TP2 ALGORITHMIQUE ET PROGRAMMATION GR 7 G2 POLYTECH UNIKIN 2021-2022

BAZAYAMA Abel METENA Yves YEKWA John

mardi 07 février 2023

1 Questions et Réponses

1.1

Le nombre d'opérations primitives exécutées par les algorithmes A et B est $50n\log(n)$ et $45n^2$, respectivement. Déterminez n_0 tel que A soit meilleur que B, pour tout $n \geq n_0$.

Résolution

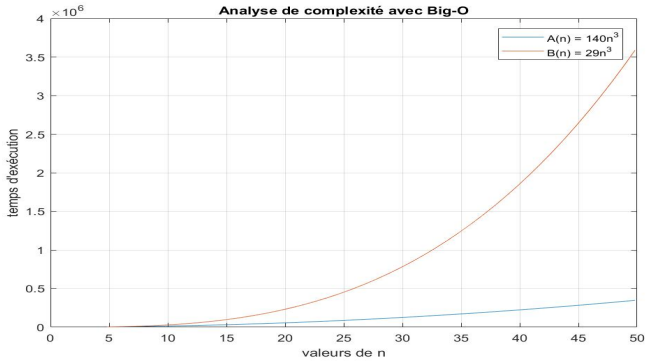
A sera meilleur que B quand
$O_A(n) \leq O_B(n) \Leftrightarrow 50nlog(n) \leq 45n^2$
$\Leftrightarrow 50nlog(n) \leq 45n^2$
$\Leftrightarrow 10log(n) \leq 9n$
$\Leftrightarrow log(n) \leq \frac{9}{10}n$
$\Leftrightarrow (\frac{9}{10})^{10} \leq n$
$\Leftrightarrow n_0 \simeq 0.349$

1.2

Le nombre d'opérations primitives exécutées par les algorithmes A et B est $(140n^2)$ et $(29n^3)$, respectivement. Déterminez n_0 tel que A soit meilleur que B pour tout $n \geq n_0$. Utiliser Matlab ou Excel pour montrer les évolutions des temps d'exécution des algorithmes A et B dans un graphique.

Résolution

A sera meilleur que B quand :
$O_A(n) \leq O_B(n) \Leftrightarrow (140n^2) \leq (29n^3)$
$\Leftrightarrow 140n^2 \leq 29n^3$
$\Leftrightarrow \frac{140}{29} \leq n$
$\Leftrightarrow n_0 \simeq 4.83$
Ci-dessous le graphique généré avec Matlab



1.3

Montrer que les deux énoncés suivants sont équivalents :

1. Le temps d'exécution de l'algorithme A est toujours $O(f(n))$.
2. Dans le pire des cas, le temps d'exécution de l'algorithme A est $O(f(n))$.

Résolution

Supposons que le temps d'exécution de l'algorithme A est $f(n) = an^2 + bn + c$ et dans le pire de cas $f(n) = tn^2 + rn + k$
$\Rightarrow O(f(n)) = f(n) = n^2$ (dans le deux cas, parce que on prend le plus grand)
on sait que pour la notation O ceci veut dire :
$\{\exists c, n_0 | 0 \leq n^2 \leq cn^2, \forall n \geq n_0\}$ est pour le deux énoncés cela se vérifie.
Alors littéralement ça s'explique comme suit : lors de la conception d'un algorithme surtout de traitement des informations on prend le cas le plus défavorable pour être sûr que tous les autres cas seront traités et que l'algorithme ne prend pas plus de temps que celui prévue lors de l'analyse, et c'est ce temps du plus défavorable des cas qui gouverne le temps total d'un algorithme, imaginez un tableau de 10 éléments et que nous voulons faire une recherche d'un élément, bien évidement on prendra le temps d'exécution selon lequel l'élément se situe à la dixième place, c'est ça le cas le plus défavorable, et ça sera le temps d'exécution de notre algorithme.

1.4

Montrer que si $d(n)$ vaut $O(f(n))$, alors $(ad(n))$ vaut $O(f(n))$, pour toute constante $a > 0$.

Résolution

Soit $\forall a \in R | a > 0$
par la définition de Big-O (O) nous avons :
$\forall n \geq n_0, f(n) \leq cf(n) \Rightarrow kf(n) \leq kcf(n)$
En choisissant le même n_0 et la constante kc, la définition est valable.

