Licence de Mathématiques et Informatique — Université de Perpignan Via Domitia
Semestre 2

Algorithmique

Correction de l'examen de première session 2014/2015

Durée : 2 heures. Aucun document autorisé.

Modalités : Répondre uniquement dans les cadres prévus à cet effet. Le barême est indiqué à droite de chaque question. La qualité de la rédaction sera prise en compte dans la notation.

Exercice 1. (23 points) **/23**

```
1  declare
.    t[25] : tableau de caractères = [a,n,t,i,c,o,n,s,t,i,t,u,t,i,o,n,n,e,l,l,e,m,e,n,t]
.    n : entier = 0
.    c : caractère
5    i : entier = 0 // itérateur tantque
.  debut
.    lire(c)
.    tant que (i < 25) faire
.      si t[i] == c alors
10       n = n + 1
.      fin si
.      i = i + 1
.    fin tant que
.    si n > 2 alors
15     afficher(n)
.    sinon
.      afficher(-1)
.    fin si
.  fin
```

1. Qu'affiche cet algorithme si l'utilisateur entre la valeur `'t'` ? **/0.5**

   `5 (anticonstitutionnellement contient 5 fois la lettre t)`

2. Qu'affiche cet algorithme si l'utilisateur entre la valeur `'a'` ? **/0.5**

   `-1 (anticonstitutionnellement contient 1 seule fois la lettre a)`

3. Qu'affiche cet algorithme si l'utilisateur entre la valeur `'l'` ? **/0.5**

   `-1 (anticonstitutionnellement contient 2 fois la lettre l)`

4. Qu'affiche cet algorithme si l'utilisateur entre la valeur `'z'` ? **/0.5**

   `-1 (anticonstitutionnellement ne contient pas la lettre z)`

5. Détailler quel traitement effectue cet algorithme. Préciser le rôle de chaque variable. **/1**

```
Cet algo parcourt la chaine de caractère t et compte le nombre n
d'occurrences de la lettre c entrée par l'utilisateur.
Ce nombre est affiché si la lettre apparait trois fois au moins.
Sinon (la lettre est présente 0, 1 ou 2 fois) la valeur arbitraire -1
est affichée.
L'itérateur i prend les valeurs de 0 à 24 pour atteindre chacune des
25 lettres de t.
```

6. Écrire une autre version de cet algorithme sans boucle `tant que`. **/1**

```
declare
  t[25] : tableau de caractères = [a,n,t,i,c,o,n,s,t,i,t,u,t,i,o,n,n,e,l,l,e,m,e,n,t]
  n : entier = 0
  c : caractère
  i : entier // itérateur pour
debut
  lire(c)
  pour i de 0 à 24 faire
    si t[i] == c alors
      n = n + 1
    fin si
  fin pour
  si n > 2 alors
    afficher(n)
  sinon
    afficher(-1)
  fin si
fin
```

7. On note `f` la fonction qui effectue **le traitement** précédent (sans les entrées/sorties) à tout tableau 1D de `n` caractères.

```
declare
  c : caractère
  res : ...
  ...
debut
  lire(c)
  ... // appel de f
  afficher(res)
fin
```

Écrire l'en-tête de `f` pour que le code incomplet ci-dessus produise le traitement de l'algorithme initial. **/1**

```
fonction f(c: caractère, tab: tableau de caractères, n: entier)
  retourne entier
```

8. Compléter le code précédent avec l'appel de `f`. **/1**

```
declare
  c : caractère
  res : entier
  t[25] : tableau de caractères = [a,n,t,i,c,o,n,s,t,i,t,u,t,i,o,n,n,e,l,l,e,m,e,n,t]
debut
  lire(c)
  res = f(c, t, 25)
  afficher(res)
fin
```

9. Écrire le corps de `f`. **/1**

```
fonction f(c: caractère, tab: tableau de caractères, n: entier)
  retourne entier
  res : entier = -1
  nb : entier = 0
  i : entier // itérateur pour
debut
  pour i de 0 à n-1 faire
    si tab[i] == c alors
      nb = nb + 1
    fin si
  fin pour
  si nb > 2 alors
    res = nb
  fin si
  retourne res
fin fonction
```

