Année universitaire : 2018 / 2019

DIU Enseigner l'Informatique au Lycée
UE 2 – Algorithmique

Epreuve Commune Anonyme
Date : 4 juillet 2019
Durée : 2h

Documents et téléphones portables interdits. Le barème est donné à titre indicatif. Travaillez au brouillon d'abord de sorte à rendre une copie propre. **Il sera tenu compte de la présentation et de la clarté de vos réponses**.

Exercice 1 : Complexité des algorithmes (8 points)

Question 1.1 : On considère le code suivant, comportant deux « tant que » imbriqués. On cherche à mesurer la complexité de cette imbrication en fonction de n. Pour cela, on utilise la variable **`compteur`**, qui est incrémentée à chaque passage dans le « tant que » interne.

```
def procedure(n) :
  compteur = 0
  i = 1
  while i < n :
     j = i + 1
     while j <= n :
        compteur = compteur + 1
        j = j + 1
     i = i * 2
```

a. Quelle est la valeur finale du compteur dans le cas où n = 16 ?
b. Considérons le cas particulier où n est une puissance de 2 : on suppose que $n = 2^p$ avec p connu. Quelle est la valeur finale du compteur en fonction de p ? Justifiez votre réponse.
c. Exprimez le résultat précédent en fonction de n.
d. En conclure la complexité dans le pire des cas, en notation O, de cette procédure.

a.
Pour i=1, j varie de 2 à 16 inclus, on fait donc 15 incrémentations du compteur.
Pour i=2, j varie de 3 à 16 inclus, on fait donc 14 incrémentations du compteur.
Pour i=4, j varie de 5 à 16 inclus, on fait donc 12 incrémentations du compteur.
Pour i=8, j varie de 9 à 16 inclus, on fait donc 8 incrémentations du compteur.
Ensuite, i vaut 16, donc on sort du « while i < n ».
Au total, on a donc fait 15+14+12+8 = 49 incrémentations du compteur. Donc compteur vaut 49 en sortie du programme.

b.
i prend successivement les valeurs suivantes : $2^0, 2^1, 2^2, \ldots 2^{p-1}$, soit 2^k avec k variant de 0 à (p-1). Pour chacune de ces valeurs, on fait (n-i) incrémentations, soit $(2^p - 2^k)$ incrémentations. Ensuite i vaut 2^p, ce qui provoque la sortie du « while i < n ». On ne fait pas d'incrémentations du compteur pour cette dernière valeur de i.
$\sum_{k=0}^{p-1}(2^p - 2^k) = p \times 2^p - \sum_{k=0}^{p-1} 2^k = p \times 2^p - (2^p - 1) = (p-1) \times 2^p + 1$
Ainsi, la valeur finale du compteur est $(p-1) \times 2^p + 1$.

c. On a $n = 2^p$ donc $p = \log_2(n)$, la valeur finale du compteur est donc $(\log_2(n) - 1) \times n + 1 = n \times \log_2(n) - n + 1$.

d. On a donc une complexité en $O(n \log n)$.

Question 1.2 : Donner la fonction Python de recherche dichotomique dans une liste triée. La liste et l'élément à rechercher sont donnés en paramètres. La fonction retourne l'indice de l'élément s'il est présent et -1 sinon. Déterminer ensuite, par la méthode du Master Theorem, la complexité de cette fonction.

L'algorithme est le suivant :

```
def rechercheDichoRec(tab,debut,fin,valeur):
    if fin < debut: return -1
    else:
        ind_milieu = (debut + fin)//2
        if tab[ind_milieu] == valeur : return ind_milieu
        if valeur < tab[ind_milieu]:
            return rechercheDichoRec(tab,debut,ind_milieu-1,valeur)
        else:
            return rechercheDichoRec(tab,ind_milieu+1,fin,valeur)

def rechercheDicho(tab,valeur):
    return rechercheDichoRec(tab,0,len(tab)-1,valeur)
```

Dans cet algorithme, le coût de la séparation des données pour réaliser l'appel est constant : il s'agit simplement de calculer la valeur stockée dans la variable ind_milieu. Ensuite le résultat est renvoyé immédiatement, donc le coût de reconstruction du résultat est nul. Le coût total de ces opérations est donc $\Theta(1)$. Pour les appels récursifs, le problème initial est divisé en 1 problème de taille deux fois plus petite. Nous avons donc, dans la notation du Master Theorem, $a = 1, b = 2$ et $f(n)$ est en $\Theta(1)$. Le temps d'exécution de la fonction récursive est donc $T(n)=T(n/2)+\Theta(1)$. Nous avons $\log_b(a) = \log_2(1) = 0$ et $f(n)=\Theta(1)=\Theta(n^0)$. Nous sommes donc dans le troisième cas du Master Theorem où les appels récursifs et les calculs extérieurs sont du même ordre. La complexité est donc en $\Theta(n^0\log_2(n))= \Theta(\log_2(n))$. Ce qui est normal pour un algorithme de recherche dichotomique dans une liste triée.

Question 1.3 : Montrer que $f(n) = 2n^2 - n + 1$ est $O(n^2)$.

Nous devons montrer qu'il existe une constante positive c et un entier constant $n_0 \geq 1$ tels que : $f(n) \leq c \times n^2$ pour tout $n \geq n_0$. C'est-à-dire que $2n^2 - n + 1 \leq cn^2$. Si on choisit par exemple $c = 2$ et $n_0 = 1$, alors nous avons bien $f(n) \leq 2n^2$ pour tout $n \geq 1$. Ce qui démontre que $f(n)$ est $O(n^2)$.

