T NSI **Méthode "diviser pour régner"**

En bref : une technique souvent efficace, lorsqu'elle est possible, consiste à diviser un problème en plusieurs sous-problèmes indépendants, puis à les résoudre récursivement.

A) Activité introductive : algorithme d'exponentiation rapide.

L'algorithme d'exponentiation rapide est un algorithme utilisé pour calculer rapidement, de grandes **puissances** entières.
En effet, le calcul "naïf " de la puissance d'un nombre a, définie par :
$a^0 = 1$ et $a^n = a \times a^{n-1}$ n'est pas optimal.

Il peut être amélioré de la manière suivante :
- $a^0 = 1$
- Si n est pair, $a^n = (a \times a)^{n/2}$;
- sinon, $a^n = a \times (a \times a)^{(n-1)/2}$

1) a) Ecrire en Python une fonction récursive puiss(a , n) traduisant l'algorithme " naïf " du calcul de a^n.
b) On note T_n le nombre d'opérations nécessaires pour évaluer une fonction récursive sur un problème de taille n. Pour évaluer T_n, on cherche une relation de récurrence impliquant T_n, puis on résout la relation de récurrence.
Pour l'algorithme "naïf" précédent, donner T_0 puis exprimer T_{n+1} en fonction de T_n.
Reconnaitre la nature de la suite (T_n) et en déduire T_n en fonction de n.
En déduire que la complexité de l'algorithme " naïf "est $O(n)$.

Info : quelques complexités classiques :

Relation de récurrence	**Complexité**
$T_n = T_{n-1} + O(1)$	$O(n)$
$T_n = T_{n-1} + O(n)$	$O(n^2)$
$T_n = 2T_{n-1} + O(1)$	$O(2^n)$

2) a) Ecrire en Python une fonction récursive puiss_rap(a , n) traduisant l'algorithme d'exponentiation rapide.
b) Ajouter une variable globale k comptant le nombre d'appels récursifs, et donner la valeur de k pour plusieurs valeurs de n .
Vérifier que k est de l'ordre de $\log_2(n)$.

En effet, à chaque appel récursif, la taille du problème est divisée par deux.
On a donc une relation de récurrence de la forme $T_{n+1} \leq T_{n/2}$ et $T_1 = 1$ qui mène à une complexité en $O(log_2(n))$.

B) Principe général de la méthode "diviser pour régner"

Le paradigme de programmation "diviser pour régner" (ou, en anglais, *divide and conquer*) consiste à ramener la résolution d'un problème dépendant d'un entier n à la résolution de un ou plusieurs sous-problèmes indépendants dont la taille des entrées passe de n à $n/2$ ou une fraction de n. Une fois les sous-problèmes résolus, on les recombine afin d'obtenir la solution du problème de départ.

Le paradigme "diviser pour régner" repose donc sur 3 étapes :

- **DIVISER** : le problème d'origine est divisé en un certain nombre de sous-problèmes
- **RÉGNER** : on résout les sous-problèmes (les sous-problèmes sont plus faciles à résoudre que le problème d'origine)
- **COMBINER** : les solutions des sous-problèmes sont combinées afin d'obtenir la solution du problème d'origine.

Les algorithmes basés sur le paradigme "diviser pour régner" sont très souvent des algorithmes récursifs.
La particularité est ici que la taille des problèmes est **divisée** à chaque appel récursif plutôt que seulement réduite d'une unité.

C) Tri partition/fusion

Nous avons déjà étudié en Première des algorithmes de tri : le tri par insertion, le tri par sélection, et le tri à bulles. Ces trois algorithmes sont quadratiques, c'est-à-dire que leur complexité est $O(n^2)$. Ils sont donc relativement peu efficaces dès que le nombre n d'éléments du tableau à trier devient grand.
Nous allons maintenant étudier une nouvelle méthode de tri : le tri partition/fusion, ou tri-fusion. Le tri-fusion est basé sur le principe "diviser pour régner".
Comme pour les algorithmes déjà étudiés, cet algorithme de tri fusion prend en entrée un tableau non trié et donne en sortie, le même tableau, mais trié.

