UNIVERSITE DE KINSHASA

FACULTE POLYTECHNIQUE

DEPARTEMENT DE GENIE ELECTRIQUE ET INFORMATIQUE

TRAVAIL PRATIQUE DU COURS D'ALGORITHMIQUE ET PROGRAMMATION

Par :
LABULU IBAM DANNY (2GEI)
ONKETU ANTEMBA BENI (2GC)
KABANGU MWATA OLIVIER (2GC)

Dirigé par l'assistant **MOBISA**

Année académique 2021-2022

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

   R/ On peut trouver n_0 en comparant les deux expressions et en trouvant à partir de quelle valeur $(50\ n \log n)$ est plus petit que $(45n^2)$.

$$50\ n\ \log n \leq 45\ n^2$$

$$50\ \log n \leq 45\ n$$

$$\frac{50}{45} \log n \leq n$$

$$\frac{10}{9} \log n \leq n$$

   On sait que $\log n \leq n\ \forall\ n > 0$, prouvons par récurrence que $\frac{10}{9} \log n \leq n\ , \forall\ n > 0$.

   Si $n = 1, 0 \leq 1$.

   Si $n = 2, 0{,}3344 \leq 2$.

   SI $n = 3, 0{,}5301 \leq 3$.

   …

   Supposons que cela soit vrai jusqu'à l'ordre k, donc que $\frac{10}{9} \log k \leq k$.

   Montrons que cela est vrai jusqu'à l'ordre $k + 1$,

$$\frac{10}{9} \log(k+1) \leq k + 1$$

$$\log(k+1)^{\frac{10}{9}} \leq \log 10^{(k+1)}$$

   $(k+1)^{\frac{10}{9}} \leq 10^{(k+1)}$, pour égaler les bases afin de comparer on pose $k = 9$, on a

   $10^{\frac{10}{9}} \leq 10^{10}$, ce qui est vrai car $\frac{10}{9} \leq 10$, donc $(k+1)^{\frac{10}{9}} \leq 10^{(k+1)}$ est vrai.

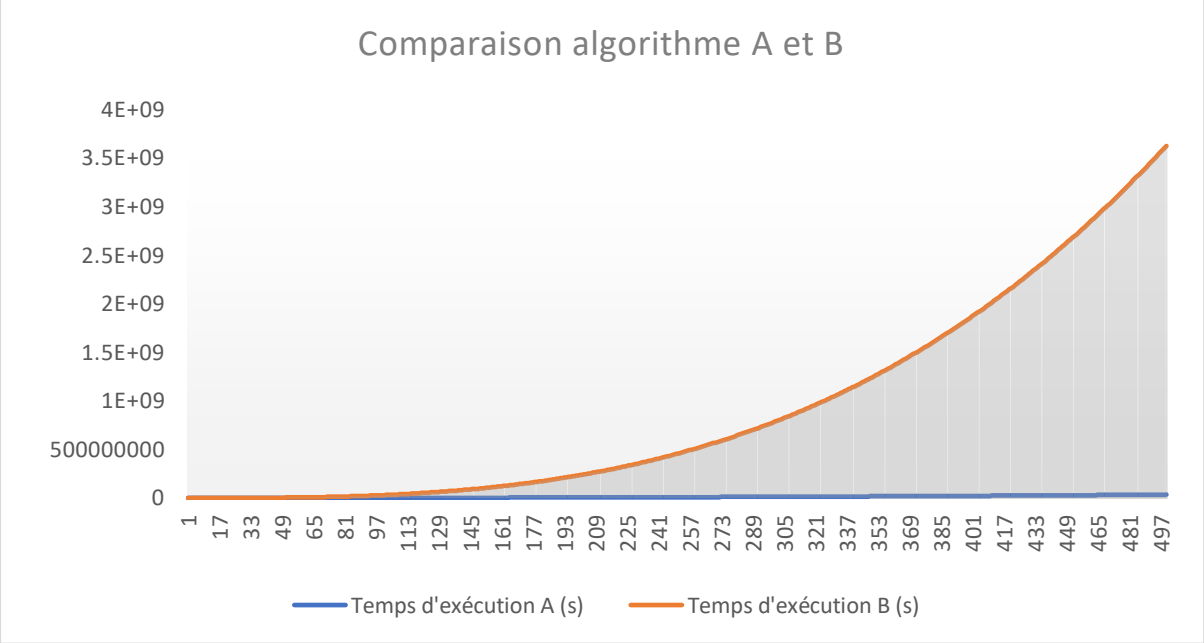
   On conclut donc que $\frac{10}{9} \log n \leq n, \forall\ n > 0$

