Récursivité

ENSIIE FISA IAP 2022-2023

Si vous avez déjà fait les exercices suivants pendant le premier TP, vous pouvez les passer. **Attention**, certains exercices (le premier par exemple) mentionnent des questions supplémentaires sur la récursivité terminale.

Exercice 1 — *Factorielle*

Ecrire un programme `fact.c` qui contient une fonction qui connaissant un entier `n` calcule sa factorielle récursivement. On rappelle que $n! = \prod_{i=1}^{n} i$. Ecrire une version récursive non terminale et une version récursive terminale.

Exercice 2 — *Dichotomie Récursive*

Ecrire un programme `dichoRec.c` qui contient une fonction qui connaissant un entier `min`, un entier `max` supérieur à `min`, un tableau `tab` contenant au moins `max` entier triés par ordre croissant et un entier `x`, vérifie si `x` est présent dans le tableau entre le `min`$^{\text{e}}$ élément et le `max - 1`$^{\text{e}}$ élément. Cette fonction doit être récursive.

Exercice 3 — *Remplacement de caractères*

Ecrire un programme `replace.c` qui contient une fonction qui connaissant une chaîne de caractères `s`, deux caractères `c` et `d` et un entier `i` remplace tous les caractères `c` situés après le `i`$^{\text{e}}$ caractère de la chaîne `s` par `d`. Cette fonction doit être récursive. (Attention à bien initialiser votre chaîne de caractères avec un tableau ou un `malloc`).

Exercice 4 — *Des maths !*

Ecrire un programme `log.c` qui contient une fonction qui connaissant un `double` $0 < \texttt{x} < 1$ calcule $-\log(1 - \texttt{x})$ comme suit :

Poser $l = 0$. Pour tout i de 1 à l'infini, si $\frac{x^i}{i} < 0.001$, renvoyer l. Sinon $l = l + \frac{x^i}{i}$.

Coder cette fonction sous forme itérative puis sous forme récursive. On pourra ajouter des paramètres à la fonction récursive si nécessaire.

Exercice 5 — *PGCD* Ecrire un programme `pgcd.c` qui calcule le PGCD de deux nombres avec la formule suivante : le PGCD de $a \in \mathbb{N}^*$ et $b \in \mathbb{N}^*$ (avec $a \leq b$) est a si $a = b$ et le PGCD de a et $b - a$ sinon.

Rajouter au début du fichier la ligne `typedef unsigned long long int ulli;`. Utiliser le type `ulli` à la place du type `int` pour a et b. Il devrait maintenant être possible appeler votre fonction avec $a = 2$ et $b = 9000000000000000000$ ($9 \cdot 10^{18}$). Que constate-t-on ? Comment améliorer l'algorithme pour qu'il termine ?

Exercice 6 — *Permutations* On souhaite, connaissant une chaîne de caractère, afficher toutes les permutations possibles de cette chaîne. Par exemple avec `abc` en entrée, la fonctoin doit afficher les chaînes suivantes `abc acb bac bca cab cba` (avec par exemple un sauf de ligne entre deux).

Créer un fichier `permut.c` qui utilise l'algorithme suivant pour effectuer cela : connaissant une chaîne `s` en entrée, on considère s_0 le premier caractère et s' la suite ; calculer récursivement toutes les permutations possibles des caractères de `s'`. Pour chaque permutation s", et pour chaque manière d'insérer s_0 dans s", on obtient une permutation de `s`.

