Université Clermont Auvergne
BUT Info 3A
Qualité algorithmique

Florent Foucaud, Pascal Lafourcade et Malika More
2024-2025

TP1 - Complexité, quelques exemples

Exercice 1 (Fibonacci)**.**

On rappelle la définition récursive du $n$ème nombre de Fibonacci $F(n)$ permettant d'approximer la taille de la $n$ème génération d'une population de lapins :

- Si $n = 0$ ou $n = 1$, $F(n) = 1$
- Si $n > 1$, $F(n) = F(n-2) + F(n-1)$

1. Implémenter un programme récursif `FiboRec(n)` qui calcule le $n$ème nombre de Fibonacci.
2. Implémenter un programme itératif `FiboIter(n)` qui calcule le $n$ème nombre de Fibonacci.
3. Tester pour diverses valeurs de n, par exemple, entre 1 et 50. Qu'observe-t-on ?
4. Calculer $F(n)$ avec les deux algorithmes pour des valeurs de n croissantes (essayer de 1 à 10, de 1 à 30, de 1 à 40), et mesurer les temps d'exécution grâce à la fonction `time.time()` (nécessite `import time`). Pour des résultats plus fiables, on peut répéter chaque calcul 2 ou 3 fois et faire la moyenne. Tracer les deux courbes de temps avec la méthode suivante :

```
import matplotlib.pyplot as plt
import time

# on suppose que les valeurs des temps sont stockées dans
# deux listes T0 et T1, chacune de la même taille

plt.title("Calcul des nombres de Fibonacci")
plt.xlabel("Valeur de n")
plt.ylabel("Durée en secondes")
plt.plot(T0,"blue")
plt.plot(T1,"red")
plt.show()
```

5. Calculer Fibonacci avec le nombre d'or $\varphi = \frac{1+\sqrt{5}}{2}$ avec la formule suivante :

$$F(n) = \frac{1}{\sqrt{5}}(\varphi^n - \varphi'^n), \text{ avec } \varphi' = \frac{1-\sqrt{5}}{2} = -\frac{1}{\varphi}$$

Exercice 2 (Ackermann)**.**

La *fonction d'Ackermann* $A(m, n)$ est définie récursivement de la façon suivante :

- $A(m, n) = n + 1$ si $m = 0$
- $A(m, n) = A(m-1, 1)$ si $m > 0$ et $n = 0$
- $A(m, n) = A(m-1, A(m, n-1))$ sinon

C'est la première fonction qui ait été découverte (par Wilhelm Ackermann), et qui n'est pas *primitive récursive*, c'est-à-dire qu'elle ne peut pas être calculée de façon itérative par des boucles imbriquées de longueur fixée à l'avance.

1. Dessiner sur feuille, les arbres des appels récursifs pour $A(1, 1)$ et $A(2, 2)$.

2. Implémenter un programme récursif `Ackermann(m,n)` qui calcule $A(m,n)$.
3. Remplir le tableau suivant avec les valeurs de $A(m,n)$. Les deux première lignes sont faciles à remplir à la main, le reste à l'aide du programme de la question précédente. Qu'observe-t-on ?
   Remarque : Vous pouvez booster la profondeur de pile de votre programme Python à l'aide des instructions suivantes :
   ```
   import resource,sys
   resource.setrlimit(resource.RLIMIT_STACK, (2**29,-1))
   sys.setrecursionlimit(10**6)
   ```

$A(m,n)$	0	1	2	3	4	5	6	7	8	9	10	11	12
0													
1													
2													
3													
4													

4. Pourrait-on accélerer le calcul avec la même méthode qu'à l'exercice 1 (Fibonacci) pour éviter de re-calculer plusieurs fois la même chose ? Quelle est la limite de cette approche ?

Exercice 3 (Exponentiation modulaire rapide)**.**

1. Écrire une fonction `exp_basique(x,n,p)` qui, pour trois entiers positifs x, n, p donnés en entrée, calcule de manière basique (sans utiliser la bibliothèque `math`) l'exponentielle modulaire

$$x^n \bmod p$$

   . Faire `(x**n)%p` n'est pas un bonne idée, expliquer pourquoi.
2. L'algorithme d'*exponentielle rapide* permet également de calculer le nombre

$$x^n \bmod p$$

   mais en utilisant l'algorithme récursif suivant :

$$\text{puissance}(x,\, n) = \begin{cases} x, & \text{si } n = 1 \\ \text{puissance}(x^2,\, n/2), & \text{si } n \text{ est pair} \\ x \times \text{puissance}(x^2,\, (n-1)/2), & \text{si n est impair} \end{cases}$$

   Écrire une fonction `exp_rapide(x,n,p)` implémentant cet algorithme.
3. Calculer le nombre suivant, avec les deux méthodes : [1]

$$123456789^{123456789} \bmod 987654321$$

   Que constatez-vous en terme de temps d'exécution ?

1. Cela vaut 598987215.

Exercice 4 (Syracuse)**.**
Une suite de Syracuse est définie par récurrence à partir d'un entier de départ $u_0 \geq 1$ et telle que, pour tout $n \geq 0$:

$$u_{n+1} = \begin{cases} \dfrac{u_n}{2}, & \text{si } u_n \text{ est pair} \\ 3u_n + 1, & \text{si } u_n \text{ est impair} \end{cases}$$

Autrement dit, quand le nombre est pair, il faut le diviser par deux, et quand il est impair, il faut le multiplier par trois et lui ajouter un. Le résultat est encore un nombre entier, sur lequel il faut répéter la procédure.

Par exemple, en partant de $u_0 = 7$, les valeurs suivantes sont $u_1 = 22$, $u_2 = 11$, $u_3 = 34$, $u_4 = 17$, $u_5 = 52$, $u_6 = 26$, $u_7 = 13$, $u_8 = 40$, $u_9 = 20$, $u_{10} = 10$, $u_{11} = 5$, $u_{12} = 16$, $u_{13} = 8$, $u_{14} = 4$, $u_{15} = 2$, $u_{16} = 1$, $u_{17} = 4$, $u_{18} = 2$, $u_{19} = 1$, $u_{20} = 4$, etc. Le calcul peut se poursuivre, mais les valeurs sont devenues périodiques, et le cycle 4 - 2 - 1 se répète indéfiniment.

En faisant varier la valeur de départ u_0, la suite des valeurs obtenues est bien entendu différente.

Le *temps de vol* d'un entier u_0 est le nombre n d'étapes du calcul pour arriver pour la première fois à $u_n = 1$. Les valeurs suivantes sont alors toujours périodiques, selon le cycle 1 - 4 - 2. Ainsi, le temps de vol de 7 est 16.

L'*altitude maximale* d'un entier est le plus grand nombre obtenu pendant son vol. Ainsi, l'altitude maximale de 7 est 52.

Il n'y a pas de méthode générale pour déterminer à l'avance le temps de vol et l'altitude maximale d'un entier. Ainsi, la durée de vol pour 77 671 est de 231 et son altitude maximale est de 1 570 824 736. De même, la suite qui débute avec le nombre 2 361 235 441 021 745 907 775 a pour durée de vol 2 284.

Question : Écrire un programme Python permettant de calculer le plus petit entier dont le temps de vol est supérieur à 100.