Informatique — MPSI/PCSI, 2024–2025

Correction du TP n° 10

1 Logarithme en base 10

Exercice 10.1

1. Le programme termine systématiquement : si $x < 1$, la boucle `while` n'est pas exécutée. Dans le cas contraire, si n_0 est le plus petit entier tel que $10^{n_0} > x$, $n_0 - n$ est un variant de boucle : on en déduit que la boucle s'arrête nécessairement.
2. Le programme n'est pas correct : si $x < 1$, il retourne systématiquement 0 alors que $\lfloor \log_{10}(x) \rfloor$ est strictement négatif et donc sa partie entière également.
3. On peut être tenter de partir de $n = 0$ et de séparer deux cas : dans le cas ou $x \geq 1$, tant que $10^n \leq x$, ajouter 1 à n, et dans le cas contraire, tant que $10^n > x$, retirer 1.

   Il faut cependant être vigilant au fait que dans le premier cas, la valeur finale de n est la première valeur telle que $10^n > x$ (on a donc $\lfloor \log_{10}(x) \rfloor = n - 1$) et dans le second, elle est la première valeur telle que $10^n \leq x$ (on a donc $\lfloor \log_{10}(x) \rfloor = n$).

   Préconditions $P(x)$: x réel tel que $x > 0$
   Post-conditions $Q(x, n)$: n entier tel que $10^n < x$ et $10^{n+1} > x$.

   On considère le programme modifié suivant.

```
1  def entlog10v2(x):
2      n=0
3      assert x>0
4      if x>=1:
5          while 10**n<=x:
6              n+=1
7          return n-1
8      else:
9          while 10**n>x:
10             n-=1
11         return n
```

À la ligne 4, on a deux cas possible.

Si la condition est vérifiée, on entre dans la boucle commençant à la ligne 5, qui se termine comme vu à la question précédente. La boucle se termine lorsque $10^n > x$, et puisqu'elle ne s'était pas terminée lors du passage précédent, on a bien $10^{n-1} \leq x$.

Dans le cas contraire, on entre dans la boucle commençant à la ligne 9.Si n_0 désigne le plus grand entier relatif tel que $10^{n_0} \leq x$, $n - n_0$ est un invariant de boucle et la boucle termine. La condition d'arrêt nous garantie qu'à la sortie de la boucle, $10^n \leq x$. De plus, la boucle ne s'étant pas arrêté au passage précédent, $10^{n+1} > x$ et la postcondition est vérifié.

Dans chacun des deux cas, la valeur retournée est bien $\lfloor \log_{10}(x) \rfloor = n$.

2 Valeur d'un polynôme en un réel

Exercice 10.2

1. Le programme termine systématiquement : si la liste est vide, il n'entre pas dans la boucle `for`. Sinon, cette boucle étant exécutée un nombre prédéfini de fois.
   En revanche, si la liste p est vide, il retourne un résultat qui n'a pas d'interprétation.
2. **Précondition :** P liste non vide
   Postcondition : s réel tel que $s = \sum_{i=0}^{n-1} P[i]x^i$

   Pour vérifier que la postcondition est remplie, on vérifie que l'invariant $s = \sum_{i=0}^{k} P[i]x^i$ est vérifié à la fin de chaque boucle.

   Avant l'entrée dans la boucle, s est égal à 0 et l'invariant est vérifié et prenant par convention le fait qu'une somme vide est égale à 0

   Supposons que l'invariant soit varié à l'indice k. Au passage suivant, s prend la valeur

   $$s + P[k+1] \times x^{k+1} = \sum_{i=0}^{k} P[i]x^i + p[k+1]x^{k+1} = \sum_{i=0}^{k+1} P[i]x^i$$

   donc l'invariant est préservé.

   Enfin, lors du dernier passage dans la boucle, l'indice vaut $n-1$. L'invariant de fin de boucle nous permet d'affirmer que $vk = \sum_{i=0}^{n-1} P[i]x^i$ et la postcondition est vérifiée.

Exercice 10.3

1. On peut écrire le programme suivant.

```
1  def horner_rec(P,x):
2      """Algorithme de horner version récursive
3
4      Paramètre :
5          P (list) : coefficients du polynôme
6          x (float) : valeur ou on évalue le polynome
7
8      Resultat (float) :
9          P(x)"""
10     if len(L)==1:
11         return L[0]
12     else:
13         return L[0]+x*horner(L[1:],x)
```

2. On peut reprendre la précondition de l'exercice précédent. Pour la postcondition, si s est le résultat retourné, on doit avoir $s = \sum_{i=0}^{n-1} P[i]x^i$.

   Montrons que l'algorithme termine s'il prend en entrée une liste non vide.

   Soit n la longueur de la liste : si $n = 1$, l'algorithme termine. Sinon, il est appelé avec une liste de longueur $n-1$. La suite des longueurs des listes passées en paramètres est donc strictement décroissante et positive, donc l'algorithme termine.

