Récursivité (suite), diviser pour régner

Comparaison expérimentale de l'efficacité des tri fusion et tri à bulles

(a) Programmer en Python le tri fusion étudié en cours. Vous ferez dans un premier temps la fonction `fusion(A,B)` fusionnant deux listes triées A et B, puis la fonction récursive `triFusion(L)` triant une liste L donnée en entrée par la méthode du tri fusion.

(b) Comparer expérimentalement la vitesse du tri à bulles avec le tri fusion. Pour cela vous tirerez aléatoirement des tableaux de taille croissante (par exemple $1000, 2000, \ldots, 10000$), puis tracerez les deux courbes correspondant au temps d'exécution du tri à bulles et du tri fusion sur ces tableaux successifs.

- Pour mesurer le temps, une méthode consiste à récupérer l'heure avec `time.perf_counter()` [1] qui renvoie l'heure courante avant et après la procédure dont on veut mesurer le temps d'exécution puis faire la différence (il faut pour cela mettre `import time` en début de programme) ;
- On peut tracer plusieurs courbes avec la librairie `matplotlib`. Par exemple :

```
import matplotlib.pyplot as plt

lx=range(10)
ly1=range(10)
ly2=[i**2 for i in range(10)]
plt.plot(lx,ly1) # [x1,x2,...xn],[y1,y2...yn]
plt.plot(lx,ly2) # [x1,x2,...xn],[y'1,y'2...y'n]
plt.show()
```

Remarques sur la mesure expérimentale du temps d'exécution :

- Pour mesurer le temps mis par l'algorithme A sur l'entrée x, il peut être intéressant de répéter le calcul de A sur x un certain nombre de fois et de prendre le *minimum* des temps d'exécution mesurés. En effet, il se peut que d'autres processus (liés au système) ralentissent certaines des exécutions.
- Pour mesurer le temps t_n mis par l'algorithme A sur les entrées de taille n dans le pire cas, il peut être utile de tirer au hasard un certain nombre d'entrées de taille n, de mesurer le temps mis par l'algorithme A sur chacune de ces entrées par la méthode ci-dessus, puis de prendre pour pour t_n le maximum des temps obtenus. Ceci est utile pour la version optimisée du tri à bulles (celle qui ne fait pas de passage inutile) puisque le nombre d'étapes de l'algorithme dépend fortement du tableau lui-même et pas uniquement de sa taille (en revanche, le nombre d'étapes de la première version du tri à bulles ou du tri fusion ne dépend essentiellement que de la taille de l'entrée).

1. Ou la fonction `time.clock()` sur les versions plus anciennes de Python.

Exponentiation rapide

(a) Écrire une fonction récursive `expoNombre(a,n)` prenant en entrée un nombre a et un entier naturel n et retournant a^n, par la méthode d'exponentiation rapide étudiée en TD.

Dans la suite, on va calculer le n-ième terme de la suite de Fibonacci en faisant peu d'opérations arithmétiques, par la méthode vue en TD consistant à calculer une puissance de matrice par la méthode d'exponentiation rapide. On code une matrice $n \times p$ par une liste de n listes, chacune composée de p éléments.

(b) Si cela n'a pas été fait avant, écrire une fonction retournant le produit de deux matrices de tailles compatibles.

En Python, on peut donner une fonction en paramètre d'une fonction. Plutôt que d'écrire une nouvelle fonction d'exponentiation rapide à chaque fois que les objets, et donc la fonction $(a, b) \mapsto a \cdot b$, changent, on peut écrire une fonction d'exponentiation rapide plus générale qui prend cette fonction produit en paramètre.

(c) Écrire une fonction récursive `expo(a,n,produit,e)` prenant en entrée un objet a, un entier naturel n, une fonction produit et l'élément neutre pour le produit e, et retournant a^n.

(d) Tester cette fonction en donnant en paramètre une fonction produit retournant le produit de deux nombres.
Remarque : On peut aussi ne pas définir une nouvelle fonction mais juste donner en paramètre `lambda x,y: x*y`, ce qui revient à définir cette fonction produit sans lui donner de nom.

(e) Écrire une fonction `fiboRapide(n)` retournant l'élément d'ordre n de la suite de Fibonacci.