Université d'El Oued
Département d'informatique

Complexité des algorithmes

Master 1 SDIA

Correction - Examen du 21 Mai 2023

Exercice 1 (5 pts)

Trouver les ordres de grandeur en notation 'Grand O' des fonctions suivantes, puis les classer :

$f_1(n) = n^4 + 2n^3 + 5n + 7$
$f_2(n) = 2^n + n!$
$f_3(n) = n^2\log(n) + 2n^2$
$f_4(n) = n\log(3^n) + (n-1)$

$\boldsymbol{f_1(n) = O(n^4)}$
$\boldsymbol{f_2(n) = O(n!)}$
$\boldsymbol{f_3(n) = O(n^2\log(n))}$
$\boldsymbol{f_4(n) = O(n^2)}$

$\boldsymbol{f_4 \ll f_3 \ll f_1 \ll f_1}$

Exercice 2 (5 pts)

Formuler et déterminer la complexité -en termes de nombre de multiplications de réels- des algorithmes de calcul de puissance x^n suivants (on considère que $n = 2^k$) :

`double PWR1(double x, int n)`
`{`
`  double rs = 1.;`
`  int i;`
 	
`  for (i = 1 ; i <= n ; i++)`
`  rs = rs * x;`
 	
`  return rs;`
`}`
	$C_{PWR1}(n) = n$
	C'est le nombre de multiplications effectue dans la boucle for

`double PWR2(double x, int n)`
`{`
`  if (n == 1)`
`  return 1.;`
`  else`
`  return (x * PWR2(x, n - 1));`
`}`
	$C_{PWR2}(n) = \begin{cases} 0 & si\ n = 1 \\ 1 + C_{PWR2}(n-1) & si\ n > 1 \end{cases}$
	$C_{PWR2}(n) = n - 1$

```
double PWR3(double x, int n)
{
    double t;

    if (n == 1)
        return 1.;
    else
    {
        t = PWR3(x, n / 2);
        return (t * t);
    }
}
```

$$C_{PWR3}(n) = \begin{cases} 0 & si\ n = 1 \\ C_{PWR3}(n/2) + 1 & si\ n > 1 \end{cases}$$

$$C_{PWR3}(n) = \log_2 n$$

```
double PWR4(double x, int n)
{
    double t;

    if (n == 1)
        return 1.;
    else
         return (PWR4(x, n / 2) * PWR4(x, n / 2));
}
```

$$C_{PWR4}(n) = \begin{cases} 0 & si\ n = 1 \\ 2 \times C_{PWR4}(n/2) + 1 & si\ n > 1 \end{cases}$$

$$C_{PWR4}(n) = n - 1$$

```
double PWR5(double x, int n)
{
    double t;

    if (n == 1)
        return 1.;
    else
        return PWR5(x * x, n / 2);
}
```

$$C_{PWR5}(n) = \begin{cases} 0 & si\ n = 1 \\ 1 + C_{PWR3}(n/2) & si\ n > 1 \end{cases}$$

$$C_{PWR5}(n) = \log_2 n$$

Exercice 3 (5 pts)

1. Décrire des algorithmes pour tester si un premier tableau est une version triée d'un second tableau, de n éléments chacun, en considérant deux cas d'utilisation : d'une (1) recherche séquentielle et d'une (2) recherche binaire (par dichotomie) pour chercher une valeur dans un tableau trié.
2. Déterminer la complexité de ce calcul (test d'une version triée) dans les deux cas.

Pour réaliser ce calcul, soient les fonctions suivantes :

```
int estTrie(double T[], int n) ;
```

Cette fonction teste si un tableau de réels de taille n est trié ou non. Le nombre de comparaisons de réels dans le pire de cas est $(n - 1)$, i.e., une complexité en $O(n)$.

```
int rechercheSequentielle(double T[], int n, double x) ;
```

C'est la fonction de recherche séquentielle (voir TD/TP1). Le nombre de comparaisons de réels dans le pire de cas est n, i.e., une complexité en $O(n)$.

```
int rechercheBinaire(double T[], int n, double x) ;
```

C'est la fonction de recherche binaire (voir TD/TP1). Le nombre de comparaisons de réels dans le pire de cas est $(\log_2 n)$, i.e., une complexité en $O(log\, n)$.

```
int estVersionTrie1(double T1[], double int T2[], int n)
{
    int i ;

    if (!estTrie(T1, n))
        return 0 ;

    for (i = 0 ; i < n ; i++)
        if (rechercheSequentielle(T1, n, T2[i]) == 0) // non trouvée
            return 0 ;

    return 1 ; // le deux tests (trié et version) sont vérifiés
}
```

La fonction qui teste si un premier tableau T1 est une version triée d'un second tableau T2 en utilisant une recherche séquentielle.

```
int estVersionTrie2(double T1[], double int T2[], int n)
{
    int i ;

    if (!estTrie(T1, n))
        return 0 ;

    for (i = 0 ; i < n ; i++)
        if (rechercheBinaire(T1, n, T2[i]) == 0) // non trouvée
            return 0 ;

    return 1 ; // le deux tests (trié et version) sont vérifiés
}
```

La fonction qui teste si un premier tableau T1 est une version triée d'un second tableau T2 en utilisant une recherche binaire.

- La complexité de la fonction `estVersionTrie1` est donnée par $C_1(n) = n + n \times n$. $C_1(n) \in O(n^2)$

- La complexité de la fonction `estVersionTrie2` est donnée par $C_2(n) = n + n \times \log_2 n$. $C_2(n) \in O(n \log n)$

Exercice 4 (5 pts)

On applique la même stratégie de partitionnement (basée pivot) que le tri rapide pour localiser le k-ième plus petit élément dans un tableau non trié de n éléments (on dit la sélection rapide) :

1. Décrire ce partitionnement récursif pour la sélection rapide.
2. Décrire les scénarios et donner les complexités moyenne et dans le pire cas.

1. La sélection rapide (quickselect) utilise la même même stratégie de partitionnement que le tri rapide (quicksort), choisissant un élément à la fois, afin de partitionner les éléments selon le pivot. Cependant, au lieu de séparer les éléments en deux parties comme dans quicksort, quickselect n'utilise la récursion que sur un le côté contenant l'élément cherché (k-ième plus petit élément).

2. Tout comme quicksort, l'efficacité de l'algorithme quickselect dépend du choix du pivot. Si le choix de pivot permet de faire décroître la taille du tableau à chaque étape d'une même fraction (1 / 2), alors la décroissance est exponentielle, et la complexité est linéaire. Si au contraire le pivot ne fait diminuer la taille que de 1, alors le pire cas a une complexité quadratique, $O(n^2)$. Cela arrive principalement si le pivot est l'élément le plus à droite et que le tableau est déjà trié.