(5\times8)}\ \underbrace{7\ 13\ \times}_{(7\times13)}}_{(5\times8)+(7\times13)}\ +\ \times}_{21\times((5\times8)+(7\times13))}$$

La calcul en écriture infixée s'écrit alors :

$$21\times\big((5\times8)+(7\times13)\big)$$

Soit : $21\times\big((5\times8)+(7\times13)\big)=21\times(40+91)=21\times131=2\,751$.

Le calcul $21\ 5\ 8\ \times\ 7\ 13\ \times\ +\ \times$ fournira la valeur 2 751.

3. On souhaite écrire une fonction Python `ValNPI` qui recevra en argument une chaîne de caractères `s` correspondant à l'écriture NPI d'un calcul et qui retournera la valeur de ce calcul. Dans ce cadre, la position d'un mot sera l'indice, dans la chaîne *s*, du premier caractère du mot. Ainsi, avec `s = '34 13 + 5 x'`, la position du mot « `13` » est 3 et celle du mot « `x` » est 10.

   Pour coder la fonction `ValNPI`, vous disposez :

   - De fonctions de manipulation de piles, notamment : `stack_create` (création d'une pile vide), `stack_isempty` (renvoie un booléen valuant True ou False selon que la pile est vide ou pas), `stack_pop` (récupération et destruction du sommet) et `stack_push` (ajout d'un élément au sommet).
   - D'une fonction `next_word` qui reçoit deux arguments : une chaîne de caractères `s` et un entier `pos`.
     - Si `pos < len(s)`, la fonction renvoie une liste :
       - Vide si `pos` ne correspond pas à l'indice du premier caractère d'un mot.
       - Comportant un seul élément, le mot en position `pos` si ce mot est le dernier mot de la liste `s`.
       - Comportant deux éléments : le mot en position pos et, si ce mot n'est pas le dernier élément de la liste, la position du mot suivant.
     - Si `pos >= len(s)`, la fonction renvoie la liste vide.

   Par exemple, avec `s = '34 13 + 5 x'` :
   - `next_word(s,0)` renverra `['34',3]`
   - `newt_word(s,3)` renverra `['13',6]`
   - `next_word(s,6)` renverra `['+',8]`
   - `next_word(s,8)` renverra `['5',10]`
   - `next_word(s,10)` renverra `['x']`

a. Coder la fonction `ValNPI`.

   Cf. le fichier Python NPI.py accessible depuis votre page.

b. **Facultatif** : coder la fonction `next_word`.

   Cf. le fichier Python NPI.py accessible depuis votre page.

Exercice N°2 – Un peu de conversion

Cet exercice comporte trois parties. Les parties A et B sont indépendantes l'une de l'autre.

Partie A : Conversion binaire → décimal

Si l'entier naturel *n* admet pour écriture en base 2 : 110101, alors son écriture décimale (c'est-à-dire son écriture en base 10) est obtenue via le calcul :
$1\times 2^5+1\times 2^4+0\times 2^3+1\times 2^2+0\times 2^1+1\times 2^0$ qui donne $32+16+4+1=53$.

1. Ecrire une fonction `bin2dec` qui reçoit en argument l'écriture binaire d'un entier naturel n (cette écriture est en fait un entier naturel ne comportant que des « 0 » et des « 1 ») et renvoie l'écriture décimale de cet entier.

   Par exemple, on devra avoir :

   ```
   >>> x = bin2dec(110101)
   >>> x
   >>> 53
   ```

   On donnera une version itérative de cette fonction en utilisant une boucle `while` et la fonction `divmod`.

   On rappelle que `divmod(a,b)` renvoie, pour `a` et `b` entiers, le quotient et le reste de la division euclidienne de `a` par `b`. Ainsi, l'instruction `q, r = divmod(23,5)` donne à `q` la valeur `4` et à `r` la valeur `3`.

   On met en place une fonction comportant principalement une boucle `while`, le principe étant de travailler, à chaque étape avec le chiffre des unités, obtenu grâce à la fonction `divmod`.

   Se reporter au fichier Python bin2dec.py accessible depuis votre page.