   Montrer que la postcondition est vérifiée.

   Si $n = 1$, le résultat retourné est a_0 qui est bien le résultat attendu.
   Sinon, on montre que la si la postcondition se conserve à chaque appel.
   Le résultat retourné est $a_0 + x * horner([a_1, \ldots, a_{n-1}], x)$. Si $horner([a_1, \ldots, a_{n-1}], x)$ retourne $\sum_{i=1}^{n-1} P[i]x^{i-1}$, ce résultat est donc

   $$a_0 + x \times \sum_{i=1}^{n-1} P[i]x^{i-1} = a_0 + \sum_{i=1}^{n-1} P[i]x^i = \sum_{i=0}^{n-1} P[i]x^i$$

et la postcondition reste vérifiée.

3. Soit n le degré du polynôme.

   En utilisant la méthode naïve, à chaque passage dans la boucle, on fait une addition, $k-1$ multiplications pour calculer x^k et une de plus pour le multiplier par $p[k]$. On fait donc n additions et $1+2+\ldots+n=\frac{n(n+1)}{2}$ multiplications, d'où une complexité en $O(n^2)$.

   Lors de l'exécution de l'algorithme de Horner, on fait à chaque passage une multiplication et une addition, soit n multiplications et n additions au total, d'où une complexité en $O(n)$.

3 Le problème du drapeau hollandais

Exercice 10.4

1. On peut proposer le programme suivant.

```
1  def drapeau(L,p):
2      n=len(L)
3      i,j,k=0,0,n # Valeurs initiales de i, j et k
4      while j<k:
5          elt=L[j] # Compare le premier element non teste à elt
6          if elt<p: # S'il est plus petit, on classe dans la deuxième partie
7              L[i]=elt
8              L[j]=p
9              j+=1
10             i+=1
11         elif elt==p: # S'il est egal, deuxieme partie
12             j+=1
13         else:
14             L[j]=L[k-1] # Sinon, partie finale
15             L[k-1]=elt
16             k-=1
17     return L
```

2. On prend comme variant de boucle $k-j$. La boucle ne s'exécute que tant que ce variant est strictement positif. De plus, à chaque passage dans la boucle, soit j augmente de 1, soit k diminue de 1, donc $k-j$ diminue de 1.
3. **Post-condition :** il existe des entiers i et j tels que :

   — $0 \leq i \leq j \leq n-1$
   — Pour tout entier l compris entre 0 et $i-1$, $L[l] < p$.
   — Pour tout entier l compris entre i et $j-1$, $L[l] = p$.
   — Pour tout entier l compris entre j et $n-1$, $L[l] > p$.

   Pour vérifier que cette post-condition est vérifiée en fin de programme, on vérifie que l'invariant suivant est vérifié à chaque fin de boucle.

   Invariant :il existe des entiers i, j et k tels que :

   — $0 \leq i \leq j \leq k \leq n-1$
   — Pour tout entier l compris entre 0 et $i-1$, $L[l] < p$.
   — Pour tout entier l compris entre i et $j-1$, $L[l] = p$.
   — Pour tout entier l compris entre k et $n-1$, $L[l] > p$.

   Avant le premier passage dans la boucle, on a $i=0$, $j=0$ et $k=n-1$, donc toutes les conditions sont vérifiées.

   Supposons que toutes les conditions sont vérifiées en début de boucle.

   Si $L[j] < p$ à la ligne 6, $L[i]$ se voit affecter la valeur de $L[j]$, et $L[j]$ se voit affecter la valeur précédente de $L[i]$, qui était égale à p d'après l'invariant de boucle. De plus, i et j sont augmentés de 1, donc les conditions 2 et 3 de l'invariant restent vérifiées. Comme $k-j>0$ au début de la boucle, on a de plus toujours $0 \leq i \leq j \leq k \leq n-1$.

Sinon, si $L[j] = p$, aucun élément n'est modifié et j est augmenté de 1, donc la troisième condition de l'invariant reste vérifiée pour la nouvelle valeur de j. Comme $k - j > 0$ au début de la boucle, on a toujours $0 \leq i \leq j \leq k \leq n - 1$.

Enfin, si $L[j] > p$, $L[k-1]$ se voit affecter la valeur de $L[j]$ qui était strictement supérieure à p, $L[j]$ se voit affecter la valeur précédente de $L[k-1]$ et k diminue de 1, donc la quatrième condition de l'invariant reste vérifiée pour la nouvelle valeur de j. Comme $k - j > 0$ au début de la boucle, on a toujours $0 \leq i \leq j \leq k \leq n - 1$,

L'invariant reste donc vérifié en fin de boucle.

Lorsque le programme s'arrête, on a $k - j = 0$ d'après la condition d'arrêt donc $j = k$. Les deux dernières conditions de l'invariant deviennent identiques, et la post-condition est vérifiée.

