Interrogation du 15 octobre 2024

durée : 1 heure 30

Pour chacun des algorithmes, on justifiera avec soin :
- que l'algorithme termine (si une boucle « tant que » est utilisée).
- que l'algorithme renvoie bien le résultat demandé.

Une réponse non justifiée sera notée sur les trois quarts des points.

On peut utiliser les opérateurs `//` et `%` sur les entiers. On évitera d'utiliser des fonctionnalités trop avancées de python, surtout celles non vues en TD : on rédigera en pseudo-code avec des techniques élémentaires. En cas de doute, poser simplement la question.

Pour les boucles « pour », on adoptera les conventions suivantes. Si $a, b \in \mathbb{Z}$, « pour i allant de a à b » signifie « pour i parcourant en croissant l'intervalle $[\![a, b]\!]$ ». Lorsque $b < a$, cet intervalle est vide, donc aucune des instructions dans la boucle « pour » n'est effectuée (ce sera par exemple le cas si on écrit « pour i allant de 1 à n », avec $n = 0$). On n'utilisera pas d'autre type de boucles « pour » (décroissantes, saut d'indice etc) : dans ces situations, on utilisera l'instruction « tant que ».

Dans cette interrogation, « complexité » signifie « complexité en temps », on ne demandera pas de complexité en espace. Les complexités sont entendues dans le pire des cas.

Exercice 1. *Écrire une fonction* `expo` *qui prend en entrée un réel a et un entier naturel n et qui renvoie a^n, en n'utilisant que des sommes et des produits (et pas l'exponentiation).*

On donnera une fonction récursive et aussi itérative, et on prouvera à chaque fois la correction de l'algorithme utilisé.

Ne pas chercher à faire de l'exponentiation rapide, c'est l'objet d'un exercice ultérieur. Utiliser l'algorithme naïf.

Exercice 2. *Écrire une fonction* `evalPoly` *qui prend en entrée un polynôme P (donné par la liste de ses coefficients, non vide par hypothèse) ainsi qu'un réel a, et qui retourne la valeur $P(a)$. Le polynôme sera entré sous forme de liste de coefficients, en commençant par le coefficient constant. Par exemple, si le polynôme est $3X^5 + X^4 + 2X - 1$, la liste des coefficients sera* `[-1,2,0,0,1,3]`.

Consignes : écrire un algorithme naïf, sans chercher à optimiser. On peut utiliser la fonction `expo` *de la question précédente.*

Prouver la correction de l'algorithme proposé. Quelle est la complexité de l'algorithme proposé, en fonction de la longueur de la liste des coefficients du polynôme ?

Dans la suite, on améliore progressivement cet algorithme d'évaluation polynomiale.

Exercice 3. *Que fait l'algorithme suivant ($a \in \mathbb{R}$ et $n \in \mathbb{N}$) et quelle est sa complexité en fonction de n ?*

```python
def f(a,n):
  if n == 0:
    return 1
  if n % 2 == 0 :
    return f(a*a, n//2)
  return f(a*a, n//2) * a
```

Exercice 4. *Écrire une fonction non récursive qui fait la même chose que la précédente, avec la même complexité en n. (Il s'agit donc de remplacer le mécanisme de récursivité par une boucle « tant que ».)*

Exercice 5. *En utilisant la fonction du deuxième exercice, proposer une amélioration* `evalPoly2` *de la fonction* `evalPoly` *pour évaluer un polynôme. Les entrées et sorties sont les mêmes qu'à l'exercice 1, la complexité doit être meilleure. Quelle est la complexité de votre algorithme ?*

Dans l'exercice suivant, on améliore encore l'algorithme.

