UNIVERSITE DE KINSHASA

FACULTE POLYTECHNIQUE

TP D'ALGORITHME ET PROGRAMATION

Réaliser par :

- ASIMWE NDRUUDJO GLOIRE 2GC
- IMANI UKURANGO CHRISTIAN 2GC
- HESHIMA MASIMIKE DAVID 2GM

Année Académique 2021 – 2022

1. **Le nombre d'opérations primitives exécutées par les algorithmes A et B est (50 n log n) et (45 n^2), respectivement. Déterminez n0 tel que A soit meilleur que B, pour tout n ≥ n_0.**

Réponse :

L'algorithme A est meilleur que l'algorithme B, si son temps d'exécution est inférieur par rapport au temps d'exécution de l'algorithme B. Donc :

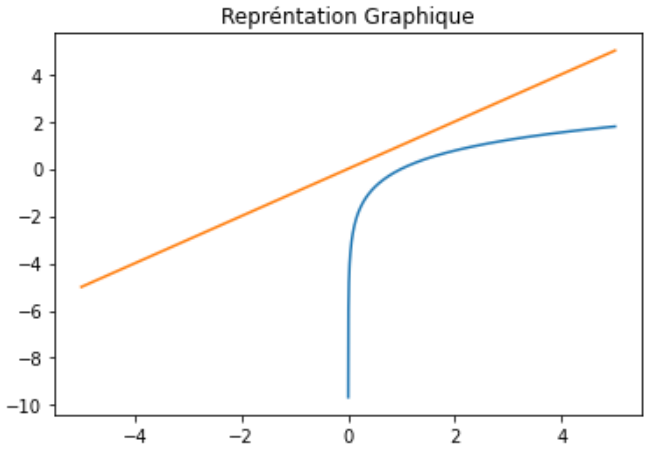
$$50\, n\log(n) < 45n^2 \;\Leftrightarrow\; n\log(n) < \frac{10}{9}$$

- **Code**

```
import numpy as np
import matplotlib.pyplot as plt
from math import*

x = np.linspace(-5,5, 30000)
y = (50/45)*np.log(x)

plt.plot(x, y)
plt.plot(x,x)
plt.title("Repréntation Graphique")
```

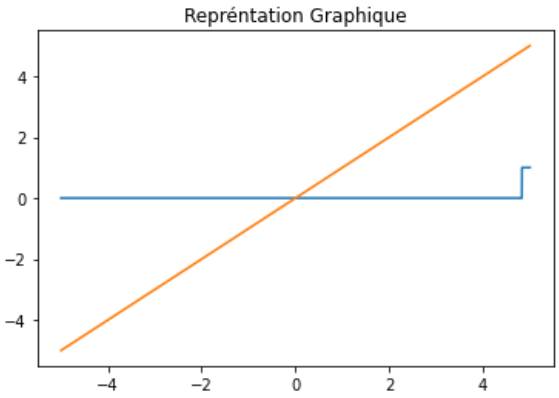


2. **Le nombre d'opérations primitives exécutées par les algorithmes A et B est (140 n^2) et (29 n^3), respectivement. Déterminez n_0 tel que A soit meilleur que B pour tout n ≥ n_0. Utiliser Matlab ou Excel pour montrer les évolutions des temps d'exécution des algorithmes A et B dans un graphique.**

Réponse :

L'algorithme A est meilleur que l'algorithme B, si son temps d'exécution est inférieur par rapport au temps d'exécution de l'algorithme B. Donc :

$$140n^2 < 29n^3 \;\Leftrightarrow\; 140 < 29n$$



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 :

Soit O(f(n)) le temps d'exécution du cas favorable ou défavorable, alors k*f(n) est supérieur au cas pire pour n >no avec k une constante, car théoriquement dans le pire de cas, on observe un temps d'exécution supérieur. On peut conclure que le pire de cas de l'algorithme A est aussi O(f(n)), vu que la forme asymptotique ne sera juste qu'un facteur multiplicatif de k du cas le plus favorable. Comme a >=b et b >=k, alors a >=k , ce qui signifie que k est 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) vaut O(f(n)), partant du raisonnement de la question précédente nous pouvons déduire facilement que *(a*d(n)) vaut O(f(n)).

