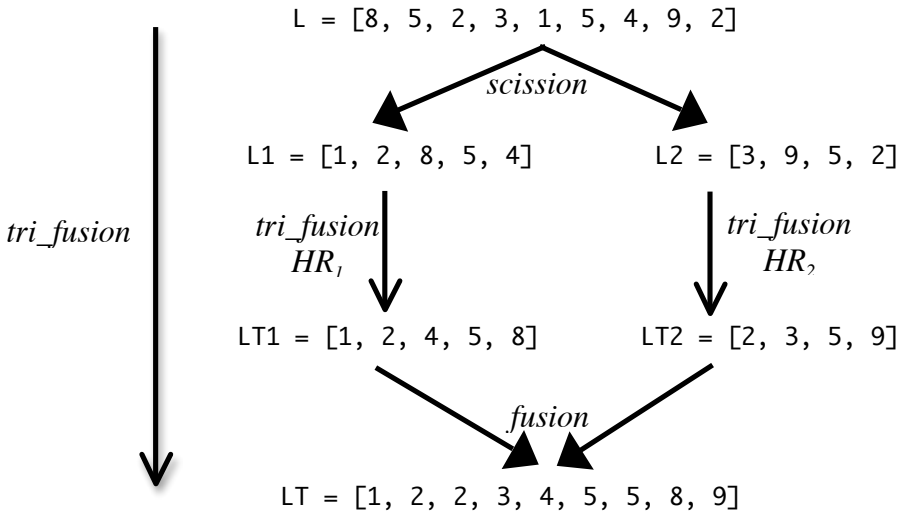
Algo & Prog, avec Python
(L1-Sciences)
TP n° 6, Automne 2016

Exercice 6.1 Le **tri par fusion** (*mergesort*).

a) Programmez la fonction `fusion(LT1,LT2)` prenant deux listes croissantes d'entiers `LT1` et `LT2`, et retournant la liste croissante mélange des deux. Exemple :

```
>>> fusion([1,3,4,7,10],[2,3,5,7,8])
[1,2,3,3,4,7,7,8,10]
```

b) Le principe du *tri par fusion* suit le diagramme dichotomique suivant, dans lequel la *scission* coupe en deux sous-listes de même longueur ou presque [à 1 élément près] :



Utilisez la fonction `fusion` pour programmer par récurrence une fonction `tri_fusion(L)` prenant une liste de nombres réels `L` et retournant une copie croissante de cette liste.

Exercice 6.2 **Le crible d'Eratosthène**. Il s'agit d'un algorithme célèbre permettant de construire la liste des entiers premiers jusqu'à n :

1. On commence par construire la liste L des entiers de 2 jusqu'à n inclus. Nous allons maintenant rayer les nombres non premiers de cette liste !
2. Le premier nombre non rayé est premier (pourquoi ?). Je raye tous ses multiples. Et je répète ce point 2 jusqu'à avoir examiné tous les nombres de la liste. Exemple, n = 26 :

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
2	3	•	5	•	7	•	9	•	11	•	13	•	15	•	17	•	19	•	21	•	23	•	25	•
2	3	•	5	•	7	•	•	•	11	•	13	•	•	•	17	•	19	•	•	•	23	•	25	•

etc jusqu'à obtenir :

```
[2, 3, 5, 7, 11, 13, 17, 19, 23]
```

Programmez la fonction `crible(n)` retournant la liste des nombres premiers de `[2,n]`.

Compléments à vérifier sur machine

Exercice 6.3 Comparez sur machine les temps de calcul pour construire la liste des nombres premiers jusqu'à 50000 :

a) En utilisant le *crible d'Eratosthènes* vu ci-dessus.

b) En utilisant la fonction `estPremier` de l'exercice 2.2b du TP2.

Exercice 6.4 a) Programmez - en une ligne ? - une fonction `copie_liste(L)` retournant une copie de la liste `L`.

b) Exécutez la session ci-dessous au toplevel Python :

```
> L = [1, 2, [3, 4, 5], 6]
> LC = copie_liste(L)
> LC
[1, 2, [3, 4, 5], 6]
> L[1] = 0
> L
[1, 0, [3, 4, 5], 6]
```

```
> LC
[1, 2, [3, 4, 5], 6]
> L[2][0] = -1
> L
[1, 0, [-1, 4, 5], 6]
> LC     # devinez SANS MACHINE !
???????
```

N.B. Le clonage de la liste ne clone en réalité que les éléments du premier niveau. Si certains éléments de la liste sont eux-mêmes de gros objets comme des listes, ils ne sont pas clonés ! Le problème du clonage est important dans les langages de programmation[1].

[1] Pour copier en profondeur, renseignez-vous sur la fonction *deepcopy* de Python.

Algo & Prog, avec Python
(L1-Sciences)
TD n° 6, Automne 2016

Exercice 6.1 a) Programmez la fonction `makelist(n,x)` retournant une liste de longueur `n` dont tous les éléments sont égaux à `x`. Exemple :

```
>>> makelist(5,8)
[8, 8, 8, 8, 8]
```

b) Complétez les portions manquantes dans la session au toplevel Python ci-dessous :

```
>>> L = makelist(3,[10,20,30])
>>> L
????????
>>> L[0][0] = 1000          # je modifie le premier élément de la première liste
>>> L
????????
```

Exercice 6.2 A la page 9 du cours 6, se trouvent les **équations de complexité du tri par pivot** `quicksort` (on mesure en moyenne, sans grande rigueur, le nombre de fois qu'on ajoute un élément à une liste) :

$$c_0 = c_1 = 0$$
$$c_n = 2c_{n/2} + 3n$$

Nous souhaitons avoir une idée de l'ordre de grandeur du terme général (c_n) de cette suite, lorsque n est grand. Supposons – sans perte de généralité – que n est de la forme 2^k et posons $d_k = c_n$.

a) Que devient la seconde équation $c_n = 2c_{n/2} + 3n$? Elle s'écrit $d_k = \cdots$

b) Calculez la valeur exacte du terme général d_k en fonction de k.

c) En déduire la valeur de c_n en fonction de n. Concluez que $c_n = O(?)$.

Exercice 6.3 Définissez la fonction *f(x) = sin(x)/x*. Utilisez ensuite une **exception** pour calculer, dans une boucle évoluant de -3 à 3 inclus, la liste des couples *(x, f(x))*. *On veut gérer la singularité en 0 par une exception, pas par un* `if`*...*

Exercice 6.4 *IMPORTANT*. Reprenons la **recherche par dichotomie** du cours 6 pages 12-14. Le programme donné en cours utilise beaucoup de constructions de *tranches* comme `L[:m]` ou `L[m+1:]` qui sont des recopies coûteuses de portions de listes. Pour les éviter et accélérer la recherche, vous allez travailler sur place [sans utiliser de listes supplémentaires] en gérant deux indices `a` et `b`. Au départ `a=0` et `b=len(L)-1`. Vous allez faire se rapprocher les curseurs `a` et `b`, en abandonnant la moitié de l'intervalle `[a,b]` chaque fois. A vous !