En termes de temps d'exécution, cela signifie que répéter un morceau de code nombre de fois ne change pas sa complexité ; Donc les boucles qui s'exécutent un nombre constant de fois n'augmentent pas la complexité du pire des cas.

1.5

Montrer que si $d(n)$ vaut $O(f(n))$ et $e(n)$ vaut $O(g(n))$, alors le produit $d(n)e(n)$ est $O(f(n)g(n))$.

Résolution

On sait que selon Big-O :
$d(n) \in O(f(n)) \Rightarrow \{\exists n_1, c_1 | d(n) \leq c_1 f(n), \forall n \geq n_1\}$
$e(n) \in O(g(n)) \Rightarrow \{\exists n_2, c_2 | e(n) \leq c_2 g(n), \forall n \geq n_2\}$
$n_0 = max(n_1, n_2)$, ainsi $\forall n \geq n_0$,
Les deux inégalités sont valables, et en multipliant nous avons :
$\{d(n)e(n) \leq (c_1 c_2) f(n)g(n), \forall n \geq n_0\}$
Ce qui signifie que : $d(n)e(n) \in O(f(n)g(n))$

Dans le code cette situation correspond à l'imbrication, par exemple : boucles for imbriquées, ou un appel à l'intérieur d'une boucle ou fonction récursive, en pratique ceci veut dire que l'imbrication augmentera la complexité du code.

1.6

Montrer que si $d(n)$ vaut $O(f(n))$ et $e(n)$ vaut $O(g(n))$, alors $d(n) + e(n)$ vaut $O(f(n) + g(n))$.

Résolution

On sait que selon Big-O :
$d(n) \in O(f(n)) \Rightarrow \{\exists n_1, c_1 | d(n) \leq c_1 f(n), \forall n \geq n_1\}$
$e(n) \in O(g(n)) \Rightarrow \{\exists n_2, c_2 | e(n) \leq c_2 g(n), \forall n \geq n_2\}$
$n_0 = max(n_1, n_2)$, ainsi $\forall n \geq n_0$,
$\{d(n) + e(n) \leq (c_1 f(n)) + (c_2 g(n)), \forall n \geq n_0\}$
Ce qui signifie que : $d(n) + e(n) \in O(f(n) + g(n))$

Dans le code cette situation se produit lorsque nous avons une séquence d'opérations, par exemple deux boucles for se succédant, une boucle for suivie d'un appel de fonction etc. Ce résultat signifie essentiellement que la complexité temporelle d'une séquence d'opérations sera dominée par l'opération la plus longue de la séquence.

1.7

Montrer que si $d(n)$ est $O(f(n))$ et $e(n)$ est $O(g(n))$, alors $d(n) - e(n)$ n'est pas nécessairement $O(f(n) - g(n))$.

Résolution

On sait que selon Big-O :
$d(n) \in O(f(n)) \Rightarrow \{\exists n_1, c_1 | d(n) \leq c_1 f(n), \forall n \geq n_1\}$
$e(n) \in O(g(n)) \Rightarrow \{\exists n_2, c_2 | e(n) \leq c_2 g(n), \forall n \geq n_2\}$
soit l'exemple suivant :
$d(n) = 2n$ et $e(n) = n$
$\Rightarrow d(n) - e(n) = n$

Comme :
$d(n) = O(f(n))$
$e(n) = O(g(n))$
$f(n) = g(n) = n$
$\Rightarrow (f(n) - g(n)) = O(n - n) = O(1)$
$\Rightarrow \{\exists c_1, c_2, n_0 | d(n) - e(n) \geq c_1 f(n) c_2 e(n), \forall n \geq n_0\}$

1.8

Montrer que si $d(n)$ est $O(f(n))$ et $f(n)$ est $O(g(n))$, alors $d(n)$ est $O(g(n))$.

Résolution

On sait que selon Big-O :
$d(n) \in O(f(n)) \Rightarrow \{\exists n_1, c_1 | d(n) \leq c_1 f(n), \forall n \geq n_1\}$
$f(n) \in O(g(n)) \Rightarrow \{\exists n_2, c_2 | f(n) \leq c_2 g(n), \forall n \geq n_2\}$
$n_0 = max(n_1, n_2)$
$c_3 = c_1, c_2$
Alors :
$f(n) \leq c_1 f(n) \leq c_1 c_2 g(n) = c_3 g(n)$
$\Rightarrow d(n) = O(g(n))$

1.9

Étant donné une séquence de n éléments S, l'algorithme D appelle l'algorithme E sur chaque élément $S[i]$. L'algorithme E s'exécute en un temps $O(i)$ lorsqu'il est appelé sur l'élément $S[i]$. Quel est le pire temps d'exécution de l'algorithme D ?

