Nom : _____

Prénom : _____

Numéro étudiant : | | | | | | | | |

Programmation récursive

9 février 2021

A lire absolument :

1. L'objectif n'est pas d'apprendre la correction par cœur, mais de comprendre les mécanismes mis en œuvre. Cela vous permettra de vous adapter face à un problème nouveau.
2. En particulier, vous devez être capable de refaire l'intégralité du sujet, seul, sans aucune aide ni support.
3. Votre travail sera corrigé automatiquement par l'outil de correction automatique CAT. Cela implique que vous devez respecter scrupuleusement les consignes de chaque exercice. Faites très attention aux messages qu'il vous est demandé d'afficher. Un espace en trop, un saut de ligne en moins et l'exercice risque d'échouer.
4. L'enseignant voit votre activité sur le site, ainsi que l'historique de vos dépôts. Pensez à déposer votre travail régulièrement afin qu'il puisse vous apporter des conseils personnalisés.
5. Si l'enseignant vous demande de rendre votre travail sur papier, vous devez répondre directement sur le sujet en respectant absolument la zone prévue à cet effet. Tout ce qui se trouve en dehors de la zone sera ignoré.
6. Si le sujet contient un QCM, vous devez colorier les cases avec un stylo bleu ou noir. Les autres couleurs seront ignorées.
7. Chaque feuille est identifiée de manière unique. Vous pouvez donc rendre votre sujet avec les feuilles mélangées, mais il est préférable de les trier car cela vous permet de vérifier que vous n'en avez pas oublié une.
8. Si vous faites face à un problème, un bug, une erreur ou que vous souhaitez participer à l'amélioration de la plateforme, envoyez un mail à l'adresse suivante : support-cat@liste.parisnanterre.fr

Ne rien écrire dans cette zone

Dans ce TD, nous allons voir comment programmer certaines fonctions usuelles de manière itérative et récursives. L'objectif étant de vous apprendre à réaliser des choix de conception. A quel moment utiliser une fonction itérative ? A quel moment préférer sa version récursive ? Comment sont affectés le temps d'exécution, la quantité de mémoire utilisée et le code source ?

Fonction puissance

```
1 int puissance(int x, int n);
```

Écrire une fonction qui attend en entrée deux entiers x et n (avec $n \geq 0$) et retourne x^n. Si $n < 0$, retourner n.

Ne rien écrire dans cette zone

Fonction puissance, version récursive

```
1 int fonction_puissance_recursive(int x, int n);
```

Écrire une fonction récursive qui accepte en arguments deux entiers x et n et qui retourne x à la puissance n (soit x^n). Pour cela, on utilisera les remarques suivantes :

1. $x^0 = 1$
2. $x^n = x \cdot x^{n-1}$ lorsque $n > 0$.

Fonction puissance, version récursive rapide

```
1 int fonction_puissance_recursive_rapide(int x, int n);
```

Ecrire une fonction récursive qui accepte en arguments deux entiers x et n et qui retourne x à la puissance n (soit x^n). Pour cela, nous allons utiliser l'algorithme d'exponentiation rapide qui permet de calculer une puissance d'un nombre plus efficacement qu'en le multipliant n fois par lui même. on utilisera les remarques suivantes :

1. $x^0 = 1$
2. $x^{2 \cdot n} = (x^n)^2$
3. $x^{2 \cdot n+1} = x \cdot (x^n)^2$

Il faut donc distinguer le cas où n est pair et celui où n est impair. Vous n'avez le droit qu'à un seul appel récursif.

Ne rien écrire dans cette zone

Fonction factorielle

La factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n. Par exemple :

- $1! = 1$
- $2! = 1 \cdot 2 = 2$
- $3! = 1 \cdot 2 \cdot 3 = 6$
- $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$
- $10! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 = 3628800$

Écrivez une fonction qui retourne la factorielle d'un entier passé en argument. On suppose que l'entier passé en argument est superieur ou égal à 1.

```
1 int factorielle_iterative(int n);
```

Fonction factorielle, version récursive

```
1 int factorielle_recursive(int n);
```

La factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n. Par exemple :

- $1! = 1$
- $2! = 1 \cdot 2 = 2$
- $3! = 1 \cdot 2 \cdot 3 = 6$
- $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$
- $10! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 = 3628800$

Écrivez une fonction récursive qui retourne la factorielle d'un entier passé en argument. Pour cela, on utilisera la remarque suivante :

1. $1! = 1$
2. $n! = n \cdot (n-1)!$ pour $n > 1$

Tous les entiers de x à y, version récursive

```
1 void tous_les_entiers_de_x_a_y_recursif(int debut, int fin);
```

Écrire une fonction récursive qui attend en arguments deux entiers *debut* et fin. Cette fonction doit afficher tous les entiers de *debut* (inclus) à fin (inclus). Lors de l'affichage de chaque nombre, vous devrez sauter une ligne. Notez que l'on ne fait d'affichage que lorsque la valeur de *debut* est inférieure ou égale a la valeur de fin.

Nombre de chiffres d'un entier, version récursive

```
1 int nombre_de_chiffres_recursif(int n);
```

Écrire une fonction récursive qui accepte en argument un entier et retourne le nombre de chiffres qui le composent. Par exemple, 1018 est composé de 4 chiffres. Le signe du nombre passé en argument est ignoré.