   Donc notre $n_0 = 1$, ce qui veut dire que l'algorithme A sera toujours meilleur que l'algorithme B peu importe le nombre d'entrées.

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 ≥ n0. Utiliser Matlab ou Excel pour montrer les évolutions des temps d'exécution des algorithmes A et B dans un graphique.

   R/ En consultant le tableau du fichier Excel joint à notre travail, on se rend bien compte qu'à partir de $n = 5$, le temps d'exécution de l'algorithme A est inférieur à celui de l'algorithme 2

   En regardant le graphique, on se rend bien compte que le temps d'exécution de l'algorithme A reste bien inférieur à celui de l'algorithme B pour des valeurs supérieures.



   Donc, notre $n_0 = 5$.

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 $O(f(n))$.

   R/ Dans le pire de cas, le temps d'exécution est $O(f(n))$, ce qui signifie qu'il y a une constante c, tel que $c * f(n)$ est supérieur au pire de cas pour $n > n_0$.

   On sait que le temps d'exécution du pire cas est toujours supérieur ou égal aux autres cas, donc si leur temps d'exécution est par exemple $g(n)$ alors $c * f(n) \geq$ temps d'exécution du pire des cas qui lui est supérieur à tous les autres cas de l'algorithme A.

Donc on $a \geq b$ et $b \geq c$, alors $a \geq c$, ce qui signifie que le temps d'exécution de l'algorithme A est $O(f(n))$.

Ces deux énoncés sont équivalents car le temps d'exécution d'un algorithme se mesure toujours par rapport au pire des cas.

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

   R/ Si $d(n) = O\big(f(n)\big)$, alors $\exists\, c$ tel que $d(n) \leq c * f(n), \forall n > n_0$.

   En multipliant $d(n)$ par a, on a : $\big(a * d(n)\big) \leq a * c * f(n) = c' * f(n)$.

   La nouvelle constante est $c * a$ qui maintient la condition de départ de O, qui sera vrai quand $n > a * n_0$.

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 $O(f(n) * g(n))$.

   R/$d(n) = O(f(n))$ et $e(n) = O(g(n))$ alors $d(n) \leq c * f(n)$ pour $f(n) > f(n_0)$ et $e(n) \leq d * g(n)$ pour $e(n) > e(n_0)$.

   On a, $d(n) * e(n) \leq \big(c * f(n)\big) * \big(d * g(n)\big)$ et $f(n) * e(n) > f(n_0) * e(n_0)$

   Ce qui signifie qu'il y a $n' = f(n) * e(n)$ et $n_0' = f(n_0) * e(n_0)$, et $c' = c * d$ alors $d(n) * e(n) \leq c'(f(n) * g(n))$ pour $n' > {n_0}'$

   Donc $d(n) * e(n) = O(f(n) * g(n))$.

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)).

   R/ $d(n) = O(f(n))$ et $e(n) = O(g(n))$ alors $d(n) \leq c * f(n)$ pour $f(n) > f(n_0)$ et $e(n) \leq d * g(n)$ pour $e(n) > e(n_0)$.

