UNIVERSITE DE KINSHASA

FACULTE POLYTECHNIQUE

DEVOIR D'ALGORITHME ET PROGRAMMATION (TP3)

Travail exécuté
par :

SEMIKENKE LWANGA DEOGRATIAS (2GC),
IKINDA BALOMBO CHADRACK (2GC)
MBALA KABENGELA MIKE (2GC)

ANNEE ACADEMIQUE 2020-2021

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

Réponse

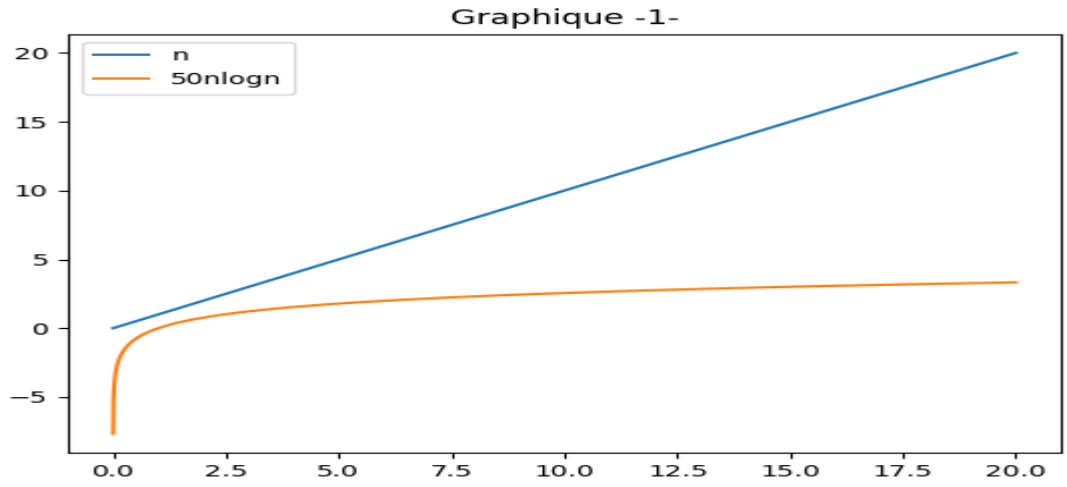
S'il nous est demandé de dire A est meilleur que B, il nous faudra comparer leurs nombres d'opérations primitives, cette comparaison nous permettra de déterminer la valeur de n_0 pour laquelle A et B auront un même nombre d'opérations.

Ayant, $50n\log n = 45n^2$, on peut autrement l'écrire après simplification, $\frac{10}{9}\log n = n$, Vu la difficulté de résoudre cette égalité par des méthodes numériques, faisons intervenir la représentation graphique des courbes $y_1 = n$ et $y_2 = \frac{10}{9}\log n$, où $n \in \mathbb{N}^*$. [*A l'aide de Python*]

Code :

```
import matplotlib.pyplot as plt
import numpy as np
from math import *
n = [n/1000 for n in range(1,20000)]
y2 = (10/9)*np.log(n)
y1 = n
plt.plot(n, y1, label = "n")
plt.plot(n, y2, label = "50nlogn")
plt.title("Graphique -1-")
plt.legend()
plt.show()
```

Graphique généré :



On peut graphiquement conclure que vu la disposition de nos courbes que $\forall n \in \mathbb{N}^*$, que $\frac{10}{9}\log n < n \Rightarrow 10\log n < 9n \Rightarrow 10n\log n < 9n^2 \Rightarrow 50n\log n < 45n^2$, tenant compte que nos deux courbes ne se croisent pas en un point quelconque, $\boldsymbol{n_0 = 1}$.

Conclusion : Connaissant que $50n\log n < 45n^2$, alors **A est meilleur que B quelque soit le nombre d'opérations primitives effectuées**.

2. Le nombre d'opérations primitives exécutées par les algorithmes A et B est ($140\ n^2$) et ($29\ n^3$), 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.

Réponse

L'objectif étant de déterminer n_0 tel que A soit meilleur que B pour tout $n \geq n_0$, nous sommes contraints de déterminer la valeur n pour laquelle A et B ont le même nombre d'opérations primitives, cela revient à résoudre l'équation : $140\ n^2 = 29\ n^3 \Rightarrow n = \frac{140}{29} \approx 4{,}8$.