Exercice 6. *L'algorithme de* ***Hörner*** *pour évaluer des polynômes consiste à ne pas recalculer les puissances à chaque fois depuis le début, mais à garder le mémoire les puissances successives pour calculer les suivantes. Typiquement, si l'on veut évaluer* $X^4 + 2X^3$ *au réel* a*, on va d'abord calculer* a^3 *puis multiplier par deux pour évaluer le premier monôme, mais il serait dommage de recommencer le calcul de* a^4 *puisqu'on a déjà* a^3 *: il suffit de multiplier par* a*, ce qui ne fait qu'une opération. Si* $P = \sum_{k=0}^{d} c_k X^k$ *et* $a \in \mathbb{R}$*, l'algorithme de Hörner consiste à écrire :*

$$P(a) = (((c_d \times a + c_{d-1}) \times a + \cdots) \times a + c_2) \times a + c_1) \times a + c_0$$

et à effectuer le calcul en commençant par la parenthèse la plus « profonde ». Exemple : pour évaluer $X^3 + 3X^2 + X + 5$ *en* a*, on calcule* $a + 3$*, on multiplie le résultat par* a *et on ajoute* 1 *au résultat, ensuite on multiplie tout ceci par* a *et on rajoute* 5*.*

1. *Écrire une fonction* `evalPolyHorner` *qui implémente cet algorithme. (Toujours avec les mêmes entrées et sorties qu'à l'exercice 2.)*
2. *Quelle est la complexité de cet algorithme ? (Toujours en fonction du degré du polynôme, c'est-à-dire de la longueur de la liste de ses coefficients.*

Attention, corrigé en cours de rédaction

Correction de l'exercice 1 Algorithme récursif :

```
def expo(a,n):
  if n == 0:
    return 1
  return a * expo(a,n-1)
```

Terminaison : la variable n passée en argument dans l'appel récursif diminue de un à chaque appel, en temps fini elle vaut 0, étape à laquelle les appels s'arrêtent.
Terminaison, version plus détaillée : on prouve que l'appel à `exp(a,n)` effectue exactement $n+1$ appels en tout. Preuve : par récurrence : `exp(a,0)` effectue exactement un appel, le premier. Ensuite, l'appel de `exp(a,n+1)` effectue un autre appel à `exp(a,n)`, ce qui fait $1 + (n + 1) = n + 2$ appels d'après l'hypothèse de récurrence.
Validité : On fixe a. Prouvons par récurrence sur $n \in \mathbb{N}$ que `expo(a,n)` retourne bien a^n. Par définition de la fonction, c'est vrai pour $n = 0$. Montrons l'hérédité. Soit $n \in \mathbb{N}$, supposons que `expo(a,n)` retourne bien a^n et montrons que `expo(a,n+1)` retourne bien a^{n+1}. Par définition de la fonction, comme $n + 1 \neq 0$, `expo(a,n+1)` retourne `a * expo(a,n)`. D'après l'hypothèse de récurrence, ceci est égal à $a \times a^n = a^{n+1}$.

Algorithme itératif :

```
def expo(a,n):
  r = 1
  for k in range(1,n+1):
    r = r * a
  return r
```

Terminaison : la boucle *for* tourne exactement n fois.
Validité : Pour $n = 0$, la fonction renvoie bien 1. Montrons la correction pour $n \geq 1$. Pour k entre 1 et n, notons r_k la valeur de la variable `r` à la fin de la boucle d'indice k.
Montrons par récurrence sur k que pour tout k entre 1 et n, $r_k = a^k$.
Pour $k = 1$ c'est vrai.
Soit $1 \leq k \leq n - 1$, supposons $r_k = a^k$ et montrons $r_{k+1} = a^{k+1}$. Dans la boucle d'indice $k + 1$, l'affectation se traduit par $r_{k+1} = ar_k$. Par hypothèse de récurrence, $r_k = a^k$, donc on a bien $r_k = a^{k+1}$.
Fin de la preuve de validité : en sortie de boucle, c'est-à-dire à la fin de la boucle d'indice n, on a bien $r_n = a^n$.

Correction de l'exercice 2

```
def evalPoly(L,a):
  d=len(L)-1
  r=0
  for k in range(d+1):
    r += L[k] * expo(a,k)
  return r
```

Terminaison : on a une boucle for qui tourne un nombre prédéterminé de fois, et qui contient une fonction `expo` qui termine.

