Informatique — MPSI/PCSI

Correction du TP n° 14

Exercice 14.1 (Échauffement)
Dans sa formulation classique, la fonction factorielle peut être définie pour tout $n > 0$ par :

$$n! = \begin{cases} 1 & \text{si } n = 1 \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

En implémentant cette expression sous la forme :

```
def factorielle(n):
    if n==0 or n==1:
        return(1)
    else:
        return(n*factorielle(n-1))
```

son exécution pour tout $n > 1$ conduit aux appels récursifs successifs en n, puis $n-1$, puis $n-2$, ..., jusqu'à 1. Le résultat est alors construit en phase de remontée. Pour définir une implémentation récursive terminale, il est nécessaire de définir une suite d'accumulateurs $(f_k)_{0 \leqslant k \leqslant n}$, initialisée à $f_0 \leftarrow 1$ (élément neutre du produit), et telle qu'à l'étape $k \in [\![0, n-1]\!]$, on ait :

$$f_{k+1} = (n-k) \times f_k = \underbrace{(n-k) \times \ldots \times (n-1) \times n}_{k+1 \text{ termes}} = \prod_{\xi=n-k}^{n} \xi$$

et vérifiant bien $f_n = n!$. L'implémentation ci-dessous convient.

```
def factorielle(n, f=1):
    if n==0 or n==1:
        return(f)
    else:
        return(factorielle(n-1, f*n))
```

Dans la suite du corrigé, on note $L = (L_i)_{0 \leqslant i < n}$ une séquence de $n > 0$ entiers.

Exercice 14.2 (Extrema)

1. On utilise la méthode du candidat. Dans la version itérative, on initialise à $c = L_0$ le candidat à la minimalité et on parcourt ensuite successivement les $(n-1)$ éléments suivants, en mettant à jour le candidat c si l'élément L_i (d'indice i) est plus petit que c ; soit :

$$\forall i \in [\![1, n-1]\!], \quad c \leftarrow \begin{cases} L_i & \text{si } L_i < c \\ c & \text{sinon.} \end{cases}$$

   Pour le traduire de façon récursive, il faut « convertir » la boucle `for` avec les $i \in [\![1, n-1]\!]$ croissants en appels récursifs avec le variant $(n-i)$. Il est donc commode d'initialiser le candidat avec le dernier élément $c = L_{n-1}$ puis d'utiliser l'invariant :

$$\forall i \in [\![0, n-2]\!], \quad \min_{i \leqslant k < n} (L_k) = \begin{cases} c = \min\limits_{i+1 \leqslant k < n} (L_k) & \text{si } L_i \geqslant c \\ L_i & \text{sinon} \end{cases}$$

   la condition d'arrêt étant $i = n-1$.
2. L'implémentation ci-dessous convient.

```
def mini(L,i=0):
    if i<len(L)-1:
        c = mini(L,i+1)
        if L[i] < c:
            c = L[i]
    else:
        c = L[i]
    return(c)
```

3. Soit $P(i)$, $i \in [\![0, n-1]\!]$, la proposition : « *c est le minimum des* $(i+1)$ *derniers termes de* L. » On le note c_i.
   — On initialise le candidat à $c_0 \leftarrow L_{n-1}$ donc $P(0)$ est vraie.
   — Soit $i \in [\![0, n-2]\!]$. On suppose que $P(i)$ est vérifiée ; alors c_i est le minimum des $(i+1)$ derniers termes de L, d'indices $k \in [\![n-i-1, n-1]\!]$. Si le terme précédent $L_{n-i-2} < c_i$ alors $c_{i+1} \leftarrow L_{n-i-2}$. Sinon, $c_{i+1} \leftarrow c_i$. Dans un cas comme dans l'autre, c_{i+1} contient la valeur du minimum des $(i+2)$ derniers termes de L. Donc $P(i+1)$ est vérifiée.