On remarquera que **A est meilleur que B pour tout $n > 4.8$**, par conséquent $\boldsymbol{n_0 = 4}$

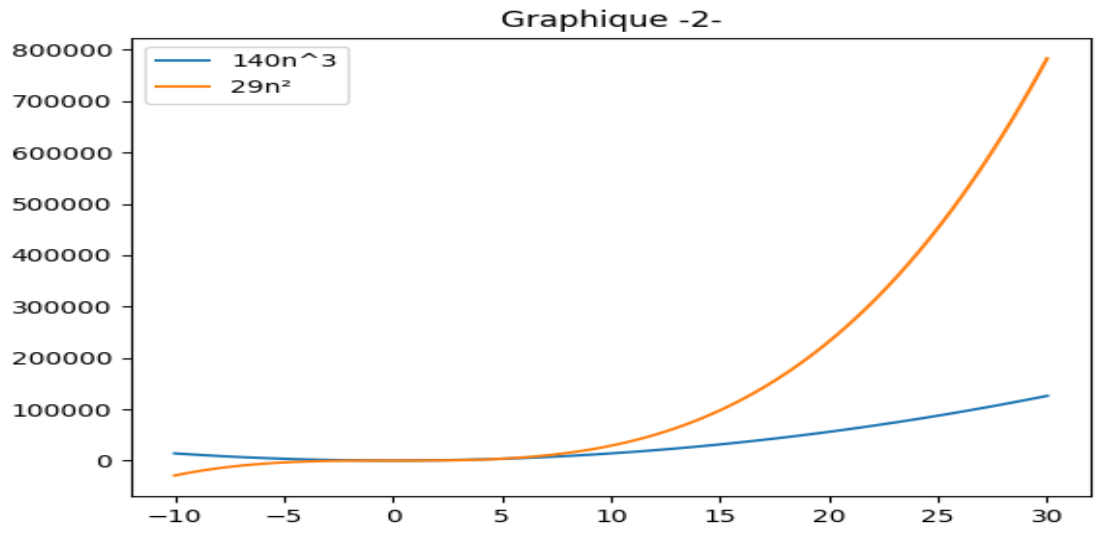
Code :

```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
n = np.linspace(-10, 30, 100)
ax.plot(n, 140*(n**2), label = '140n^3')
ax.plot(n, 29*(n**3), label = '29n²')

plt.title("Graphique -2-")
plt.legend()
plt.show()
```

Graphique généré :



3. Montrer que les deux énoncés suivants sont équivalents :
(a) Le temps d'exécution de l'algorithme A est toujours O(f(n)).
(b) Dans le pire des cas, le temps d'exécution de l'algorithme A est O(f(n)).

Réponse

Connaissant que le temps d'exécution pour le pire des cas de l'algorithme A est $O(f(n))$, c'est supposé qu'il existe une certaine constante c telle que $c * f(n) >$ au cas pire, pour tout $n \geq n_0$.

Dans tous les cas, nous connaissons que le temps d'exécution du pire cas est toujours supérieur à ceux de tous les autres cas, alors $c * f(n) >$ au cas pire > tous les autres cas de A.

Vu qu'il s'agit simplement de l'ajout d'une constante multiplicatif, alors on ne peut que conclure que le temps d'exécution de l'algorithme A est aussi $O(f(n))$.

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

Réponse

Si $d(n) = O(f(n))$, il existe une constante c telle que $d(n) \leq c * f(n)$ pour tout $n \geq n_0$.

Donc si nous multiplions $d(n)$ par a, on obtiendra dans ce cas : $a * d(n) \leq a * c * f(n)$.

En posant $a * c = k = cste$, on revient à la forme $a * d(n) \leq k * f(n)$ répondant à la condition de la notation asymptotique O. D'où $\boldsymbol{a * d(n) = O(f(n))}$.

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

Réponse

Pour faciliter la différence de nos différents n, posons $e(n) = e(n')$

Si d(n) = O(f(n)), alors $\exists\ c = cste$ tel que $d(n) \leq c * f(n)$ pour tout $n \geq n_0$ et Si e(n') = O(g(n)), alors $\exists\ k = cste$ tel que $e(n') \leq k * g(n)$ pour tout $n' \geq {n'}_0$

Etant donné que $d(n) \leq c * f(n)$ et $e(n') \leq k * g(n)$ alors on peut en déduire que le produit

$d(n) * e(n') \leq c * f(n) * k * g(n) \Rightarrow d(n) * e(n') \leq (c * k) * (f(n)g(n))$, avec $n * n' \geq n_0 * {n'}_0$.