Validité : Notons r_k la valeur de r à la fin de la boucle d'indice k. Alors, on a $r_k = \sum_{j=0}^{k} c_j a^j$. Preuve : récurrence sur k.
En fin de boucle, la variable `r` vaut donc $r_d = \sum_{j=0}^{d} c_j a^j = P(a)$.

Complexité : la fonction `expo(a,k)` est de complexité $O(k)$, donc $O(n)$ en majorant. La fonction `evalPoly` est de complexité $O(n^2)$ avec n la longueur de la liste.

Correction de l'exercice 3 C'est exponentiation rapide, avec une implémentation légèrement différente de celle vue en cours.

Terminaison : Pour tout entier n, la suite d'entiers (u_k) définie par $u_0 = n$ et la relation de récurrence $u_{k+1} = u_k//2$ est une suite qui stationne à zéro. L'algorithme termine donc.

Terminaison, version plus précise : si $u_k \geq 2$, alors u_{k+1} a exactement un chiffre de moins que u_k en base deux. Sinon, c'est-à-dire si $u_k < 2$, alors $u_{k+1} = 0$ et la récursivité s'arrête à l'étape d'après. Si ℓ est le nombre de chiffres en base deux de $n \neq 0$, alors `f(a,n)` provoque exactement ℓ autres appels récursifs à la fonction. Par exemple `f(a,10)` provoque quatre autres appels.

Validité : montrons que la fonction `f(a,n)` retourne a^n. Pour $n \in \mathbb{N}$, on note $P(n)$ l'assertion : « pour tout[1] réel a, la fonction `f(a,n)` retourne a^n. »Montrons maintenant par récurrence forte l'assertion $\forall n \in \mathbb{N}, P(n)$.

Initialisation : $P(0)$ est vraie par définition de la fonction.

Hérédité : soit $n \in \mathbb{N}$, supposons $P(k)$ vraie pour tout $j \leq n$ et montrons $P(n+1)$. Soit donc $a \in \mathbb{R}$. Comme $n+1 > 0$, la première condition est sautée. Si $n+1$ est pair, la quantité $(n+1)//2$ vaut donc $(n+1)/2$. La fonction retourne $f(a^2, (n+1)/2)$. Comme $(n+1)/2 \leq n$, l'hypothèse de récurrence forte implique que $f(a^2, (n+1)/2) = (a^2)^{(n+1)/2} = a^{n+1}$. Si par contre $n+1$ est impair, alors $(n+1)//2 = n/2 \leq n$, et toujours par hypothèse de récurrence forte, on a $a \times f(a^2, n/2) = a \times (a^2)^{n/2} = a \times a^n = a^{n+1}$.

Complexité : Étant donné un exposant d'entrée n, notons p le nombre d'appels à la fonction qui vont être effectués, y compris le premier appel. À l'intérieur de la fonction, hors récursivité, la complexité est en $O(1)$ (entre une et deux comparaisons, moins de deux produits, moins d'une division). La complexité de l'appel total, récursivité comprise, est donc en $O(p)$. Comme $p = O(\log_2 n)$, on en déduit que la complexité totale est en $O(\log_2 n)$.

Remarque : comme dans la preuve précise de terminaison, on peut calculer p en fonction de n : c'est un de plus que le nombre de chiffres en base deux de n, si $n > 0$. (Par exemple, si $n = 2^k$, alors $p = k+2$.)

Correction de l'exercice 4

Proposition d'algorithme itératif d'exponentiation plus rapide que l'algorithme naïf :

```python
def f(a,n):
  if n == 0:
    return 1
  r=1
  while n >= 1:
    k=1
    b=a
    while 2 * k <= n:
      k = 2 * k
      b = b * b
    r = r * b
    n = n - k
  return r
```

Preuve de terminaison : la boucle la plus profonde termine car la suite 2^k est non bornée. Après l'arrêt de la boucle while la plus profonde, l'instruction `n = n-k` fait décroitre strictement n car $k \geq 1$. Ceci montre que la boucle extérieure s'arrête en temps fini.