   Donc P est un invariant de boucle et le programme est partiellement correct.
   Le variant $(n-i)$ est initialisé à n (valeur de $i=0$ par défaut à l'appel de `mini(L)`) et décroit strictement (appel en $i+1$ à chaque itération) jusqu'à 1 (on fixe $i = n-1$ comme condition d'arrêt, correspondant à l'initialisation du candidat). Donc le programme termine.
   Comme le programme est partiellement correct et qu'il termine, il est correct.
4. Ce programme utilise une variable en plus de la séquence de n entiers, soit une complexité spatiale en $O(n)$.
   Comme on réalise $(n-1)$ appels récursifs avec pour chacun un test, la complexité temporelle est en $O(n)$.
5. Pour traduire notre fonction en fonction récursive terminale, il est tout d'abord nécessaire de comprendre que notre fonction constitue une pile d'appels pour aller chercher la dernière valeur, puis met à jour le candidat « à la dépile », ce qui est très gourmand en mémoire.
   Pour éviter de constituer une telle pile, il est nécessaire de définir un accumulateur associé au candidat, de sorte que le minimum de L soit obtenu directement au dernier appel, sans constituer de pile et donc sans besoin de la dépiler. Afin de pouvoir l'initialiser facilement, on choisit de fixer son indice $c = 0$. Le premier élément à tester a donc l'indice $i = 1$. Si $L_i < L_c$, on met à jour l'indice du candidat. Puis, on appelle la fonction pour tester l'indice d'après, jusqu'à $i = n$ où on renvoie alors L_c. Le code suivant convient.

```
def mini(L,c=0,i=1):
    if i<len(L):
        if L[i]<L[c]:
            c=i
        return(mini(L,c,i+1))
    else:
        return(L[c])
```

6. Pour définir une fonction permettant de trouver les deux extrema, il suffit d'ajouter dans la formulation la recherche du maximum, dont l'indice `M` peut aussi être initialisé à 0. En renommant simplement `m` l'indice du minimum dans le code précédent, il vient le code suivant.

```
def extrema(L,m=0,M=0,i=1):
    if i<len(L):
        if L[i]<L[m]:
            m=i
        elif L[i]>L[M]:
            M=i
        return(extrema(L,m,M,i+1))
    else:
        return(L[m],M[M])
```

Exercice 14.3 (Recherche dans une séquence triée)

1. On utilise une recherche par dichotomie. On suppose la liste L triée de façon croissante et x l'objet que l'on recherche dans la liste. Dans la version itérative, on initialise à $(i, j) = (0, n-1)$ les deux

bornes de l'intervalle et, après s'être assuré que $x \geqslant L_0$ et $x \leqslant L_{n-1}$, on évalue l'élément de L d'indice $m = \left\lfloor \dfrac{i+j}{2} \right\rfloor$:

— si $L_m = x$, on renvoie m ;
— si $L_m > x$, alors on conserve le demi-intervalle de gauche et $j \leftarrow m-1$;
— sinon, on conserve le demi-intervalle de droite et $i \leftarrow m+1$.

On continue ce processus tant que $j - i > 1$. Ainsi, lorsque $j = i+1$, soit x est en position L_i ou L_j, soit il n'est pas dans L. Pour le traduire de façon récursive, il faut « convertir » la boucle `while j-i>1` en appels récursifs avec le variant $(j-i)$. Pour l'invariant, il est nécessaire de construire deux suites (i_k) et (j_k) telles que :

$$\forall k \in [\![0, m]\!], \quad x \in L \implies x \in (L_\xi)_{i_k \leqslant \xi \leqslant j_k}$$

où $m = -\left\lfloor 1 - \dfrac{\ln(n)}{\ln(2)} \right\rfloor$ est l'entier tel que $2^m < n \leqslant 2^{m+1}$ et telles que l'on ait à chaque nouveau découpage d'indice k, notant $m_k = \left\lfloor \dfrac{i_k + j_k}{2} \right\rfloor$ l'indice du « milieu » de $[\![i_k, j_k]\!]$, l'évolution :

$$x \in (L_\xi)_{i_k \leqslant \xi \leqslant j_k} = \begin{cases} x \in (L_\xi)_{i_k \leqslant \xi < m_k} & \text{si } x < L_{m_k} \\ x \in (L_\xi)_{m_k < \xi \leqslant j_k} & \text{si } x > L_{m_k} \\ x = L_{m_k} & \text{sinon.} \end{cases}$$

2. L'implémentation ci-dessous convient.

```
def dichotomie(L,x,i=0,j=None):
    if j==None:
        j=len(L)-1
    if x<L[i] or x>L[j]: # vérification des bornes
        return(False)
    else:
        if j>i+1: # pour un intervalle de plus de 2 élements
            m=(i+j)//2
            if L[m]==x:
                return(m)
            elif L[m]>x:
                j=m
            else:
                i=m
            return(dichotomie(L,x,i,j))
        else: # ne restent que deux éléments à étudier
            if L[i]==x:
                return(i)
            elif L[j]==x:
                return(j)
            else:
                return(False)
```

3. Soit $P(k)$, $k \in [\![0, m]\!]$, $m = -\left\lfloor 1 - \dfrac{\ln(n)}{\ln(2)} \right\rfloor$, la proposition : « *Si x est dans L, alors* $x \in (L_\xi)_{i_k \leqslant \xi \leqslant j_k}$. »