Calcul du PGCD

```
1 int pgcd(int a, int b);
```

Écrire une fonction récursive qui calcule le PGCD de deux nombres entiers. Pour rappel, on a : $PGCD(a, 0) = a$ et $PGCD(a, b) = PGCD(b, amodb)$ si $b \neq 0$).

Suite de Fibonacci, version récursive

```
1 int fibonacci_recursif(int n);
```

La suite de Fibonacci est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précèdent. Plus précisément, on a :

- $U_0 = 0$ et $U_1 = 1$.
- $U_n = U_{n-1} + U_{n-2}$.

A titre d'exemple, ses premiers termes sont :
$U_0 = 0, U_1 = 1, U_2 = 1, U_3 = 2, U_4 = 3, U_5 = 5, U_6 = 8, U_7 = 13, U_8 = 21$. Écrire une fonction récursive qui retourne la valeur du n_{ieme} terme, n étant passé en argument. Si $n < 0$ on retourne n.

Suite récursive

```
1 int autre_suite(int n);
```

Soit la suite numérique définie comme suit :

- $U_0 = 3$ et $U_1 = 5$.
- $U_n = U_{n-1} \cdot U_{n-1} \cdot U_{n-2}$.

Écrire une fonction récursive qui retourne la valeur du n_{ieme} terme de cette suite, n étant passé en argument. Si $n < 0$ on retourne n. Vous n'avez le droit qu'à deux appels récursifs.

Ne rien écrire dans cette zone

Une autre suite récursive

```
1 int encore_suite(int n);
```

Soit la suite numérique définie comme suit :

- $U_n = 2 \cdot U_{n-2}$ si $n \geq 3$ et n impair.
- 1 sinon

Écrire une fonction récursive qui retourne la valeur du n_{ieme} terme de cette suite, n étant passé en argument. Si $n < 0$ on retourne n.

Ne rien écrire dans cette zone

Recherche par dichotomie, version récursive

```
1 int dichotomie_recursive(int * tab, int debut, int fin, int valeur);
```

Lorsqu'un tableau est trié (par ordre croissant par exemple), il est possible de retrouver efficacement une valeur recherchée en utilisant une recherche par dichotomie. Le principe est de comparer m, la valeur stockée au milieu du tableau, à *valeur*, la valeur recherchée. Si elles sont égale, c'est que la valeur est présente dans le tableau. Si $m > valeur$, il faut chercher la valeur dans la moitié inférieur du tableau, sinon dans la moitié supérieure. On recommence alors sur la portion restante du tableau et ainsi de suite jusqu'à trouver la valeur recherchée ou que l'espace de recherche restant soit vide. Écrire une fonction récursive qui implémente cette méthode. Elle devra retourner l'indice de la case dans laquelle se trouve la valeur recherchée si elle est présente dans le tableau, -1 sinon.

Recherche par dichotomie

```
1 int dichotomie(int * tab, int taille, int valeur);
```

Lorsqu'un tableau est trié (par ordre croissant par exemple), il est possible de retrouver efficacement une valeur recherchée en utilisant une recherche par dichotomie. Le principe est de comparer m, la valeur stockée au milieu du tableau, à *valeur*, la valeur recherchée. Si elles sont égales, c'est que la valeur est présente dans le tableau. Si $m > valeur$, il faut chercher la valeur dans la moitié inférieur du tableau, sinon dans la moitié supérieure. On recommence alors sur la portion restante du tableau et ainsi de suite jusqu'à trouver la valeur recherchée ou que l'espace de recherche restant soit vide. Écrire une fonction (itérative) qui implémente cette méthode. Elle devra retourner l'indice de la case dans laquelle se trouve la valeur recherchée si elle est présente dans le tableau, -1 sinon.

Ne rien écrire dans cette zone

Fonction de partition

```
1 int partition(int * tab, int d, int f);
```

Ecrire une fonction qui partitionne une sous partie des éléments d'un tableau (délimitée par les indices *debut* et fin) de telle sorte que toutes les valeurs de cette sous partie qui sont inferieures à celle du pivot se trouvent à gauche du pivot, les autres se trouvant à droite du pivot. Par défaut, la première valeur de la sous partie à partitionner servira de pivot (c'est à dire la valeur d'indice *debut*).
Cette fonction doit retourner l'indice du pivot car il risque d'être déplacé.
Exemple :

- tableau en entrée : 1|2|-1|7|4|8|3|9|1|0|2|1|2|3
- debut : 4
- fin : 10
- tableau après partitionnement : 1|2|-1|7|1|0|2|3|4|8|9|1|2|3
- retour de la fonction : 8

Ne rien écrire dans cette zone

Algorithme de tri : QuickSort

```
1 void tri_rapide(int * tab, int d, int f);
```

Implémentez l'algorithme du tri rapide. Il s'agit de décomposer un tableau en deux sous-tableaux grâce à une opération de partition :
int partition(int tab[],int d,int f)
Cette opération de partition détermine un pivot et retourne l'indice de ce pivot. Il suffit ensuite de trier récursivement sur la partie gauche (allant de l'indice debut inclus à l'indice du pivot exclus) puis sur la partie droite (allant de l'indice du pivot exclu jusqu'à l'indice de fin inclus).

Ne rien écrire dans cette zone