10. Justifier que cette fonction termine. **/2**

```
La version avec la boucle pour est plus simple à analyser.
Le nombre nb est initialisé à 0 avant la première itération de la
boucle pour.
Cette boucle effectue n itérations exactement. A chaque itération :
. l'accès à tab[i] s'effectue correctement car tab est de longueur n
(et ses valeurs sont indicées de 0 à n-1);
. nb est éventuellement incrémenté.
La boucle termine donc avec nb ≤ n.
Selon cette valeur, res vaut sa valeur initiale -1 ou nb si nb > 2.
Cette valeur de res est retournée par la fonction qui termine ainsi.
```

11. Quel est un bon paramètre de complexité de cet algorithme ? Justifier. **/2**

L'algorithme effectue des comparaisons (l.9) et des additions (l.10 et l.12 pour la version tant que).
Le nombre de ces opérations dépend de la longueur n du tableau de caractères. Cette longueur n est le paramètre de complexité si on compte le nombre de comparaisons et/ou d'additions.

12. Expliciter la fonction de complexité correspondante. On pourra donner un résultat sous la forme d'encadrement ou simplement un majorant. **/2**

l.8. : la boucle pour effectue exactement n itérations.
l.9 : une comparaison à chaque itération de la boucle pour ;
l.10 : au plus une addition à chaque itération de la boucle pour, soit $0 \leq$ nb add $\leq$ n ;
l.12 : la mise à jour (explicite ou transparente) de l'indice de boucle ajoute (si besoin) autant d'additions que de nombre d'itérations.
L'éxecution de la boucle prend donc entre n+0+n=2n et n+n+n=3n opérations. S'ajoute la dernière comparaison (l.14).
Donc $2n+1 \leq C(n) \leq 3n+1$. Le membre de droite représente la complexité dans le pire cas de cet algorithme.

13. Quelle est la nature de la complexité asymptotique ? **/1**

$2n+1 <= C(n) <= 3n+1$, donc la complexité est asymptotiquement linéaire, $C(n) = \theta(n)$.

14. Écrire une spécification (entrées, sorties, traitement) de cet algorithme afin de prouver sa correction dans la questions suivante. **/1**

Entrées : une lettre c, un tableau t de n caractères t.
Sortie : un entier égal à -1 ou supérieur strictement à 2.
Traitement : cet algo retourne la valeur arbitraire -1 si la lettre c est présente 0, 1 ou 2 fois dans un tableau t.
Sinon il retourne le nombre d'occurrences (>2) de la lettre c dans t.

15. Proposer un invariant de la boucle `pour`. **/2**

L'algorithme se décompose en deux parties :
. le décompte du nombre d'occurrences nb de c dans t de longueur n, réalisé par la boucle (pour ou tant que)
. le bloc si/alors/sinon retourne une valeur res (=-1 ou > 2) selon ce nombre nb.
Le début du traitement est le décompte d'occurrences nb.
L'invariant de la boucle pour est :
Avant l'itération i, nb égale le nombre d'occurrences de c dans le sous-tableau t[0, i-1].

16. Prouver cet invariant. **/3**

```
L'invariant de la boucle pour est :
Avant l'itération i, nb égale le nombre d'occurrences de c dans le
sous-tableau t[0, i-1].
. Initialisation : avant l'itération 0, nb est initailsé à 0. Le
sous-tableau t[0,-1] est vide et contient donc 0 occurrences de c.
. Conservation : Supposons l'invariant vrai avant l'itération i. Cette
itération compare t[i] et c. Si t[i]==c, nb est incrémenté de 1 et
vaut donc le nombre d'occurrences de c dans t[0,i-1] plus cette
nouvelle occurrence. Sinon, il est inchangé est vaut aussi le nombre
d'occurrences de c dans t[0,i].
nb compte bien le nombre d'occurrences de c dans t[0,i] avant
l'itération i+1.
. Terminaison : i==n est la dernière valeur de i avant de terminer
les itérations de la boucle pour. nb vaut donc le nombre
d'occurrences de c dans t[0,n-1]. t est un tableau de longueur n
indicé de 0 à n-1. Donc nb compte bien le nombre d'occurrences de c
dans tout le tableau t.
```