— On initialise les indices de l'intervalle $(i, j) \leftarrow (0, n-1)$ donc $P(0)$ est vraie.
— Soit $k \in [\![0, m-1]\!]$. On suppose que $P(k)$ est vérifiée ; donc si $x \in L$, alors $x \in (L_\xi)_{i_k \leqslant \xi \leqslant j_k}$. On évalue le terme de L d'indice $m_k = \left\lfloor \dfrac{i_k + j_k}{2} \right\rfloor$. Si $L_{m_k} < x$ alors x se trouve parmi les éléments d'indices $[\![i_k, m_k - 1]\!]$, donc $(i_{k+1}, j_{k+1}) \leftarrow (i_k, m_k - 1)$. Sinon, si $L_{m_k} > x$ alors x se trouve parmi les éléments d'indices $[\![m_k + 1, j_k]\!]$, donc $(i_{k+1}, j_{k+1}) \leftarrow (m_k + 1, j_k)$. Dans un cas comme dans l'autre, si x est dans L, alors il se trouve parmi les éléments d'indices $[\![i_{k+1}, j_{k+1}]\!]$, donc $P(k+1)$ est vraie. Enfin, si $L_{m_k} = x$, alors $x \in L$ et on court-circuite les appels récursifs.

Donc P est un invariant de boucle et le programme est partiellement correct.

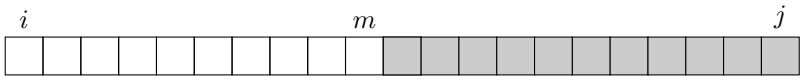
Un variant raisonnable est $j - i$ qui décroit strictement à chaque appel vers 1 (cas de 2 cases adjacentes). Donc le programme termine.
Comme le programme termine et qu'il est partiellement correct, alors il est correct.

4. Avec au plus m divisions de l'intervalle, $m = -\left\lfloor 1 - \frac{\ln(n)}{\ln(2)} \right\rfloor$, la complexité temporelle est en $O(m) = O(\ln n)$. En négligeant l'espace mémoire occupé par le triplet d'indices `(i,j,m)` devant celui occupé par la liste `L`, la complexité spatiale est en $O(n)$.
5. Cette fonction est déjà récursive terminale, car il n'y a pas d'étape de remontée.

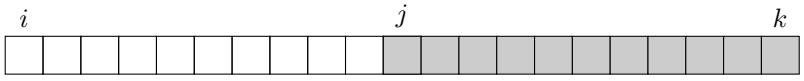
Exercice 14.4 (Tri)

1. L'algorithme de tri le plus efficace pour une séquence d'entiers est le tri-comptage. Il s'agit de déterminer dans un premier temps les valeurs minimale m et maximale M de la séquence L afin de construire la séquence des nombres d'occurrences de chaque entier de L, de longueur $p = M - m + 1$, que l'on note N ; ce qui se fait en $O(n)$. On reconstruit ensuite la séquence triée en parcourant N ; ce qui se fait en $O(p)$. Cette méthode est en $O(n) + O(p) \approx O(n)$. Le problème de cette méthode est que l'on n'agît pas sur place.

   Le second tri le plus efficace (en $n \ln(n)$) est le tri-fusion. Le principe de l'algorithme est de couper la séquence en deux morceaux et de répéter l'opération sur les sous-séquences jusqu'à obtenir des séquences de longueur 1 (déjà triées donc). Il faut ensuite fusionner ces séquences (triées) par paires, pour former une plus grande séquence triée.
   Pour trier « sur place », il est nécessaire de remarquer que les sous-séquences à trier ou à fusionner sont forcément contigües. Ainsi, on utilisera pour la phase de découpage les indices i, j correspondant aux indices des éléments des extrémités de la sous-séquence de L à découper $(L_\xi)_{i \leqslant \xi \leqslant j}$ et $m = \left\lfloor \frac{i+j}{2} \right\rfloor$ celui du « milieu », à droite duquel découper ; ce que l'on peut représenter comme :



   À l'issue de cette découpe, il s'agit ensuite d'appeler récursivement la fonction sur chacune des deux sous-séquences ayant chacune pour intervalles d'indices $[\![i, m]\!]$ et $[\![m+1, j]\!]$, si elles sont de longueur strictement supérieure à 1.
   Pour la phase de fusion, on choisit d'utiliser le triplet (i, j, k), tel que les deux sous-séquences triées à fusionner soient $(L_\xi)_{i \leqslant \xi < j}$ et $(L_\xi)_{j \leqslant \xi \leqslant k}$.