   Donc $d(n) + e(n) \leq (c * f(n) + d * g(n))$ et $n > f(n_0) + e(n_0) \rightarrow n' = f(n) + e(n)$ et $n_0' = f(n_0) + e(n_0)$, alors

   $\rightarrow d(n) + e(n) \leq (c * f(n) + d * g(n))$ pour $n > {n_0}'$

   $\rightarrow d(n) + e(n) \leq f(c * n) + g(d * n)$ pour $n > {n_0}'$, on pose

   $n'' = c * n' \rightarrow n' = \frac{n''}{c} \rightarrow \frac{n''}{c} \geq \frac{{n_0}''}{c}$ et $\frac{n''}{cd} \geq \frac{{n_0}''}{cd}$.

   On obtient $d(n) + e(n) \leq f(n) + d(n)$ pour $n \geq {n_0}''$ qui respecte la condition $O(f(n) + d(n))$.

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)).

   R/ Si $d(n) = n$ et $e(n) = n$, avec $f(n) = n$ et $g(n) = n \rightarrow d(n) \leq C\big(f(n)\big)$ pour $n \geq 0$, et $e(n) \leq C(f(n))$ pour $n \geq 0$.

   $f(n) - g(n) = 0$ et $d(n) - e(n) = n$.

   Il n'y a aucune valeur pour $n > 0 \rightarrow 0 \geq n$ donc, $d(n) - e(n) \neq O(f(n) - g(n))$.

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

   R/ $d(n) = O\big(f(n)\big)$ et $f(n) = O(g(n))$, alors $d(n) \leq c * f(n)$ pour $f(n) > f(n_0)$ et $f(n) \leq d * g(n)$ pour $g(n) > g(n_0)$.

   Si c'est vrai, alors $d(n) \leq c * f(n) \leq c * (d * g(n))) = c * d * g(n) = c' * g(n)$ par substitution, ce qui est vrai pour $f(n) * g(n) > f(n_0) * g(n_0)$, ou $n > n_0$.

   Alors $d(n) = O\big(g(n)\big)$.

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?

   R/ L'algorithme D appelle l'algorithme E n fois, alors le temps d'exécution du pire des cas est $O\big(G(n)\big) * O(i))$, qui correspond à $O(n * 1) = O(n)$.

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.

    R/ La notation Osignifie qu'il y a une constante C tel que $f(n) < C * g(n)$.

    Si l'algorithme d'Alphonse est meilleur que $A * (n * \log n)$ et celle de Bob meilleur que $B * (n^2)$,

On peut résoudre l'équation au moment où $A * (n * \log n) = B * (n^2)$, on sait que cela est vrai quand $n = 100$ alors $\frac{A}{B} = \frac{(100)}{(\log 100)} = 15{,}055$.

Cela signifie que l'algorithme d'Alphonse est 15 fois plus lent pour une seule itération, mais il performe moins d'opérations, il commence à mieux former pour de grandes valeurs 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.

R/

- Conception de l'algorithme

ENTRÉE : UNE SÉQUENCE DE NOMBRE « SEQUENCE », UN ENTIER L'INDICE INITIALISÉ À 0 « I » ET LA VALEUR MAXIMUM INITIALISÉE À 0 « MAX ».

SORTIE : MAX TEL QUE POUR TOUT SEQUENCE[I], MAX ≥ SEQUENCE[I].

FONCTION MAXIMUM (SEQUENCE, I=0, MAX=0)

SI I < TAILLE(SEQUENCE) :

SI I == 0 OU SEQUENCE[I] > MAX :

MAX = SEQUENCE[I]

MAX = MAXIMUM (SEQUENCE, I+1, MAX)