17. Écrire une fonction récursive `g_rec` (en-tête et corps) ainsi que l'appel qui réalise le traitement de la boucle de l'algorithme. Expliquer le principe qui permet la récursion. **/3**

```
Le décompte d'occurrences est récursif si on considère la comparaison
de la première valeur de t et le décompte du nombre d'occurrences du
sous-tableau droit restant. Pour cela, il faut préciser les indices
gauche g et droit d du sous-tableau t[g,d].
L'arrêt de la récursion s'effectue lorsque le sous-tableau droit est
vide, cad pour g>d.

fonction g_rec(c, tab, g, d) retourne entier
debut
  si g > d alors                       // tab est vide
    retourne 0                           // zéro occurrence
  sinon
    si tab[g] == c alors               // c est rencontré
      retourne 1 + g_rec(c, t, g+1, d) // une de plus donc !
    sinon                              // c'est pas c !
      retourne g_rec(c, t, g+1, d)     // rien de plus.
    finsi
  fin si
fin fonction

Appel pour t[25] : tableau de caractères = [a,n,t, ...
declare
  ...
debut
  ...
  nb = g_rec(c, t, 0, 24)
  ...
fin
```

Exercice 2. (12 points) **/12**

On souhaite évaluer un polynôme `p` de degré `n` à coefficients flottants en une valeur `x` flottante. On suppose que $p(x) = \sum_{k=0}^{n} p_k x^k$. On suppose aussi qu'**il n'existe pas** de fonction `**` qui calcule `x**k`=x^k pour `x` et `k`, respectivement flottant et entier.

1. Quelle structure de données utiliser pour représenter le polynôme `p` ? **/1**

```
Un tableau p de longueur n+1 à valeurs flottantes permet de conserver
(stocker)  les n+1 coefficients  p_0, p_1, ..., p_n.
```

2. Écrire sous la forme d'une fonction, un algorithme itératif de complexité au pire quadratique qui calcule l'évaluation de `p` en `x` ? **/1**

```
Il s'agit de l'algorithme naif vu en cours.
fonction eval_poly(n: entier, p[n]: tableau de flottants, x: flottant)
  retourne flottant
  res : flottant = 0.0  // car accumulation de sommes dans res
  y   : flottant = 1.0  // car accumulation de produits dans y
  i, k : entier         // itérateur pour
debut
  pour k de 0 à n faire   // pour chaque terme p[k]*x**k du polynome
    pour i de 0 à k faire // calcul de x**k stocké dans y
      y = y * x
    fin pour
    res = res + p[k] * y  // accumulation k-ième terme du polynome
    y = 1.0               // maj avant prochain calcul de y
  fin pour
  retourne res
fin fonction
```

3. Justifier le caractère au pire quadratique de cette solution. **/2**

```
La complexité en temps mesure le nombre d'opérations arithmétiques
+, *. Le nombre total d'opérations dépend du degré n du polynôme.
Cet algo présente deux boucles imbriquées.
. La boucle intérieure calcule x**k en k multiplications.
. La boucle extérieure fait varier k entre 0 et n.
A chaque itération, elle effectue 1 addition et k+1 multiplications
(1 + k de la boucle interne); soit k+2 opérations pour l'itération k.

Au total on a donc :
C(n)= 2 + (2+1) + (2+2) + ... + (2+k) + ... (2+n)
= 2(n+1) + (1+2+...+n) = 2(n+1) + n(n+1)/2 = (n+1)(n+4)/2.
C(n) est bien quadratique.
```

4. Rappeler l'algorithme d'évaluation de Horner (version itérative `H`) et sa complexité. **/2**

```
Voir cours. La complexité de Horner est linéaire.
```

5. Écrire une version récursive `H_rec` de l'évaluation de Horner. **/3**

```
fonction H_rec(n: entier, p[n]: tableau de flottants, x: flottant)
  retourne flottant
debut
  si n==0
    retourne p[0]
  sinon
    retourne p[n] + x * H_rec(n-1, p, x)
  fin si
fin fonction
```

6. Quelle est la complexité de cette version récursive de l'évaluation de Horner ? **/1**

```
La même : la complexité de Horner est linéaire.
```

7. Expliciter les états successifs de la pile des appels récursifs pour évaluer un polynôme de degré 3 avec l'algorithme de Horner. **/2**

```
                H(0)
            H(1) H(1) H(1)
        H(2) H(2) H(2) H(2) H(2)
. H(3) H(3) H(3) H(3) H(3) H(3) H(3) H(3) .
```