En posant $c * k = c' = cste$, alors on peut écrire $d(n) * e(n') \leq c' * (f(n)g(n))$, avec $n * n' \geq n_0 * {n'}_0$ qui répond également à la condition de la notation asymptotique O car $\boldsymbol{c'}$ est juste une constante multiplicative.

D'où la conclusion : $d(n) * e(n') = O\big(f(n)g(n)\big)$, ayant posé ci-haut que $e(n) = e(n')$, alors on a donc, : $\boldsymbol{d(n) * e(n) = O\big(f(n)g(n)\big)}$.

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)).

Réponse

Pour faciliter la différence de nos différents n, posons $e(n) = e(n')$

Si d(n) = O(f(n)), alors $\exists\ c = cste$ tel que $d(n) \leq c * f(n)$ pour tout $n \geq n_0$ et si e(n') = O(g(n)), alors $\exists\ k = cste$ tel que $e(n') \leq k * g(n)$ pour tout $n' \geq {n'}_0$

Etant donné que $d(n) \leq c * f(n)$ et $e(n') \leq k * g(n)$ alors on peut en déduire que le produit $d(n) + e(n') \leq c * f(n) + k * g(n)$, avec $n + n' \geq n_0 + {n'}_0$.

Dans le but de simplifier les expressions, posons $n'' = n + n'$ et ${n''}_0 = n_0 + {n'}_0$, On a donc : $d(n) + e(n') \leq c * f(n) + k * g(n)$, avec $n'' \geq {n''}_0$ et ceci peut être réarrange en écrivant $d(n) + e(n') \leq f(cn) + g(kn)$, cette dernière manipulation n'aura aucun impact sur la notation asymptotique de nos fonctions f et g, d'où on peut se passer des constantes c et k et ne retenir que $d(n) + e(n') \leq f(n) + g(n)$. Ayant posé au préalable $e(n) = e(n')$, alors $d(n) + e(n') \leq f(n) + g(n) \Rightarrow d(n) + e(n) \leq (f(n) + g(n))$ qui respecte la condition de la notation asymptotique O.

Conclusion : $\boldsymbol{d(n) + e(n) = O(f(n) + g(n))}$.

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)).

Réponse

Si $d(n) = O\big(f(n)\big) \Rightarrow \exists\ c\ tel\ que\ d(n) \leq c * f(n), pour\ tout\ n > 0$ et si $e(n) = O\big(g(n)\big) \Rightarrow \exists\ k\ tel\ que\ e(n) \leq k * g(n), pour\ tout\ n > 0.$

En principe, $d(n) - e(n) = c * f(n) - k * g(n)$,

Dans le but de montrer de manière illustrative que $d(n) - e(n)$ n'est pas nécessairement égal à $O(f(n) - g(n)$, prenons le cas où $d(n)\ et\ e(n)$ sont des fonctions constantes égales.

Soit $d(n) = 10 = e(n)$,

- En se basant sur le fait que $d(n)$ vaut $O(f(n) \Rightarrow f(n) = n$
- En se basant également sur le fait que $e(n)$ vaut $O\big(g(n)\big) \Rightarrow g(n) = n$

Tentant d'appliquer $d(n) - e(n) \leq c * f(n) - k * g(n)$,

On aura : $0 \leq c * n - k * n \iff \boldsymbol{(c - k) * n \geq 0}$, cette dernière égalité ne pouvant pas être vérifié pour le cas où $k > c$. On peut trivialement conclure que $d(n) - e(n) = O(f(n) - g(n))$ pour certains cas.

D'où $\boldsymbol{d(n) - e(n)}$ **n'est pas nécessairement** $\boldsymbol{O(f(n) - g(n))}$

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

Réponse

Si $d(n)$ est $O(f(n))$ alors $d(n) \leq c * f(n)$, pour tout $n \geq n_0$

Si $f(n)$ est $O(g(n))$ alors $f(n) \leq k * g(n)$, pour tout $n' \geq {n'}_0$

Se servant de nos deux conditions précédentes, on peut donc écrire :

$d(n) \leq c * f(n) \leq c * k * g(n)$, pour tout $n * n' > n_0 * {n'}_0$

En posant $c' = c * k = cste$, on a l'expression $d(n) \leq c * f(n) \leq c' * g(n)$, pour tout $n * n' > n_0 * {n'}_0$. On déduit de ce qui précède que $\boldsymbol{d(n) \leq c' * g(n)}$ répondant correctement à la condition de la notation asymptotique O.

On conclut donc que : $\boldsymbol{d(n)\ est\ O(g(n))}$

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 ?