4. Soit n la taille du tableau.

   Tous les éléments du tableau sont testés à tour de rôle, on effectue donc n tests.

   Le cas le plus favorable est celui où on ne fait aucun échange, par exemple si tous les éléments sont "blancs".

   Le cas le plus défavorable est celui où on fait après chaque test deux échanges, par exemple si tous les éléments sont "rouge" : on effectue dans ce cas $2n$ échanges.

   Remarque : la complexité en moyenne est difficile à évaluer car elle dépend de la répartition initiale des valeurs de la liste. Si on fait l'hypothèse que chaque élément, au moment où on l'évalue, à une chance sur trois d'être dans chacune des trois catégories, on a deux cas où on doit réaliser deux échanges, et un cas où on n'en réalise pas : le nombre moyen d'échanges dans ce cas est donc environ égal à $2 \times \frac{2n}{3} = \frac{4n}{3}$.

4 La fin des haricots

Exercice 10.5

1. Le programme peut prendre entrée deux entiers n et b représentant les nombres initiaux respectifs de haricots noirs et blancs.

   Il faut veiller à ce qu'il y ait un nombre positif de haricots noirs et blancs, et au moins deux haricots au total. On peut donc proposer les spécifications suivantes.

   Préconditions $n \geq 0$ et $b \geq 0$ et $n + b > 1$.

2. Le code suivant convient :

```
1  def gries(n,b):
2      """Determine la couleur du haricot restant
3      Paramètres
4          n (int) : nombre de haricots noirs
5          b(int) : nombre de haricots blancs
6
7      Retour (int) : 0 si le haricot est blanc, 1 sinon
8      """
9      assert n>=0
10     assert b>=0
11     tot= n+b # Nombre de haricots total
12     assert tot>1 # Verifie qu'on a au moins deux haricots au depart
13     while tot>1:
14         n1=random.randrange(tot) # Tirage premier haricot
15         if n1<b: # cas haricot blanc
16             c1=1
17             b-=1
18             tot-=1
19         else: # cas haricot noir
20             c1=0
21             n-=1
22             tot-=1
```

```
23          n2=random.randrange(tot)# Tirage deuxieme haricot
24          if n2<b: # cas haricot blanc
25              c2=1
26              b-=1
27          else: # cas haricot noir
28              c2=0
29              n-=1
30          if c1==c2: # teste si les haricots sont de la meme couleur
31              n+=1
32          else:
33              b+=1
34      return b # Retourne le nombre final de haricots blancs (0 ou 1)
```

3. On peut utiliser comme variant de boucle la variable interne *tot* égal à $n + b$: *tot* ne peut être inférieur ou égale à 2 d'après la condition d'arrêt, et elle diminue de 1 à chaque passage dans la boucle.
4. On peut remarquer que le dernier haricot est systématiquement noir si le nombre de haricots blancs initial est impair, noir sinon. Notons que ce résultat est déterministe, et ne dépend pas du nombre de haricots noirs de départ !
5. On peut être tenter d'utiliser l'invariant de boucle suivant : $b \equiv b_{init}[2]$. Pour conclure cependant, il faut également vérifier que $b = 0$ ou $b = 1$ à la sortie de la boucle.

   On utilise donc l'invariant complet suivant : $b \equiv b_{init}[2]$ et $n \geq 0$ et $b \geq 0$.

   Cet invariant est vrai avant la premier passage dans la boucle du fait des pré-conditions.

   Supposons que l'invariant soit vrai en début de boucle. Le premier haricot tiré ne peut être blanc (ligne 15) que si $b > 0$. Dans ce cas, b diminue de 1 mais $b \geq 0$ reste vérifié.

   Dans le cas contraire, on a nécessairement $b < tot$, donc $n > 0$. Dans ce cas, n diminue de 1 mais $n \geq 0$ reste vérifié.

   On peut faire le même raisonnement pour le tirage du second haricot (ligne 24).

   Enfin, lors du test de l'égalité des couleurs (ligne 30) :

   — Si on a tiré deux haricots blancs, on remet un noir : le nombre de haricots blancs diminue de 2.
   — Si on a tiré deux haricots noirs, on remet un noir : le nombre de haricots blancs est inchangé.
   — Si on a tiré deux haricots de couleur différente, on remet le noir : le nombre de haricots noirs est inchangé.

   Le nombre de haricots blanc de change pas de parité et $b \equiv b_{init}[2]$ reste vrai.

   Lors de la sortie de la boucle, on a $tot = n + b = 1$ d'après la condition d'arrêt, $n \geq 0$ et $b \geq 0$. On en déduit que $b = 0$ ou $b = 1$.

   Par ailleurs le fait que b et b_{init} aient la même parité garantit que $b = 0$ si b_{init} était pair et $b = 1$ sinon : le haricot restant est blanc si et seulement si le nombre initial de haricots blancs était impair.

* *
*