1. Créer une fonction qui, connaissant une chaîne s, un caractère c et un entier i entre 0 et $|s|$, renvoie une nouvelle chaîne contenant s où le caractère c a été inséré en position i (l'indice de c dans la chaîne résultante est i).
2. Créer la fonction demandée par l'exercice. Attention à libérer la mémoire.
3. Transformer la fonction pour qu'elle devienne récursive terminale. Pour cela il faut modifier les paramètres d'entrées, considérez la fonction qui prend en entrée une liste l de chaînes de caractères et une chaîne de caractères s, et qui renvoie toutes les manières d'insérer les lettres de s dans toutes les chaînes de la liste l. Par exemple, avec l =[`ab`, `ba`] et s = `cd` alors le résultat est [`cdab`, `cadb`, `cabd`, `acdb`, `acbd`, `abcd`,`dcab`, `dacb`, `dabc`, `adcb`, `adbc`, `abdc`, `cdba`, `cbda`, `cbad`, `bcda`, `bcad`, `bacd`, `dcba`, `dbca`, `dbac`, `bdca`, `bdac`, `badc`]

Exercice 7 — *Statues compatibles*

On considère un musée de sculpture disposant d'un grand nombre de statues. Il souhaite mettre en avant certaines d'entre elles en les mettant à l'entrée du musée Cependant, certaines statues ne peuvent être mises ensemble à l'entrée du musée pour des raisons variées : répétition de couleur, de thématique, de période, d'auteur, ... On dit qu'elles ne sont pas compatibles. Comment mettre en avant le plus de statues compatibles possibles ?

Récupérez les fichiers `statue1.txt` `statue2.txt` et `statue3.txt`. Les fichiers sont au format suivant :

```
N
Nom_statue_1
Nom_statue_2
...
Nom_statue_N
0 1 0 0 0 1 1 0 1 1 ... 1 0
0 0 1 0 1 0 1 0 1 1 ... 1 0
...
0 0 0 0 0 1 1 1 1 1 ... 0 0
```

Le prermier nombre `N` est le nombre de statues. Les `N` lignes suivantes contiennent les nom des `N` statues, sans espace. Les `N` lignes suivantes contiennent chacune `N` entiers. Le j^{e} entier de la i^{e} ligne indique si la i^{e} statue est compatible avec la j^{e} statue : si c'est 1 alors Oui et si c'est 0 alors Non.

On suppose que le fichier est donnée en entrée standard de votre programme (`./programme < statue1.txt` par exemple).

1. Créez un programme `statue.c` contenant fonction qui lit en entrée standard la première ligne et renvoie `N`. Pour rappel, si l'entrée standard contient un entier, vous pouvez utiliser pouvez utiliser `scanf("%d", &x)` pour lire son contenu et le transférer dans un entier `x`.
2. Créez une fonction qui reçoit `N` en entrée et lit en entrée standard `N` lignes et renvoie un pointeur vers `N` chaînes de caractères contenant les `N` noms des statues. Utilisez `scanf` avec le format `"%s"`.
3. Créez une fonction qui reçoit `N` en entrée et lit en entrée standard `N` lignes et renvoie un pointeur vers `N` pointeurs sur `N` entiers contenant la matrice des 1 et des 0 du fichier. Pour pouvez utiliser `scanf` pour chaque entier.
4. Utilisez maintenant l'algorithme suivant pour trouver le plus grand ensemble de statues compatibles :
   - Il y a deux choix possibles pour la première statue s_1 : soit on la garde dans la solution, soit on ne la garde pas.
   - Si on la garde, alors lister l'ensemble S des statues compatibles avec s_1. Trouver récursivement la solution optimale si on avait en entrée que les statues de S, et ajouter s_1 à cette solution
   - Si on ne la garde pas, alors soit S l'ensemble des statues autres que s_1. Trouver récursivement la solution optimale si on avait en entrée que les statues de S.
   - Puisqu'on ne sait pas à l'avance s'il faut garder ou non s_1, il suffit de calculer et comparer ces deux solutions, et renvoyer la meilleure.