Résolution

Le pire temps d'exécution de D , c'est le cas le plus défavorable, ici expliqué par le fait que D appelle E à $S[n]$ ce qui fera que E s'exécute en un temps $O(n)$ à la $n^{émè}$ fois , ce qui nous donne :
$O(D(n))(O(i))$ ce qui semble être $O(n * 1) = O(n)$ de la description nous aurons enfin $O(D(n)) * (O(i))$ qui semble être $O(n * 1) = O(n)$ à partir du descriptif.

1.10

Alphonse et Bob se disputent à propos de leurs algorithmes. Alphonse revendique le fait que son algorithme de temps d'exécution $O(nlogn)$ est toujours plus rapide que l'algorithme de temps d'exécution $O(n^2)$ de Bob. Pour régler la question, ils effectuent une série d'expériences. À la consternation d'Alphonse, ils découvrent que si $n < 100$, l'algorithme de temps $O(n^2)$

s'exécute plus rapidement, et que c'est uniquement lorsque $n \geq 100$ est le temps $O(nlogn)$ est meilleur. Expliquez comment cela est possible.

Résolution

La notation O signifie qu'il existe une constante c telle que $f(n) < cg(n)$. Par conséquent, si les algorithmes d'Alphonse fonctionnent mieux que $A(nlogn)$ et que ceux de Bob l'algorithme fonctionne mieux que $B(n^2)$, nous pouvons résoudre la valeur $Anlogn = Bn^2$, dont nous savons qu'elle est vraie lorsque $n = 100$ cela signifie que $(A/B) = (100)/(log(100)) = 15.05$

Cela signifie que le temps d'exécution de l'algorithme sur une seule itération est 15 fois plus lent, mais puisqu'il effectue globalement moins d'opérations, il commence à mieux fonctionner à de grandes valeurs de n.

1.11

Concevoir un algorithme récursif permettant de trouver l'élément maximal d'une séquence d'entiers. Implémenter cet algorithme et mesurer son temps d'exécution. Utiliser Matlab ou Excel pour "fitter" les points expérimentaux et obtenir la fonction associée au temps d'exécution. Calculer par la méthode des opérations primitives le temps d'exécution de l'algorithme. Comparer les deux résultats.

Résolution

Voici un algorithme permettant de trouver un max d'une séquence d'entiers :

```
fonction trouve_max(sequence){
si(longueur(sequence)==1){
sortie sequence[0]
}sinon{
max_suiv = trouve_max(sequence - sequence[0])
si(sequence[0]>max_suiv){
sortie sequence[0]
}sinon{
sortie max_suiv
}
}
}
```

1.12

Concevoir un algorithme récursif qui permet de trouver le minimum et le maximum d'une séquence de nombres sans utiliser de boucle.

Résolution

Voici un algorithme permettant de trouver l'élément maximal et minimal d'une séquence d'entiers.

```
fonction min_max_recursive(sequence){
si (longueur(sequence) == 0){
sortie sequence[0] et sequence[0]
}sinon{
minimal et maximal = min_max_recursive(sequence - sequence[0])
si (sequence[0] < minimal){
minimal = sequence[0]
}
si (sequence[0] > maximal){
maximal = sequence[0]
}
sortie minimal,maximal
}
}
```

1.13

Concevoir un algorithme récursif permettant de déterminer si une chaine de caractères contient plus de voyelles que de consonnes.

Résolution

Voici un algorithme :

```
fonction p_voyelles(chaine){
si (longueur(chaine)==0){
sortie Vrai
}sinon{
premier_caract = chaine[0]
si (premier_caractere.minuscule() est dans ['a','e','o','i','u','y']){
sortie p_voyelles(chaine - chaine[0])
}sinon{
sortie pas p_voyelles(chaine - chaine[0])
}

}
}
```

commentaires :

la fonction $p_voyelles$ prend en entrée une chaine de caractères chaine ; si la chaine est vide, donc la foncion retourne vrai ;

sinon on examine le premier caractère de la chaine $premier_caract$ si c'est une voyelle on fait l'appel recursif de la partie restante ; si c'est une consonne on fait un appel inverse du précédant car on cherche à savoir s'il y a plus de voyelles que de consonnes.