Partie B : Conversion décimal → binaire

On se donne l'écriture décimale d'un entier *n* et on en cherche l'écriture binaire.
La démarche consiste à décomposer l'entier en une combinaison linéaire de puissances de 2, les coefficients étant égaux à 0 ou 1 (une telle décomposition est alors unique).
Par exemple :

$$\begin{aligned}3756 &= 2048+1024+512+128+32+8+4\\ &= 1\times 2^{11}+1\times 2^{10}+1\times 2^9+1\times 2^7+1\times 2^5+1\times 2^3+1\times 2^2\\ &= 1\times 2^{11}+0\times 2^{10}+1\times 2^{10}+1\times 2^9+0\times 2^8+1\times 2^7+0\times 2^6\\ &\quad +1\times 2^5+0\times 2^4+1\times 2^3+1\times 2^2+0\times 2^1+0\times 2^0\end{aligned}$$

En recopiant les seuls coefficients, on obtient l'écriture binaire de 3756 : 111010101100.

On notera que 2048 est la plus grande puissance de 2 inférieure ou égale à 3756 et on rappelle que tout entier naturel *n* non nul peut être encadré comme suit : $2^N \leq n < 2^{N+1}$ où *N* est un entier naturel.

Pour effectuer la conversion décimal → binaire, on propose la fonction récursive suivante :

```
def dec2bin(n):
    if n <= 1:
        return n
    else:
        N = int(log2(n))
        return 10**N + dec2bin(n - 2**N)
```

Remarque : `log2` désigne le logarithme de base 2 et `log2(n)` équivaut à `log(n)/log(2)`.

2. Expliquer précisément le fonctionnement de la fonction `dec2bin`.

Tout entier naturel *n* s'écrit de façon unique : $n = \sum_{k=0}^{N} a_k 2^k = a_N 2^N + a_{N-1} 2^{N-1} + ... + a_1 2 + a_0$ où tous les a_k valent 0 ou 1. Par ailleurs, *N* est l'unique entier naturel tel que :

$$2^N \leq n < 2^{N+1}$$

c'est-à-dire l'exposant de la plus grande puissance de 2 inférieure ou égale à *n*.

De cette double inégalité on tire alors : $\log_2\left(2^N\right) \leq \log_2(n) < \log_2\left(2^{N+1}\right)$, c'est-à-dire :

$$N \leq \log_2(n) < N+1$$

D'où : $N = \mathrm{E}\left(\log_2(n)\right)$.

Ainsi, on se donne l'entier naturel *n* écrit sous forme décimale (i.e. en base 10).
S'il vaut 0 ou 1, son écriture binaire est … identique à son écriture décimale (c'est même vrai pour n'importe quelle base !).
Sinon, on calcule $N = \mathrm{E}\left(\log_2(n)\right)$ (instruction `int(log2(n))`) et on a $a_N = 1$ qui donne un « 1 » à la $N+1$ ème position (d'où le premier terme de ce qui est renvoyé par le `return`).
On recommence alors avec $n - a_N 2^N = \sum_{k=0}^{N-1} a_k 2^k = a_{N-1} 2^{N-1} + ... + a_1 2 + a_0$. D'où le deuxième terme de ce qui est renvoyé par le `return` : appel récursif de la fonction elle-même avec `n - 2**N` comme argument.

Partie C : Conversion vers une base « quelconque »

On souhaite obtenir l'écriture en base *b* (*b* est un entier naturel supérieur ou égal à 2 et que l'on supposera strictement inférieur à 10 …) d'un entier naturel *n* dont on connaît l'écriture décimale.

3. En vous aidant de la fonction `dec2bin`, écrire une fonction `dec2base` recevant comme arguments un entier naturel `n` et la base `b` et effectuant la conversion demandée.

   ATTENTION ! En base *b*, les chiffres autorisés sont 0, 1, 2, …, $b-1$.

Se reporter au fichier dec2base.py accessible depuis votre page.

Exercice N°3 – Le tri boustrophédon

1. Une procédure utile au tri à bulles

Rappelons le fonctionnement précis de `range` à travers l'exemple suivant (écrit dans une console Python) :

```
>>> for i in range(5, 1, -1):
        print(i, end = " ")
5 4 3 2
```

On donne l'algorithme suivant :

```
def redescendre(L):
    for j in range(len(L)-1, 0, -1):
        if L[j] < L[j-1]:
            L[j], L[j-1] = L[j-1], L[i]
    return(L)
```

Qu'affiche la commande `print(redescendre([5,4,3,2,1]))` ?

On obtient la liste : `[1, 5, 4, 3, 2]`.

2. Mécanisme du tri à bulles

Considérons le programme suivant :

```
def trier(L):
    """ Trie la liste passée en argument. """
    for i in range(len(L)):
        L = redescendre(L)
        print(L)
```

Qu'affiche la commande `trier([5, 4, 3, 2, 1])` ?

