Exponentiation rapide

1 Un premier algorithme

a désigne un nombre réel et n un entier naturel. Voici ci-dessous un algorithme qui permet de calculer a^n.

Par exemple pour calculer a^5, l'algorithme effectue les opérations $((((a \times a) \times a) \times a) \times a)$.

```
(1) Fonction puissance(a, n) :
(2)       p ← 1
(3)       k ← 0
(4)       Tant que k ≠ n
(5)           p ← p ∗ a
(6)           k ← k + 1
(7)       Fin Tant que
(8) Renvoyer p
```

1.1 Compléter la trace suivante :

On suppose que $a = 3$ et que $n = 5$:

ligne	p	k
(1)	?	?
(2)	1	?
(3)	1	0
(4)	1	0
(5)	3	0
(6)	3	1
(4)	.	.
(5)	.	.
(6)	.	.
(4)	.	.
(5)	..	.
(6)	..	.
(4)	..	.
(5)	..	.
(6)	..	.
(4)	..	.
(5)	...	.
(6)	...	.
(7)	...	.
(8)	...	.

1.2 Nombre d'opérations effectuées par l'algorithme

Pour compter le nombre d'opérations qu'effectue cet algorithme, on considère que toutes les opérations de calcul arithmétiques ($+$, $-$, $*$, $/$, $//$, $\%$) et booléènnes (*and* , *or* , *not*), d'affectation($=$) et de comparaison($<$, $>$, $\leq$, $\geq$, $==$, $!=$) ont le même coût.

On parle d'opérations élémentaires.

a) Expliquer pourquoi lorsque $n = 5$, le nombre d'opérations de cet algorithme est 17.

Réponse :

b) Evaluer le nombre d'opérations effectuées lorsque $n = 20$.

Réponse :

c) Quel est en fonction de n le nombre d'opérations effectuées ?

Réponse :

1.3 Lorsque n est très grand

Voici une implémentation dans le langage python de cet algorithme :

[6]:
```python
def puissance(a,n):
    p = 1
    k = 0
    while k != n:
        p = p * a
        k = k + 1
    return p

puissance(3,5) # Appel de cette fonction à cette ligne.
```

[6]:
```
243
```

A l'aide d'un chronomètre, mesurer le temps d'éxécution de ce programme lorsque l'on appelle cette fonction avec $a = 3$ et $n = 500000$, puis $n = 1000000$:

Temps approximatif d'éxécution pour $n = 500000$:

Temps approximatif d'éxécution pour $n = 1000000$:

1.4 Une représentation graphique.

a) L'instruction " %timeit " mesure le temps d'exécution d'une instruction ou d'une expression. Le code est exécuté plusieurs fois et le temps moyen d'exécution est affiché.
Exécuter le code ci-dessous en changeant la valeur de n à chaque fois et noter les résultats ci-après :

[ ]:
```python
%timeit puissance(3,10000)
```

Valeur de n : 10000
Temps d'éxécution :

Valeur de n : 100000
Temps d'éxécution :

Valeur de n : 500000
Temps d'éxécution :

b) Exécuter le code ci-dessous, et soyez patients... Cela vous semble-t-il cohérent avec tous les résultats précédents ?

Réponse :

[ ]:
```python
import timeit
%matplotlib inline
import matplotlib.pyplot as plt

valeurs_n = list(range(1,21,1)) + list(range(20,1020,20))
temps1_n = []

for n in valeurs_n :
    temps1_n.append(min(timeit.Timer('''from __main__ import puissance ; puissance(3,{})'''.format(n)).repeat(repeat=5, number = 1000))/1000)

plt.rcParams.update({'font.size': 16})
plt.figure(figsize=(20, 6))
plt.plot(valeurs_n , temps1_n , alpha=0.8, marker='o', lw=3)
plt.xlabel('Exposant n')
plt.ylabel('Temps de calcul (ms)')
plt.grid()
plt.show()
```

2 Un deuxième algorithme

Voici un autre algorithme qui permet de calculer le nombre a^n :
Cette fois, pour calculer a^5, cet algorithme effectue les calculs $(a \times (a \times a)^2))$

Fonction puissance_rapide(a, n) :
    $p \leftarrow 1$
    $b \leftarrow a$
    $m \leftarrow n$
    Tant que $m > 0$
        Si $m\%2 = 1$
            $p \leftarrow p * b$
        $b \leftarrow b * b$
        $m \leftarrow m//2$
    Fin Tant que
Renvoyer p

2.1 Compléter les traces suivantes :

a) On suppose que $a = 3$ et que $n = 5$:

ligne	p	b	m
(1)	?	?	?
(2)	1	?	?
(3)	1	3	?
(4)	1	3	5
(5)	1	3	5
(6)	1	3	5
(7)	3	3	5
(8)	3	9	5
(9)	3	9	2
(5)	.	.	.
(6)	.	.	.
(8)	.	..	.
(9)	.	..	.
(5)	.	..	.
(6)	.	..	.
(7)	.	..	.
(8)	...	....	.
(9)	...	....	.
(10)	...	....	.
(11)	...	....	.

b) On suppose que $a = 3$ et que $n = 6$:

ligne	p	b	m
(1)	?	?	?
(2)	1	?	?
(3)	1	3	?
(4)	1	3	6
(5)	.	.	.
(6)	.	.	.
(8)	.	.	.
(9)	.	.	.
(5)	.	.	.
(6)	.	.	.
(7)	.	.	.
(8)	.	..	.
(9)	.	..	.
(5)	.	..	.
(6)	.	..	.
(7)	...	..	.
(8)	...	....	.
(9)	...	....	.
(10)	...	....	.
(11)	...	....	.

c) On suppose que $a = 3$ et que $n = 8$:

ligne	p	b	m
(1)	.	.	.
(2)	.	.	.
(3)	.	.	.
(4)	.	.	.
(5)	.	.	.
(6)	.	.	.
(8)	.	.	.
(9)	.	.	.
(5)	.	.	.
(6)	.	.	.
(8)	.	..	.
(9)	.	..	.
(5)	.	..	.
(6)	.	..	.
(8)	.	....	.
(9)	.	....	.
(5)	.	....	.
(6)	.	....	.
(7)	....	....	.
(8)	....	....	.
(9)	....	....	.
(10)	....	....	.
(11)	....	....	.

