TP n°3

Récursivité

Vous ecrirez les methodes dans un même chier, et vous les testerez dans la methode `main`.
Les questions precedees d'(**) ne sont a faire que si vous avez prealablement ni le sujet ou si vous pensez pouvoir les faire en moins de 5 minutes.

Exercice 1 *Mise en bouche*
On rappelle qu'un algorithme est dit *recursif* s'il s'appelle lui-même.

1. Ecrire de deux manieres di erentes (imperative et recursive) la fonction factorielle.
2. Ecrire de deux manieres di erentes le calcul naïf de x^n.
3. Reecire le `pgcd` vu dans le TD3 de maniere recursive.
4. Reecrire la methode `int produit(int x, int y)` vu dans le TD3 de maniere recursive.
5. En suivant le même schema, ecrire le calcul de x^n de maniere plus e cace.
   (**) En quoi est-ce plus e cace ?
6. Ecrire la methode `boolean rechercheAux(int x, int[] tab, int a, int b)` qui recherche de maniere dichotomique si l'element x est dans un tableau d'entier trie tab entre les indices a et b de maniere recursive.
   Completer le programme par une methode `boolean recherche(int x, int[] tab)` qui recherche x dans le tableau trie tab en appellant la methode auxiliaire `rechercheAux`.
   (**) Combien y a t-il d'appels a `rechercheAux` pour un tableau de taille n ?

Exercice 2 *Tri selection*
Dans cet exercice nous verrons une version recursive du tri par selection vu en cours.

1. Ecrire une methode (imperative ou recursive) `int minimum(int[] tab, int a)` qui recherche le plus petit element du tableau d'entier tab a partir de l'indice a et renvoie son indice.
2. Ecrire une methode `void triAux(int[] tab, int a)` qui permute l'entier d'indice a avec le minimum a partir de l'indice a, puis s'appelle elle même pour trier le tableau tab a partir de l'indice a + 1.
3. Completer votre programme par d'une methode `void tri(int[] tab)` qui trie tab en appelant la methode auxiliaire `triAux`.

Exercice 3 *Recursivite mutuelle (ou recursivite croisee)*
La recursion mutuelle (ou recursion croisee) est une recursion ou deux fonctions ou plus sont de nies l'une en termes des autres.

1. Ecrire deux methodes `boolean estPaire(int i)` et `boolean estImpaire(int i)` mutuellement recursives qui repondent si i est respectivement pair et impair, sans utiliser de division (ni /, ni %).

2. Faire deux methodes `boolean verifieP(int i)` et `boolean neVerifiePasP(int i)` pour la propriete P sachant que :
   { les nombres impaires veri ent la propriete P,
   { si i veri e la propriete P alors $i * 2$ ne la veri e pas,
   { si i ne veri e pas la propriete P alors $i * 2$ la veri e,
   { P n'est pas bien de nie en 0, on considerera arbitrairement que 0 veri e la propriete P.
3. (**) A quoi correspond la propriete P ?

Exercice 4 *Suite de Fibbonacci*
On rappelle la de nition de la suite de Fibonacci :

$$F_0 = 0 \qquad F_1 = 1$$
$$F_{n+2} = F_{n+1} + F_n \qquad \text{pour } n \geq 0$$

1. Ecrire une methode recursive `static int Fibonacci(int n)` qui calcule le terme F_n.
2. Y ajouter une instruction pour que la methode a che le message `\calcul de F(n)"` au debut de son execution. Lorsqu'elle sera lancee, on verra alors a l'ecran comment se suivent les appels recursifs.
3. Arbre des appels : il represente la maniere dont les appels recursifs sont imbriques : l'appel de `Fibonacci(n)` provoque les appels de `Fibonacci(n-1)` et `Fibonacci(n-2)`, qui eux-mêmes provoquent les appels de ....

   Modi er la methode pour qu'elle a che non seulement les appels recursifs, mais aussi cette structure d'imbrication. L'appel de `Fibonacci(4)` provoquera par exemple l'a chage suivant (qui peut legerement varier suivant votre algorithme) :

   ```
   Calcul de F(4)                F(4) appelle
   -Calcul de F(3)                   F(3) qui appelle
   --Calcul de F(2)                       F(2) qui appelle
   ---Calcul de F(1)                            F(1)
   ---Calcul de F(0)                            et F(0)
   --Calcul de F(1)                       et F(1)
   -Calcul de F(2)                   et F(2) qui appelle
   --Calcul de F(1)                       F(1)
   --Calcul de F(0)                       et F(0)
   ```

   Pour cela la methode a chera le message `\---...-Calcul de F(n)"` au debut de son execution, ou le nombre de tirets est la profondeur de l'appel depuis l'appel initial (*indication : ajouter un argument a la methode, indiquant le nombre de tirets a a cher*).
4. En observant le resultat, on se rend compte que de nombreux calculs sont executes plusieurs fois. Combien y-a-t'il d'appels a `F` pour calculer F_n.
5. On peut eviter l'explosion exponentielle du nombre d'appels. A l'aide d'une methode recursive `Fibo_econome(int n)` renvoyant le couple (F_n, F_{n+1}), un seul appel recursif est necessaire.

   Pour cela, creer une petite classe `Couple` contenant uniquement 2 champs `public`, et permettant de representer des couples d'entiers (ecrivez cette classe dans votre chier `Prog.java`, a côte de la classe `Prog`).

   Puis ecrire une methode recursive `static Couple Fibo_econome(int n)` renvoyant un objet de classe `Couple` representant (F_n, F_{n+1}).

6. Y ajouter ce qu'il faut pour a cher les appels recursifs. Et constater l'economie de calculs.
7. (**) Voyez vous une maniere plus rapide pour calculer le terme F_n ? (*indication : cette methode utilise le calcul matriciel*)