   Ainsi, deux cas se présentent :
   — soit $L_i < L_j$, auquel cas L_i est bien placé et il faut ensuite opérer la fusion des sous-séquences $(L_\xi)_{i+1 \leqslant \xi < j}$ et $(L_\xi)_{j \leqslant \xi \leqslant k}$;
   — soit $L_j < L_i$, auquel cas, pour conserver le caractère trié de chacune des sous-séquences, il est nécessaire de sauvegarder $a \leftarrow L_j$, puis d'opérer un glissement :

$$\forall \xi \in [\![0, j-i-1]\!], \quad L_{j-\xi} \leftarrow L_{j-\xi-1}$$

   puis d'affecter la valeur sauvegardée à $L_i \leftarrow a$. Il faut ensuite opérer la fusion des sous-séquences $(L_\xi)_{i+1 \leqslant \xi < j+1}$ et $(L_\xi)_{j+1 \leqslant \xi \leqslant k}$.
   Dans les deux cas, i est incrémenté ; mais j n'est incrémenté que dans le cas $L_j < L_i$.
2. On veille à faire la fusion de deux sous-séquences d'au moins un élément. L'implémentation ci-dessous correspond à l'algorithme présenté ci-dessus et convient pour la fusion.

```
def fusion(L,i,j,k):
    if j>i and k+1>j:
        if L[i]>L[j]:
            a = L[j]
            for s in range(j,i,-1):
                L[s]=L[s-1]
            L[i]=a
```

```
            j+=1
        i+=1
        fusion(L,i,j,k)
```

Pour le tri-fusion, on initialise les indices par défaut à ceux correspondants à la longueur de L, puis, si la sous-séquence est de longueur supérieure ou égale à deux, alors on la scinde en deux sous-séquences que l'on trie, puis fusionne. L'implémentation ci-dessous convient.

```
def tri(L,i=0,j=None):
    if j==None:
        j=len(L)-1
    if j>i:
        m=(i+j)//2
        if m>i:
            tri(L,i,m)
        if j>m+1:
            tri(L,m+1,j)
        fusion(L,i,m+1,j)
```

3. Pour simplifier l'étude de l'algorithme, on suppose que la séquence est de longueur $n = 2^p$. On commence par étudier la fusion. Soit $P(q)$, $k \in [\![0, p]\!]$, la proposition : « *les* 2^{p-q} *sous-séquences de* L *d'indices* $[\![c \times 2^q, (c+1) \times 2^q - 1]\!]$, $c \in [\![0, 2^{p-q} - 1]\!]$, *sont triées.*

   — Pour $q = 0$, les 2^p séquences de 1 élément sont, par définition, triées. Donc $P(0)$ est vraie.
   — Soit $q \in [\![1, p]\!]$. On suppose que $P(q)$ est vérifiée. La fusion opère sur les couples de sous-séquences de L d'indices $[\![c \times 2^q, (c+1) \times 2^q - 1]\!]$ et $[\![(c+1) \times 2^q, (c+2) \times 2^q - 1]\!]$, avec $c \in [\![0, 2^{p-q-1} - 1]\!]$. On note $i = c \times 2^q$, $j = (c+1) \times 2^q$ et $k = (c+2) \times 2^q - 1$. Comme décrit précédemment, deux cas se présentent :
      — soit $L_i < L_j$, auquel cas L_i est bien placé et il faut ensuite opérer la fusion des sous-séquences $(L_\xi)_{i+1 \leqslant \xi < j}$ et $(L_\xi)_{j \leqslant \xi \leqslant k}$;
      — soit $L_j < L_i$, auquel cas, pour conserver le caractère trié de chacune des sous-séquences, il est nécessaire de sauvegarder $a \leftarrow L_j$, puis d'opérer un glissement :

$$\forall \xi \in [\![0, j - i - 1]\!], \quad L_{j-\xi} \leftarrow L_{j-\xi-1}$$

      puis d'affecter la valeur sauvegardée à $L_i \leftarrow a$. Il faut ensuite opérer la fusion des sous-séquences $(L_\xi)_{i+1 \leqslant \xi < j+1}$ et $(L_\xi)_{j+1 \leqslant \xi \leqslant k}$.

   Dans un cas comme dans l'autre, l'élément d'indice i est le plus petit des éléments de $(L_\xi)_{i \leqslant \xi \leqslant k}$, ce qui nous permet d'initialiser le processus de fusion à répéter avec un second invariant.

