TP3 : Diviser pour régner

Le but de ce TP est d'implanter des algorithmes de type « diviser pour régner » vus en cours. On fournit une trame de code pour les deux premiers exercices dans les fichiers `TP4Exo1TriFusion.py` *et* `TP4Exo2Rang.py`.

Exercice 1. *Tri fusion*

Le but de cet exercice est d'implanter le tri fusion, comme étudié en cours. *À chaque question, testez impérativement ce que vous venez de faire sur un petit tableau tiré au hasard (choix 1) et vérifiez que le résultat est correct !*

1. Regarder la structure générale du code, les deux options d'exécutions, les deux tris à implémenter et la procédure mise en place pour comparer leur temps de calcul.

   ▷ Noter l'usage de la fonction `list()` dans le code. Si $T1$ est un tableau, alors `T2=T1` crée un tableau $T2$ qui est un alias sur le tableau $T1$, c'est juste un autre nom, un pointeur en clair. Tout changement sur $T2$ sera effectif sur $T1$. Ce n'est pas le cas avec la commande `T2=list(T1)` qui crée un nouveau tableau $T2$ et recopie les valeurs de $T1$ dedans. Un changement sur $T2$ n'affecte alors pas le tableau $T1$.

2. Compléter la fonction `TableauAuHasard(n)` qui remplit un tableau de taille n avec des entiers à choisir au hasard entre 1 et 10000.
   Décommenter la ligne `#Tab=TableauAuHasard(n)`.
3. Compléter la fonction `TriFusion(n,T)`. Ne pas hésiter à faire des affichages pour vérifier que le découpage de tableau se passe bien.
4. Compléter la fonction `Fusion(n1,n2,T1,T2,T)` qui fusionne les tableaux `T1` et `T2` de tailles respectives `n1` et `n2` dans le tableau `T`.
5. Compléter la fonction `TriBulles(n,T)`, qui trie le tableau `T` à l'aide d'un tri à bulles.

   ▷ L'instruction `pass` est une instruction vide, qui permet ici d'éviter d'avoir une fonction vide (avant que vous ne la complétiez), ce que Python ne veut pas...

6. Comparer les temps mis par le tri fusion et le tri à bulle. Ça vaut le coup d'avoir un bon algorithme tout de même...

Exercice 2. *Calcul de rang*

Le but de cet exercice est d'implémenter un calcul de rang linéaire, comme étudié en cours. *Tester vos programmes sur l'exemple fixe proposé avant de les faire tourner sur des tableaux aléatoires de tailles plus importantes.*

1. On propose trois choix possibles pour `choixPivot` : un pivot fixe (`T[0]`), un pivot qui assure que $n_{\text{inf}} \leq \lceil 3n/4 \rceil$ et $n_{\text{sup}} \leq \lceil 3n/4 \rceil$, comme dans le cours, et un pivot aléatoire. Compléter la fonction `int ChoixPivot(n,T,typepivot)` pour implanter ces différents choix de pivot. Le paramètre `typepivot` permet de savoir quel type de pivot sera choisi. Les variables `nappelChoixPivot` et `npivot` comptent respectivement le nombre d'appels à la fonction `ChoixPivot` et le nombre de pivots effectués.

   ▷ Le mot clé `global` permet d'utiliser une variable dans une fonction qui est déclarée en dehors de la fonction. Sans ce mot clé, Python redéclarerait une variable différente localement avec le même nom que la variable globale.

2. La tâche suivante est l'implantation de la fonction `Rang(n,k,T,typepivot)` permettant de trouver le $k^{\text{ème}}$ rang du tableau `T` de taille `n`. Compléter le calcul de `Tinf` et `Tsup`, et les appels récursifs (*Attention au rang passé dans l'appel récursif, il ne faut pas se tromper dans les calculs... Faîtes des petits exemples/tests/dessins...)*
3. Dans le cas de choix du pivot avec répétition de tirage (qui assure que $n_{\text{inf}} \leq \lceil 3n/4 \rceil$ et $n_{\text{sup}} \leq \lceil 3n/4 \rceil$), on a argumenté en cours que le nombre de tirage moyen par appel à la fonction choixPivot était de 2. Vérifier sur quelques exemples que cela est crédible. Tracer des courbes permettant de comparer les temps de calcul et les nombres de choix de pivots effectués : voit-on une différence sur un tableau aléatoire ? sur un tableau initialement trié (ordre croissant ou décroissant) ? (*Attention ! Aucune trame de code n'est prévue pour cette question...*)

4. *(bonus)* Pour aller plus loin : écrire une fonction `rang-par-tri(k,n,T)` qui calcule le $k^{\text{ème}}$ rang du tableau `T` en le triant (avec `tri-fusion` par exemple) et affiche sa valeur. Comparer les temps d'exécution de `rang` et de `rang-par-tri`.

Exercice 3. *Multiplication d'entiers (bonus)*

L'objectif de cet exercice, pour lequel aucune trame de code n'est fournie, et d'implanter la multiplication d'entiers par l'algorithme naïf et celui de Karatsuba et de comparer leurs performances. Pour cela, on se servira d'entiers comme encodés par Python, **mais on n'utilisera la multiplication $*$ de Python que pour multiplier des entiers compris entre 0 et 9 ou pour multiplier par une puissance de 10**. Par contre, on s'autorisera à sommer avec + n'importe quels entiers.

Il vous faudra implanter (au minimum) les fonctions suivantes :

1. `mult-classique-1-chiffre(x,c)` qui effectue la multiplication de l'entier `x`, qui est quelconque, et de l'entier `c` qui est compris entre 0 et 9.
2. `mult-classique(x,y)` qui effectue la multiplication classique des entiers `x` et `y`.
3. `mult-kara(x,y)` qui effectue le produit des entiers `x` et `y` *via* l'algorithme de Karatsuba.
4. Comparer les temps de calcul respectifs `mult-classique` et `mult-kara`, en traçant les temps d'exécution en fonction du nombre de chiffres des entiers passés en paramètre.