5. M**ontrer 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)).**

Réponse :

Nous pouvons déduire une fois de plus des questions précédentes comme suit, si d(n) a comme expression de temps d'exécution O(f(n)) et e(n) O(g(n)), alors d(n)<= k*f(n) pour n_f>n_f0 et e(n)<= l*g(n) pour n_e>n_e0 Ceci veut dire que , d(n)*e(n) <= (c*f(n))*(d*g(n)) et n_f*n_e > n_f0*n_e0 ce qui signifie en d'autres termes qu'il existe une nouvelle constante n' = n_f*n_e et et n0' = n_f0*n_e0, et un k' = k*l tel que d(n)*e(n) <= k'(f(n)*g(n)) pour n'>n0', On conclut aussi que d(n)*e(n) est O(f(n)*g(n))

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 :

Une fois de plus si d(n) a comme expression asymptotique O(f(n)) et e(n) O(g(n)), alors d(n)<= k*f(n) pour n_f>n_f0 et e(n)<= h*g(n) pour n_e>n_e0. On peut déduire que d(n) + e(n) <= (k*f(n)) + (h*g(n)) et n > n_f0+n_e0, ce qui nous pousse à dire qu'il existe un nouveau n' = n_f+n_e et et n0' = n_f0+n_e0, tel que d(n) + e(n) <= k*f(n) + h*g(n) pour n>n0' ; En effet, on peut continuer en disant que d(n) + e(n) <= f(k*n)+ g(h*n), ce qui va nous conduire à 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 :

Soit d(n) et e(n) a pour notation asymptotique O(f(n)) , et O(g(n)) respectivement, Avec comme exemple d(n) = n et e(n) = n avec f(n) = n et g(n) = n, alors on peut dire d(n) <= k*(f(n)) pour n>=0, et e(n) <= 2*k*(g(n)) pour n>=0 f(n)- g(n) = 0 et d(n) - e(n) = n-n , pas de valeur pour n>0 telle que 0>=n, ce qui signifie que d(n) -e(n) n'est pas 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) vaut O(f(n)) et f(n) vaut O(g(n)), alors d(n)<= k*f(n) pour n_f>n_f0 et f(n)<= h*g(n) pour n_g>n_g0 Si c'est vrai, alors d(n)<=k*f(n)<=k*(d(g(n))) = k*d*g(n) = k'*g(n) par substitution, ce qui est valide pour n_f*n_g>n_f0*n_g0, ou n>n0'

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 :

Soit un algorithme constitué de deux blocs, D et E parcourant une séquence S de n éléments. Le temps d'exécution de E est O(i), il correspondra au temps d'exécution O(n) de l'algorithme E, car n est la taille de notre séquence, de ce fait le pire temps d'exécution de D est $O(n^2)$, vu que la séquence à n éléments sera appelé par l'algorithme D qui appellera à son tour l'algorithme E n fois de suite.

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(n2) 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(n2) 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 :

Soit f(n) < C*g(n) en tenant compte de notre résultat précédent. Si l'algorithme de Alphonse A(n*log(n)) fonctionne mieux que celui de Bob $B(n^2)$, On peut résoudre l'expression $n*\log(n)=n^2$ Le rapport entre $n*\log(n) / n^2$ donne 100/ 100log (100) = 15.5 s Ceci nous permet de conclure que l'algorithme d'alphonse est 15 fois plus lents sur une itération, mais ceci puisqu'il effectue moins d'opérations. Par contre plus l'algorithme d'Alphonse effectue beaucoup d'opérations, plus il est meilleur que celui de BOB.

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 :

L'algorithme pour trouver l'élément maximum peut bien évidement être rendu récursif. Dans cette tâche, l'algorithme doit examiner chaque élément de la liste, il n'y a donc qu'un seul cas de base : lorsque la fin de la liste est atteinte. Une entrée doit être ajoutée pour garder une trace de l'élément maximum entre les appels récursifs. Le cas récursif est presque identique à la recherche linéaire, il déplace simplement le processus vers l'élément suivant de la liste.

- **Code**

```
- def trouver_maximum(nombres):
-     if len(nombres) == 1:
-         return nombres[0]
-     else:
-         max_precedent = trouver_maximum(nombres[1:])
-         return nombres[0] if nombres[0] > max_precedent else
  max_precedent
-
- liste_nombres = [12, 25, 7, 45, 15, 6, 4]
- maximum = trouver_maximum(liste_nombres)
- print("le maximum est: ", maximum)
```

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

Réponse :

```
def trouver_min_max(sequence):
    if len(sequence) == 0:
        return None, None
    if len(sequence) == 1:
        return sequence[0], sequence[0]

    min_gauche, max_gauche = trouver_min_max(sequence[:len(sequence)//2])
    min_droite, max_droite = trouver_min_max(sequence[len(sequence)//2:])

    if min_gauche < min_droite:
        min_global = min_gauche
    else:
        min_global = min_droite

    if max_gauche > max_droite:
        max_global = max_gauche
```

```
    else:
        max_global = max_droite

    return min_global, max_global
```

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

Réponse :

```
def count_vowels(string):
    vowels = "aeiouAEIOU"
    if len(string) == 0:
        return 0
    elif string[0] in vowels:
        return 1 + count_vowels(string[1:])
    else:
        return -1 + count_vowels(string[1:])

def contains_more_vowels(string):
    count = count_vowels(string)
    return count > 0
```