      Soit $P_2(i)$, $i \in [\![c \times 2^q, (c+2) \times 2^q - 1]\!]$, la proposition : « *la sous-séquence de* L *d'indices* $[\![c \times 2^q, i]\!]$ *est triée et tous ses éléments sont inférieurs ou égaux à* $\min (L_\xi)_{i < \xi < (c+2) \times 2^q}$. Par construction, $P_2(0)$ est vraie. Soit $i \in [\![c \times 2^q, (c+2) \times 2^q - 1]\!]$. On suppose $P_2(i)$ vérifiée. On opère donc la fusion sur les couples de sous-séquences de L d'indices $[\![i+1, j-1]\!]$ et $[\![j, k]\!]$. Comme décrit précédemment, deux cas se présentent :
      — soit $L_{i+1} < L_j$, auquel cas L_{i+1} est bien placé et il faut ensuite opérer la fusion des sous-séquences $(L_\xi)_{i+2 \leqslant \xi < j}$ et $(L_\xi)_{j \leqslant \xi \leqslant k}$;
      — soit $L_j < L_{i+1}$, auquel cas, pour conserver le caractère trié de chacune des sous-séquences, il est nécessaire de sauvegarder $a \leftarrow L_j$, puis d'opérer un glissement :

$$\forall \xi \in [\![0, j - i - 2]\!], \quad L_{j-\xi} \leftarrow L_{j-\xi-1}$$

      puis d'affecter la valeur sauvegardée à $L_{i+1} \leftarrow a$. Il faut ensuite opérer la fusion des sous-séquences $(L_\xi)_{i+2 \leqslant \xi < j+1}$ et $(L_\xi)_{j+1 \leqslant \xi \leqslant k}$.

   Dans un cas comme dans l'autre, l'élément d'indice $i+1$ est le plus petit de $(L)_{i < \xi \leqslant k}$ et est forcément supérieur ou égal à L_i. Donc $P_2(i+1)$ est vérifiée. Donc P_2 est un invariant de boucle. Comme $k - i$ décroit strictement à chaque appel, la fusion termine et est donc correcte.

   En fin de fusion, la sous-séquence d'indices $[\![c \times 2^q, (c+2) \times 2^q - 1]\!]$ est triée, donc $P(q+1)$ est vraie.

Donc P est un invariant de boucle et le programme est partiellement correct.
Un variant raisonnable est $j-i$ qui décroit strictement à chaque appel vers 0 (cas de sous-séquences triées de longueur 1). Donc le programme termine.
Comme le programme termine et qu'il est partiellement correct, alors il est correct.

4. La complexité en temps de l'algorithme de tri-fusion dépend :
   — du nombre de comparaisons de la fonction `fusion`, soit $n-1=O(n)$ pour 2 séquences contenant n éléments en tout ;
   — du nombre d'appels récursifs de la fonction `fusion` pour réaliser :

$$\begin{aligned} &\left\lfloor \frac{n}{2} \right\rfloor \text{ fusions de 2 éléments} \\ +&\left\lfloor \frac{n}{4} \right\rfloor \text{ fusions de 4 éléments} \\ +&\dots \\ +&\left\lfloor \frac{n}{2^p} \right\rfloor = 1 \text{ fusion de } 2^p \text{ éléments} \end{aligned}$$

   où $p \simeq \log_2(n)$. Il vient alors par somme et au pire :

$$\sum_{i=1}^{\log_2(n)} \left\lfloor \frac{n}{2^i} \right\rfloor 2^i = n \sum_{i=1}^{\log_2(n)} 1 = n\log_2(n) = O\left(\ln(n)\right)$$

   La complexité est donc constante et vaut $O\left(n\ln(n)\right)$. Comme on agît sur place, la complexité en espace est en $O(n)$.

5. On peut convertir le tri-fusion sous forme récursive terminale en économisant la phase de découpe. En supposant dans un premier temps que les séquences à trier sont de longueur $n=2^p$, il s'agît de fusionner successivement les paires de cases adjacentes, puis les quadruplets, etc. Il vient alors :

$$\forall k \in [\![0,p]\!], \quad \forall c \in [\![0,2^{p-k}-1]\!], \quad \text{fusionner } (L)_{c2^{k+1}\leqslant i<(2c+1)2^k} \quad \text{et} \quad (L)_{(2c+1)2^k\leqslant i<(c+1)2^{k+1}}$$

   Le code suivant convient.

```
def tri(L,k=0,c=0):
    p = -ma.floor(1-ma.log(len(L))/ma.log(2))
    if k<=p:
        if c<2**(p-k):
            fusion(L,c*2**(k+1),c*2**(k+1)+2**k,(c+1)*2**(k+1)-1)
            tri(L,k,c+1)
        else:
            tri(L,k+1,0)
```

* *
*