Principe du tri fusion (on parlera ici de liste, en vue d'une implémentation en Python) :

On veut trier la liste L suivant le principe "diviser pour régner" :

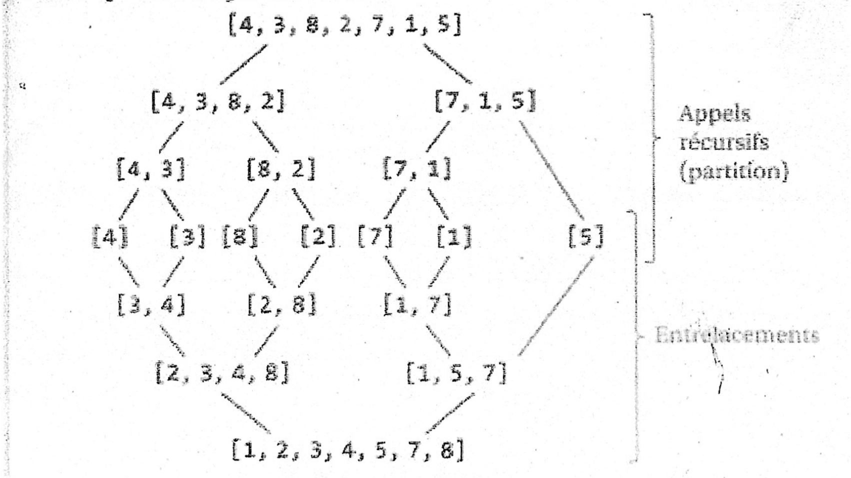
▪ Pour rassembler (= fusionner) deux listes L1 et L2 déjà triées, on peut les interclasser de la manière suivante : on compare les plus petits éléments de chacune d'elles, on place le plus petit des deux dans une nouvelle liste lst, on le "supprime" de la liste d'où il provient, et on poursuit cette opération jusqu'à épuisement d'une des deux listes. On complète alors lst en ajoutant à la fin de celle-ci les éléments de la liste non vide.
En pratique, on ne supprime pas vraiment le plus petit élément d'une des listes L1 ou L2 : on utilise deux indices i1 et i2 qui désigneront le début de chaque liste, comme des curseurs que l'on va faire évoluer jusqu'à ce que l'un deux atteigne la fin de la liste.

Exemple :

L1 = [1, 5, 7] ; L2 = [3 , 8 , 12, 20] ; lst = [ ]
i1 = 0 et i2 = 0
On compare L1[i1] et L2[i2] c'est-à-dire L1[0] et L2[0].
On a L1[0] < L2[0] car 1 < 3 donc on ajoute L1[0] c'est-à-dire 1 à la liste lst.
On aura donc lst = [1].
On "supprime" L1[0] en incrémentant l'indice i1 : i1 = 1.
A présent, le "premier" élément de L1 est L1[i1] c'est-à-dire L1[1] = 5.
etc.

▪ Le tri fusion est alors défini de la manière suivante :
► Si la liste L a au plus un élément, elle est déjà triée.
► Si la liste L a deux éléments ou plus, on partage L en deux sous-listes L1 et L2 de même taille à un élément près, puis on appelle récursivement la fonction sur chacune des sous-listes et on interclasse (=fusionne) les sous-listes triées.

■ Exemple illustré par un schéma :



Exercice 1 : Illustrer par un schéma du même type le tri fusion de la liste suivante :
[10, 2 , 7 , 3 , 4 , 1 , 8 , 6 , 2]

Exercice 2 : Implémentation en Python du tri fusion.

1) Créer une fonction `interclassement ( L1 , L2 )` qui prend en paramètres deux listes triées L1 et L2 et retourne une liste lst triée contenant tous les éléments de L1 et L2.

2) Créer une fonction récursive `tri_fusion( L )` qui retourne la liste triées des éléments de L. Cette fonction utilisera la fonction `interclassement` définie précédemment.

Rappel : en Python, L[n :] désigne la liste constituée des éléments de L depuis l'indice n jusqu'à la fin. L n'est pas modifiée.
L[: m] désigne la liste constituée des éléments de L depuis l'indice 0 jusqu'à l'indice m – 1. L n'est pas modifiée.
Exemple : L = [1, 2 , 3 , 4]
L [2 :] renvoie la liste [3 , 4]
L [: 2] renvoie la liste [1 , 2]

3) Evaluer la complexité de la fonction tri_fusion.