Terminaison, version plus précise : le nombre de chiffres de n en base deux décroit strictement à chaque

1. attention ici il est important de mettre le pour tout a dans l'assertion à prouver, car on va appliquer l'hypothèse de récurrence sur une autre valeur de a.

passage dans la boucle extérieure.

Preuve de validité : notons n_0 et r_0 les valeurs initiales de n et r, avant la boucle while. Notons n_i et r_i les valeurs de n et r à la fin du i-ème passage dans la boucle while extérieure. Alors, on a toujours, en fin de boucle extérieure : $a^{n_0} = r_i \times a^{n_i}$. (Récurrence sur i.) En fin de boucle, on a $n_i = 0$ et donc $r_i = a^{n_0}$, qui est la valeur retournée.

Plus malin, avec une seule boucle while, en mettant en mémoire les exposants à rajouter dans le cas impair :

```
def f(a,n):
  if n == 0:
    return 1
  b = a
  c = 1
  while n > 1:
    if n % 2 == 1:
      n = n-1
      c = c * b
    else:
      n = n / 2
      b = b * b
  return b * c
```

Exercice : comprendre le fonctionnement de cette fonction, prouver la terminaison, la correction, comparer la complexité avec la proposition antérieure.

Correction de l'exercice 5 On fait tout simplement , en notant `exporapide` une fonction d'exponentiation rapide comme celle donnée plus haut :

```
def evalPoly(L,a):
  d=len(L)-1
  r=0
  for k in range(d+1):
    r += L[k] * exporapide(a,k)
  return r
```

Estimation de la complexité :
On note d le degré du polynôme en entrée et on calcule la complexité en fonction de d. À l'intérieur de la boucle, la complexité de `exporapide(a,k)` est en $O(\log k)$. Comme $k \leq d$, ceci est en $O(\log d)$.

Chacun des $d+1$ appels est donc en $O(\log d)$. La complexité totale est en $O(d \log d)$.

Correction de l'exercice 6

```
def evalPolyHorner(L,a):
  d = len(L)-1
  r = L[d]
  k = d
  while k >= 1:
    r = r * a + L[k-1]
    k = k-1
  return r
```

Preuve de terminaison : la variable k décroit de 1, elle finit donc par être nulle, moment où la boucle s'arrête. On pourrait aussi réécrire l'algorithme avec une boucle « for », d'ailleurs.

Preuve de validité : notons $P = \sum_{k=0}^{d} c_k X^k$ le polynôme. Pour k entre d et zéro, notons r_k la valeur de la variable r juste avant le test d'entrée de la boucle d'indice k. On a $r_d = c_d$, $r_{d-1} = ac_d + c_{d-1}$, $r_{d-2} = a^2 c_d + ac_{d-1} + c_{d-2}$. Plus généralement, pour $k \in [\![0, D]\!]$, notons $A(k)$ l'assertion «$r_k = \sum_{j=k}^{d} c_j a^{j-k}$».

Montrons par récurrence descendante sur $k \in [\![d, 0]\!]$ que $k \in [\![0, D]\!], A(k)$.
Initialisation : $A(d)$ est vraie.
Soit $k \in [\![1, d]\!]$. Supposons $A(k)$. Montrons $A(k-1)$.
On a donc $r_k = \sum_{j=k}^{d} c_j a^{j-k}$ et lors du passage dans la boucle, on a :

$$r_{k-1} = ar_k + c_{k-1} = a\sum_{j=k}^{d} c_j a^{j-k} + c_{k-1} = \sum_{j=k}^{d} c_k a^{j-(k-1)} + c_{k-1} = \sum_{j=k-1}^{d} c_k a^{j-(k-1)}.$$

On a donc $r_0 = \sum_{j=0}^{d} c_j a^j = P(a)$, puis k passe à -1, et au test suivant la boucle s'arrête. L'algorithme retourne alors la bonne valeur.