Exercice 2 : Algorithme glouton – Problème du sac à dos (6 points)

On dispose d'un ensemble S de n objets. Chaque objet i possède une valeur b_i et un poids w_i. On souhaiterait prendre une partie T de ces objets dans notre sac à dos, malheureusement, ce dernier dispose d'une capacité limitée (en poids) W. On cherche à maximiser la somme des valeurs des objets que l'on peut mettre dans le sac à dos, sans en dépasser la capacité.

Mathématiquement, cela se traduit par :

$$\max_{T \subseteq S} \sum_{i \in T} b_i \quad \text{avec} \quad \sum_{i \in T} w_i \leq W$$

L'idée à suivre, se reposant sur le principe des algorithmes gloutons, est d'ajouter les objets de valeurs élevées en premier, jusqu'à saturation du sac.

Prenons l'exemple suivant d'un ensemble S de $n = 14$ objets et d'un sac à dos de capacité $W = 26$.

Objet	A	B	C	D	E	F	G	H	I	J	K	L	M	N
Valeur	4	3	8	5	10	7	1	7	3	3	6	12	2	4
Poids	2	2	5	2	7	4	1	4	2	1	4	10	2	1

Suivons le principe de la méthode et prenons les objets de plus grande valeur d'abord. Ça nous donne le sous-ensemble d'objets suivant : $T = \{L(12,10);\ E(10,7);\ C(8,5);\ F(7,4)\}$. Notre sac est tout juste saturé (poids de 26) et la somme des valeurs des objets qu'il contient est de 37. Mais cette solution est-elle optimale ?

Supposons la liste Python **`objets`** qui contient les objets disponibles sous la forme `[ [ objet1 , valeur1 , poids1 ] , [ objet2 , valeur2 , poids2 ] , ... ]` et triée par ordre décroissant de valeur.

Question 2.1 : Ecrire la fonction **`sacADos`**, qui prend en argument la capacité W du sac à dos et qui retourne la liste des noms des objets de valeur maximale grâce à la stratégie présentée ci-dessus.

```
objets = [['L',12,10], ['E',10,7], ['C',8,5], ['F',7,4], ['H',7,4], ['K',6,4], ['D',5,2],
['A',4,2], ['N',4,1], ['B',3,2], ['I',3,2], ['J',3,1], ['M',2,2], ['G',1,1]]

def sacADos(W) :
   poidsActuel = 0
   solution = []
   for i in range(len(objets)) :
      if poidsActuel + objets[i][2] <= W:
         poidsActuel += objets[i][2]
         solution.append(objets[i][0])
   return solution

print(sacADos(26))
```

Question 2.2 : Que va retourner cette fonction pour une capacité de sac à dos de 40 avec les objets suivants ? Qu'en pensez-vous ?

Objet	A	B	C	D	E	F
Valeur	30	12	12	12	12	4
Poids	39	10	10	10	10	1

L'algorithme choisira l'objet A et l'objet F, ce qui fera une somme des valeurs de 34. Pourtant, on remarque directement qu'en choisissant les 4 objets B,C,D,E on aurait pu atteindre une somme des valeurs de 48, pour le même poids. L'algorithme n'a pas produit une solution optimale.

Exercice 3 : Correction des algorithmes (6 points)

Question 3.1 : Ecrire une version naïve de la fonction qui calcule la valeur de x^n. Cette fonction prendra x et n en paramètre et retournera la valeur x^n. Cette fonction utilisera la méthode des multiplications successives (multiplier n fois x avec lui-même).

L'algorithme est le suivant :

```
def puissance(x,n) :
   res = 1
   for i in range(n) :
      res = res * x
   return res
```

Question 3.2 : Démontrer la terminaison de votre fonction. Quelles en sont les préconditions ?

Nous pouvons choisir la valeur n-i comme variant pour la boucle pour. n-i vaut n initialement qui est positif ou nul (précondition de notre fonction). i croît par pas de 1 jusqu'à la valeur n. Notre variant est donc positif ou nul et strictement décroissant, ce qui prouve la terminaison de notre fonction (pour tout n positif ou nul). Les préconditions sont donc : n est un entier positif ou nul et x est un nombre réel.

Question 3.3 : Donner un invariant pour la boucle que vous avez créé et démontrer la correction de votre fonction.

Un invariant possible est « Au début de l'itération i, res contient x^i ».

Initialisation : En entrant dans la première itération (c'est-à-dire pour i=0), res contient initialement 1. Or on a bien $1 = x^0$ donc la propriété est vérifiée.

Conservation : On suppose que l'invariant est vrai pour i, c'est-à-dire que « Au début de l'itération i, res contient x^i » et on veut montrer que l'invariant est vrai pour i+1, c'est-à-dire que « Au début de l'itération i+1, res contient x^{i+1} ». A l'itération i+1, on exécute le code res=res*x donc on a $res = x^i \times x = x^{i+1}$. Cela démontre bien l'invariant pour i+1. L'invariant est donc conservé.

Conclusion : En sortant de la boucle, i a pour valeur n (ce qui provoque la sortie), et l'invariant nous dit que res contient x^n. Comme c'est bien la valeur que l'on retourne, la correction de la fonction est prouvée.