# Colle n°3

Tous vos fichiers sont à renommer comme d'habitude (en y ajoutant votre nom en majuscule) et à **déposer sur Moodle** dans la section `Informatique - Devoirs`, tout en bas dans 1ère session. Vous devrez terminer les exercices de cette colle et me remettre l'ensemble des fichiers corrigés sur Moodle au même endroit avant dimanche soir.

Pensez à toutes les bonnes pratiques de programmation : réfléchir sur papier, tester dès que cela est possible, méthode des petits pas, programmation défensive...

Si vous avez terminé en moins d'1h, venez me demander un exercice supplémentaire.

**Exercice 1 ((PISCH, ROGER, YAO, TOURNIE)).**

**Toutes les fonctions (sauf la toute dernière) devront être récursives**

Elles seront écrites en OCaml dans un fichier `colle1.ml`

1. Écrire une fonction `puissance_naive_rec : int -> float -> float` pour calculer $x^n$ avec $x$ un réel et $n$ un entier naturel en faisant $n-1$ multiplications.
2. Écrire une fonction `puissance_rec : int -> float -> float` pour résoudre ce problème en utilisant le principe de la dichotomie
3. Combien de multiplications sont effectuées ?
4. Écrire une fonction `nb_bits : int -> int` qui calcule le nombre de bits de la représentation binaire d'un entier argument.
5. Écrire une fonction `binaire : int -> int array` qui renvoie le tableau de la représentation binaire d'un entier. Vous pourrez dans un premier temps écrire une fonction itérative, puis transformer les boucles pour les écrire de manière récursive.
6. Quel est le lien avec l'algorithme précédent ? En déduire une estimation plus fine du nombre de multiplications effectuées par l'algorithme de la question précédente en fonction du nombre de 1 et de 0 dans la représentation binaire de $n$
7. Écrire `puissance_iter : int -> int -> int` qui est l'équivalent de `puissance_rec : int -> float -> float` codée de manière itérative

**Exercice 2 (Recherche dichotomique (ROBIN, PANAZZOLO, MULLER, PUYSSEGUR)).**

Toutes les fonctions de l'exercice seront codées dans un fichier `recherche_dicho_amelioree.ml`

**Toutes les fonctions devront être récursives**

On s'intéresse à la question de la recherche d'un élément dans un tableau d'entiers trié. Notons `tab` un tableau d'entiers trié par ordre croissant et `x` un entier.

On testera sur des tableaux simples de taille raisonnable déclarés manuellement.

1. On cherche tout d'abord à déterminer un indice `i` tel que `tab[i]= x`. On renverra $-1$ s'il n'en existe pas.
   a. Écrire une fonction `recherche_dichotomique : int array -> int -> int` pour résoudre ce problème.
   b. Combien de tests d'égalités et d'inégalités sont effectués ?
2. On cherche maintenant à déterminer tous les indices `i` tels que `tab[i]= x`.
   a. Justifier que c'est un intervalle.
   b. Écrire une fonction `recherche_prefixe : int array -> int -> int -> int -> int` qui prend un tableau `tab` , un élément `x` , deux indices `debut` et `fin` , et cherche le plus **grand** $j$ dans `[debut, fin ]` tel que`tab[j] = x`. On renverra $-1$ s'il n'en existe pas.
   c. Écrire une fonction `recherche_suffixe : int array -> int -> int -> int -> int` qui prend un tableau `tab` , un élément `x` , deux indices `debut` et `fin` , et cherche le plus **petit** $j$ dans `[debut, fin ]` tel que`tab[j] = x`. On renverra $-1$ s'il n'en existe pas.
   d. Écrire une fonction `recherche_intervalle : int array -> int -> (int,int)` qui résout ce problème en renvoyant une paire $(g, d)$ représentant l'intervalle des indices `i` tels que `tab[i]= x`. La fonction renverra $(-1, -1)$ s'il n'en existe pas.
   e. Combien de tests d'égalités et d'inégalités sont effectués ?