On obtient :

```
[1, 5, 4, 3, 2]
[1, 2, 5, 4, 3]
[1, 2, 3, 5, 4]
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

3. Débogage

Dans une première version erronée de `redescendre()`, on avait écrit −1 à la place de 0. Il en résultait qu'en exécutant les instructions suivantes, les listes affichées changeaient à chaque ligne au lieu de rapidement converger vers la liste `[0,1,2,3]`). Pourquoi ?

```
L = [2,1,3,0]
while 1:
     L = redescendre(L)
     print(L)
```

Si on remplace le 0 par un -1, la dernière comparaison dans la boucle for se fera entre l'élément d'indice 0 et celui d'indice -1, c'est-à-dire le dernier de la liste. Or le dernier de la liste sera rapidement plus grand que le terme d'indice 0, donc ils seront échangés. Cet échange désordonne complètement la liste que les prochaines itérations tenteront de rectifier, jusqu'à ce qu'à nouveau, le terme d'indice 0 soit échangé avec celui d'indice -1. En fait, rapidement, la liste sera triée à permutation cyclique près, mais à chaque itération de la boucle `while`, elle subira un décalage cyclique vers la gauche.

4. Amélioration

On se propose d'améliorer les performances de `trier()` sur deux axes :
– la liste est souvent triée bien avant que la boucle de `trier()` ne s'achève.
– il n'est pas non plus nécessaire que la boucle de `redescendre()` aille jusqu'au bout.

Réécrire la fonction `redescendre()` pour que la nouvelle fonction trier() ci-dessous soit fonctionnelle.

```
def trier(L):
    gauche = 0
    while gauche < len(L):
        L, gauche = redescendre(L, gauche)
    return L
```

En particulier :

– la nouvelle fonction `redescendre()` prend deux paramètres, et renvoie le tuple `(L, gauche)`.
– la boucle de la nouvelle fonction `redescendre()` devra faire le moins possible de tours.

```
def redescendre4(L, gauche):
    new_gauche = len(L)
    for i in range(len(L)-1, gauche, -1):
        if L[i] < L[i-1]:
            L[i], L[i-1] = L[i-1], L[i]
            new_gauche = i
    return (L, new_gauche)
```

5. Faire progresser les bulles dans l'autre sens

Ecrire une fonction `remonter()` sur le même principe que la fonction `redescendre()` présentée en début d'exercice de sorte que le programme suivant :

```
def trier(L):
    for i in range(len(L)):
        L = remonter(L)
        print(L)
```

produise :

```
>>> trier([6,5,1,9,7,2])
[5, 1, 6, 7, 2, 9]
[1, 5, 6, 2, 7, 9]
[1, 5, 2, 6, 7, 9]
[1, 2, 5, 6, 7, 9]
[1, 2, 5, 6, 7, 9]
```

```
def remonter(L):
    """ Faire évoluer les bulles à gauche. """
    for j in range(0, len(L)-1):
        if L[j+1] < L[j]:
            L[j], L[j+1] = L[j+1], L[j]
    return(L)
```

6. Le tri boustrophédon

Enfin, réécrire les fonctions `redescendre()` et `remonter()` pour que le nouveau programme `trier()` soit opérationnel.

```
def tri_boust(L):
""" Tri boustrophedon. """
     gauche, droite = 0, len(L)-1
     while gauche < droite:
          L, droite = remonter(L, gauche, droite)
          L, gauche = redescendre(L, gauche, droite)
return L
```

```
def remonter(L, gauche, droite):
""" Faire remonter les bulles à droite. """
    new_droite = 0
    for i in range(gauche, droite):
        if L[i] > L[i+1]:
            L[i], L[i+1] = L[i+1], L[i]
            new_droite = i
    return (L, new_droite)

def redescendre(L, gauche, droite):
""" Faire évoluer les bulles à gauche. """
    new_gauche = droite
    for i in range(droite, gauche,-1):
        if L[i] < L[i-1]:
            L[i], L[i-1] = L[i-1], L[i]
            new_gauche = i
    return (L, new_gauche)
```

7. Apport du tri boustrophédon

Déterminer un cas où le tri boustrophédon est plus efficace que le tri à bulles de la question 4.

Considérons la liste `[n]+[i for i in range(n, 0, -1)]`. Elle est classée en une étape et n comparaisons par le tri boustrophédon, alors qu'elle est classée en $n-1$ étapes et $\frac{n(n-1)}{2}$ comparaisons par le tri à bulles descendantes.