RETOURNER MAX

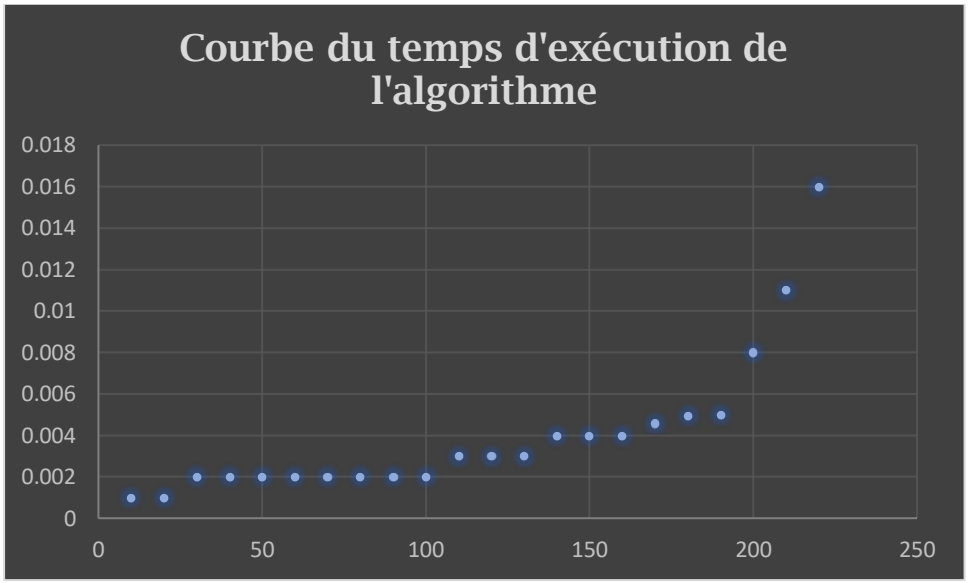
- Implémentation et mesure du temps d'exécution

```
import time
import random
def maximum(sequence, i=0, max=0):
if i < len(sequence):
if i == 0 or sequence[i] > max :
max = sequence[i]
max = maximum(sequence, i+1, max)
```

```
return max
liste = []
n = [130,140,150,160,170,180]
for r in n:
for x in range (r) :
liste.append(random.randint(1, 1000000))

y = time.time()
print (maximum(liste))
z = time.time()
print(b)
```

- Courbe du temps d'exécution obtenue avec Excel



En regardant la forme de la courbe on peut la comparer à la courbe associée à la fonction $O(n^n)$.

- Fonction du temps d'exécution en fonction des opérations primitives

  Pour la récursivité, le temps d'exécution sera $O(n)$ car il y n appels récursifs, qui prenne chacun $O(1)$ pour tourner.

- Comparaison des deux temps d'exécution

On se rend compte que pour la méthode des opérations primitives, le temps d'exécution considère 1 comme l'exposant de n tandis que la méthode expérimentale considère n comme son propre exposant.

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

R/

ENTRÉE : UNE SÉQUENCE DE NOMBRE « S », UN ENTIER L'INDICE INITIALISÉ À o « I ».
SORTIE : LE MINIMUM ET LE MAXIMUM DE LA SÉQUENCE DES NOMBRES.
FONCTION MINI_MAXI (S, I=o) :
SI I == TAILLE(S)-1 :
RETOURNER S[I], S[I]
SINON :
MINI_C, MAXI_C = MINI_MAXI (S, I+1)
RETOURNER MIN(S[I], MINI_C), MAX(S[I], MAXI_C)

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

R/

```
Voyelles = ['a','e','i','o','u','y']
Entrée : une séquence de lettres « m », un entier l'indice
des éléments de la séquence « i ».
Sortie : la présence ou non de plus de voyelles ou de
consonnes dans la séquence.
Fonction consonne_ou_voyelle (m, i=0) :
a = -1 si m[i] in voyelles sinon 1
si i == taille(m)-1 :
Retourner a
sinon :
retourner (a + consonne_ou_voyelle (m, i+1))
fonction vérifier_voyelles(m) :
réponse = consonne_ou_voyelle(m)
si réponse > 0 :
retourner ''il y a plus de consonnes''
et si réponse < 0 :
retourner ''il y a plus de voyelles''
sinon :
retourner ''il y a autant de consonnes que des voyelles''
```