Exercice 8 — *Stockage de points* On considère une structure qui permet d'organiser rapidement la recherche de points 2D dans une liste. Creer un programme `2Dtrees.c` qui définit une structure `struct s_point2D` contenant deux champs entiers `x` et `y`, et une structure récursive `struct s_tree2D` contenant

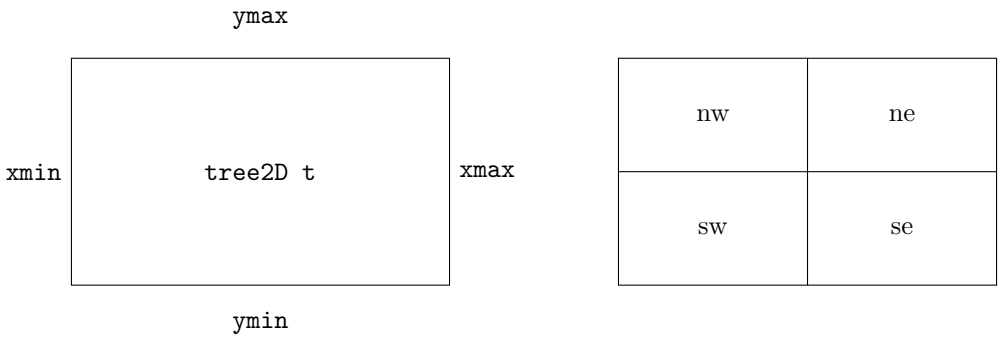
— un pointeur vers un `struct s_point2D` nommé `elem`

— 4 pointeurs vers un `struct s_tree2D` nommés `nw`, `ne`, `sw` et `se`

— 4 champs entiers `xmin`, `xmax`, `ymin` et `ymax`.

1. Utiliser un `typedef` pour renommer les pointeurs vers `struct s_point2D` et `struct s_tree2D` en `point2D` et `tree2D`.

Le `tree2D` représente un espace rectangulaire situé entre les coordonnées `xmin`, `xmax`, `ymin` et `ymax`. Un tree2D `t` contient tous les éléments de `L` qui sont situés entre ces coordonnées. Les 4 pointeurs `nw`, `ne`, `sw` et `se` représentent respectivement les quarts de rectangles supérieur gauche, supérieur droit, inférieur gauche et inférieur droit.



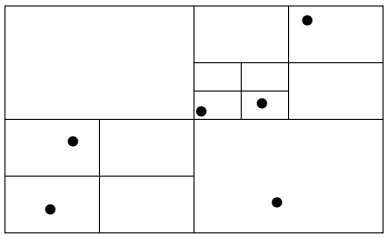
Il y a trois possibilités :

— Aucun point n'est situé dans le rectangle, alors `elem`, `nw`, `ne`, `sw` et `se` sont tous `NULL`

— il y a exactement un seul point, alors il est pointé par le champs `elem`. Et les 4 autres champs sont `NULL`.

— il y a plusieurs points dans le rectangle, alors les points situés dans le rectangle supérieur gauche sont récursivement contenu par `t->nw`, les points situés dans le rectangle supérieur droit sont récursivement contenu par `t->ne`, et ainsi de suite.

Par exemple, on peut donc arriver à la structure récursive suivante.



1. Créer une fonction qui initialise un `tree2D t`, elle reçoit en entrée les 4 coordonnées `xmin`, `xmax`, `ymin` et `ymax`.
2. Créer une fonction qui, connaissant un `tree2D t` et un `point2D p` ajoute le point `p` à `t` renvoie 1 si `p` est contenu dans `t` et 0 sinon.

3. Créer une fonction qui, connaissant un `tree2D t` et un `point2D p` ajoute le point `p` à `t` si ce point n'était pas déjà présent dans t.
4. Effectuer deux fonctions qui respectivement ajoute un `point2D p` à un tableau de `point2D` s'il n'est pas déjà dedans et recheche un point `p` dans un tel tableau.
5. Comparer le temps d'exécution de ces quatre fonctions en ajoutant `1000000` de points générés aléatoirement avec des coordonnées entre 0 et 1000, puis en recherchant `1000000` générés aléatoirement également. On peut utiliser par exemple le code suivant pour mesurer le temps.

```
#include <time.h>
...
double time_spent = 0.0;
clock_t begin = clock();
...
clock_t end = clock();
time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

```