**Exercice 3 (Listes OCaml : THIRY, MASSE, PUSNIAK ).**

Toutes les fonctions de cet exercice seront codées dans un fichier `exercices-listes2.ml`. Les fonctions

du module `List` vues en cours sont autorisées, mais on peut faire sans !
Pour les fonctions récursives, on ne se focalisera pas sur le fait de les rendre terminales. Cela pourra être fait en complément à la fin de l'exercice.

1. Écrire une fonction **récursive** `merge : int list -> int list -> int list` qui reçoit deux listes d'entiers triés et qui renvoie une liste formée des éléments des deux listes qui soit triée. Par exemple, `merge [1;5;7] [2;4;6;8;10]` renvoie `[1;2;4;5;6;7;8;10]`
2. Écrire une fonction **récursive** `max_liste_generique: ('a -> 'a -> bool) -> 'a list -> 'a` qui renvoie le plus grand élément d'une liste non vide, selon une fonction de comparaison donnée.
3. Écrire une fonction **récursive** `split : int -> int list -> int list * int list` qui reçoit un entier `n`, une liste d'entiers et qui renvoie deux listes, la première étant formée des `n` premiers éléments de la liste, la seconde étant formée des éléments restants. Si la liste contient moins de `n` éléments, le résultat sera formé de la liste initiale et d'une liste vide. L'ordre des éléments de la liste initiale doit être préservé dans les deux listes renvoyées. Par exemple `split 3 [1;2;3;4]` renvoie `([1;2;3], [4])` .
4. Écrire une fonction `dispo` qui renvoie le plus petit entier **naturel** ( i.e positif) qui n'apparaît pas dans la liste passée en argument. La liste peut contenir des entiers quelconques, y compris négatifs, et n'est généralement pas triée.

**Exercice 4 (Sommes consécutives (KLEIN)).**

Le problème de la sous-liste maximale est le problème de trouver, dans une liste d'entiers (positifs ou négatifs) la sous-liste ayant la plus grande somme.
Par sous-liste, on entend ici un liste d'éléments contigus.
Par exemple, dans la liste d'entiers `[-1;1;-3;4;-1;2;1;-5;4]` une sous-liste ayant la plus grande somme est la sous-liste `[4;-1;2;1]` avec une somme 6.

Soit $l = [l_0; l_1; \ldots; l_{n-1}]$ une liste de taille $n$. Si $i$ et $j$ sont des entiers naturels strictement inférieurs à $n$, on appelle *somme consécutive* (ou tranche) dans $l$ la quantité :

$$\sum_{k=i}^{j} l_k$$

On note $s$ la valeur maximale des sommes consécutives.

Toutes les fonctions de cet exercice seront codées dans un fichier `kadane.ml`.

**1.** Écrire une fonction `max_sum_naive` qui calcule $s$ avec une complexité *quadratique* en la taille de $l$.

**2.** Si $j \in [\![0, n-1]\!]$, on note $s_j$ la plus grande somme consécutive finissant en $j$.

$$s_j = \max_{0 \leqslant i \leqslant j} \sum_{k=i}^{j} t_k$$

**3.** Calculer à la main proprement sur papier toutes les plus grandes sommes consécutives si $l = [1, -4, 1, 5, -7, 0]$.

**4.** Écrire une fonction `max_liste_generique: ('a -> 'a -> bool) -> 'a list -> 'a` qui renvoie le plus grand élément d'une liste non vide, selon une fonction de comparaison donnée.

**5.** En déduire une nouvelle fonction `max_sum_kadane` **récursive** qui calcule $s$ avec une complexité *linéaire* en la taille de $l$. *Il peut être utile d'utiliser une seconde liste intermédiaire* $[0; m_1; m_2; \ldots; m_n]$ *pour mémoriser les plus grandes sommes intermédiaires. Il peut également être utile d'utiliser la fonction ci-dessus.*

**6.** Étendre la fonction `max_sum_kadane` en une fonction `kadane: int list -> int list` qui renvoie une sous-liste de somme maximale.

**7.** Établir la terminaison de `max_sum_kadane`.

*Jay Kadan (Carnegie Mellon University, Pittsburg, USA) a proposé cet algorithme de complexité linéaire en 1984.*