2.2 Nombre d'opérations

Evaluer pour $n = 5$ le nombre d'OPEL réalisées par cet algorithme.

Réponse :

2.3 Lorsque n est très grand

Voici une implémentation de cet algorithme en langage python.

[8]:
```
def puissance_rapide(a,n):
    p = 1
    b = a
    m = n
    while m > 0 :
        if m % 2 == 1 :
            p = p * b
        b = b * b
        m = m // 2
    return p

puissance_rapide(3,5) # Appel de cette fonction à cette ligne.
```

[8]:
```
243
```

A l'aide d'un chronomètre, mesurer le temps d'éxécution de ce programme lorsque l'on appelle cette fonction avec $a = 3$ et $n = 500000$, puis $n = 1000000$.
Temps approximatif d'éxécution pour n = 500000 :
Temps approximatif d'éxécution pour n = 1000000 :

2.4 Une représentation graphique

a) Exécuter le code ci-dessous, et soyez patients…

[9]:
```
temps2_n = []

for n in valeurs_n :
    temps2_n.append(min(timeit.Timer('''from __main__ import puissance_rapide ;␣
 ↪puissance_rapide(3,{})'''.format(n)).repeat(repeat=5, number = 1000))/1000)
```

b) Exécuter le code ci-dessous et comparer les deux courbes. On pourra faire varier les paramètres dans la ligne plt.ylim (échelle de l'axe des ordonnées).

[ ]:
```
plt.rcParams.update({'font.size': 16})
plt.figure(figsize=(20, 6))
plt.ylim([0.0,0.00035])
plt.plot(valeurs_n , temps1_n , label='Algorithme 1' , alpha=0.8, marker='o',␣
 ↪lw=3)
plt.plot(valeurs_n , temps2_n , label='Algorithme 2' , alpha=0.8, marker='o',␣
 ↪lw=3)
plt.xlabel('Exposant n')
plt.ylabel('Temps de calcul (ms)')
plt.legend(loc=4)
plt.grid()
plt.show()
```

c) Interprétez les résultats obtenus.
Réponse :

3 Complexité

- Pour mesurer l'efficacité d'un algorithme, on peut étudier son temps d'éxécution (on pourrait aussi s'intéresser à l'espace mémoire dont il a besoin).
- Pour estimer le temps d'éxécution, indépendemment des performances de la machine ou du langage de programmation utilisé, on compte le nombre d'opérations élémentaires effectuées par l'algorithme.
- Plutôt que de déterminer exactement ce nombre, on cherche un ordre de grandeur.
- Cet ordre de grandeur dépend de la taille de l'entrée (ici de la valeur de l'exposant).
- On parle de complexité :
  - Celle du premier algorithme est de l'ordre de n (notée $\mathcal{O}(n)$), elle est dite linéaire.
  - Celle du second algorithme est de l'ordre du logarithme en base 2 de n (notée $\mathcal{O}(log_2(n))$, du nom d'une fonction que vous étudierez en terminale.

Exemple : Pour calculer 3^{100000}

- Le premier algorithme effectue un nombre d'opérations de l'ordre de 100000.
- Le deuxième algorithme effectue un nombre d'opérations de l'ordre de $log_2(100000) \simeq 17$.

3.1 Combien d'opérations élémentaires faut-il à l'algorithme 1 pour calculer $5^{3000000}$?

Réponse :

3.2 Même question avec l'algorithme 2.

Réponse :

4 Quelques ordres de grandeurs

Pour aller plus loin, quel algorithme utilise la fonction math.pow ?

[11]:
```
temps3_n = []

for n in valeurs_n:
    temps3_n.append(min(timeit.Timer('''from math import pow ; pow(2,{})'''.
 ↪format(n)).repeat(repeat=5, number=1000))/1000)
```

Et l'opérateur $**$?

[12]:
```
temps4_n = []

for n in valeurs_n:
    temps4_n.append(min(timeit.Timer('''3**{}'''.format(n)).repeat(repeat=5,␣
 ↪number=1000))/1000)
```

Comparons les courbes :

[ ]:
```
plt.rcParams.update({'font.size': 16})
plt.figure(figsize=(20, 6))
plt.ylim([0.0,0.00035])
plt.plot(valeurs_n , temps1_n , label='Algorithme 1' , alpha=0.8, marker='o',␣
 ↪lw=3)
plt.plot(valeurs_n , temps2_n , label='Algorithme 2' , alpha=0.8, marker='o',␣
 ↪lw=3)
plt.plot(valeurs_n , temps3_n , label='math.pow' , alpha=0.8, marker='o', lw=3)
plt.plot(valeurs_n , temps4_n , label='**' , alpha=0.8, marker='o', lw=3)
plt.xlabel('Exposant n')
plt.ylabel('Temps de calcul (ms)')
plt.legend(loc=4)
plt.grid()
plt.show()
```

4.1 Cette représentation graphique semble-t-elle facile à interpréter ?

Réponse :

4.2 Proposer une modification du code puis le tester.

Réponse :