Réponse

Etant donné que nous avons deux algorithmes, l'un appelant l'autre, on déduit que nous devons avoir deux blocs correspondant à nos deux algorithmes (c'est-à-dire D et E).
Connaissant que E est appelé sur chaque élément de $S[i]$, et que S possède n éléments, le temps d'exécution $O(i)$ de E peut autrement s'écrire $O(n)$.
Vu que D fait appel à E ayant comme temps d'exécution $O(n)$ sur chaque élément (c'est-à-dire n fois), alors on peut sans hésiter conclure que D a comme temps d'exécution $O(n^2)$

10. Alphonse et Bob se disputent à propos de leurs algorithmes. Alphonse revendique le fait que son algorithme de temps d'exécution O(n log n) 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 ≥ 100 est le temps O (n log n) est meilleur. Expliquez comment cela est possible.

Réponse

L'explication la plus simple serait de comparer les deux temps d'exécutions pour trouver la valeur pour laquelle les nombres d'opérations primitives de deux algorithmes auront le même temps d'exécution.

Ceci nous demandera de résoudre l'équation : $n \log n = n^2$ qui n'est pas une chose facile. Par contre on peut tenter se souvenir des notions sur la notation O, on sait qu'un algorithme ayant comme fonction asymptotique $O(n \log n)$ implique que le nombre d'opération primitive peut être donné par une fonction $f(x)$ telle qu'il existe c et que $f(x) \leq c * n \log n$ (Cas d'alphonse), en appliquant la même chose pour le cas de BOB, on aura une fonction $g(x) \leq k * n^2$ ce qui nous ramène au cas de la question n°1 déjà traitée. Se basant au résultat obtenu ci-haut, on peut directement dire que L'algorithme de Alphonse est plus rapide que celui de bob. Et ce résultat prend plus de l'ampleur avec l'augmentation de la valeur de n.

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.

Réponse

```python
def maximalRecursif(sequence, k, l):
    if k == len(sequence):
        print("Le maximun de la sequence vaut :" +str(l))
        return l

    else:
        if k == 0:
            l = sequence[0]
        elif l < sequence[k]:
            l = sequence[k]
        print("Appel fonction recursive = " + str((k+1)))
        return maximalRecursif(sequence, k+1, l)

sequence = [10, 22, 25, 0, 57, 64, 35, 954, 224, 364, 7201, 1253, 125, 963, 539]
print("\n\n", sequence)
maximalRecursif(sequence, 1, 2)
```

Résultat de la récursivité

```
C:\Users\deogr\Documents\Mes docs\Fac\AlgoProgram\TP_Algo>python quest11.py


 [10, 22, 25, 0, 57, 64, 35, 954, 224, 364, 7201, 1253, 125, 963, 539]
Appel fonction recursive = 2
Appel fonction recursive = 3
Appel fonction recursive = 4
Appel fonction recursive = 5
Appel fonction recursive = 6
Appel fonction recursive = 7
Appel fonction recursive = 8
Appel fonction recursive = 9
Appel fonction recursive = 10
Appel fonction recursive = 11
Appel fonction recursive = 12
Appel fonction recursive = 13
Appel fonction recursive = 14
Appel fonction recursive = 15
Le maximun de la sequence vaut :7201
```

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

Réponse

```python
def find_min_max(liste, i=0):
    if i == len(liste)-1:
        return liste[i], liste[i] #The current min and max values
    else:
        minimum, maximum = find_min_max(liste, i+1)
        return min(liste[i], minimum), max(liste[i], maximum)

#Programme principal
maListe = [-20, 15, 12, -8, 25, 580, 260]
print(find_min_max(maListe))
```

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

Réponse

```python
listeVoy = ['a','e','i','o','u']

def nombre_voyelle(caractere):
    return caractere.lower() in listeVoy

def count(chaine, taille):
    if (taille==1):
        return nombre_voyelle(chaine[taille-1])

    return (count(chaine, taille-1) + nombre_voyelle((chaine[taille-1])))

nom = 'Semikenke'
nombre_voyelle = count(nom, len(nom))
nombre_consonne = len(nom)-nombre_voyelle
if (nombre_consonne-nombre_voyelle) != -1 :
    print("\n")
    print(f"Dans < {nom} >, Il y a {nombre_consonne-nombre_voyelle} consonnes de plus que les voyelles\n\n\n")
else:
    print("Votre mot n'est composé que d'une voyelle")
```

Résultat : 1 consonne de plus que les voyelles.