Comparaison des tris

[1]:
```
def tri_selection(L):
    """
    L'idée est d'aller chercher le plus petit élément de L
    puis de le placer en première position.
    On recommencer à partir de la position 2.
    """
    n = len(L)
    for i in range(0, n-1):
        # on cherche le minimum de L[i:]
        m = i # indice courant du minimum
        for j in range(i, n):
            if L[j] < L[m]:
                m = j
        # le minimum de L[i:] est à la position m,
        # plaçons le au début, c'est à dire en i
        L[i], L[m] = L[m], L[i]
    return L
```

[2]:
```
def tri_insertion(L):
    for i in range(1, len(L)):
        # on va placer l'élément d'indice i au bon endroit dans la liste
        j = i - 1 # indice juste avant
        x = L[i] # on sauvegarde l'élément à placer
        while j >= 0 and L[j] > x:
            L[j + 1] = L[j]
            # on fait reculer l'élément en position j de 1 place.
            j = j - 1
        L[j + 1] = x
    return L
```

[3]:
```
def fusion(L1, L2):
    """
    L1 et L2 sont deux listes triées.
    Retourne la liste triée contenant les éléments de L1 et L2.
    Si des éléments sont égaux dans les listes L1 et L2,
    on place d'abord les éléments de L1
    """
    L = []
```

```
    i1 = 0 # indice courant dans L1
    i2 = 0 # indice courant dans L2
    n1 = len(L1)
    n2 = len(L2)
    while i1 < n1 and i2 < n2:
        if L1[i1] <= L2[i2]:
            L.append(L1[i1])
            i1 += 1
        else:
            L.append(L2[i2])
            i2 += 1
    # on a épuisé une des deux listes, il suffit de mettre tous les éléments
    # restant dans l'autre à la fin de L
    L.extend(L1[i1:])
    L.extend(L2[i2:])
    return L

def tri_fusion(L):
    n = len(L)
    if n <= 1: # cas d'arrêt de la récursion
        return L
    moitie = n // 2
    L1 = tri_fusion(L[:moitie])
    L2 = tri_fusion(L[moitie:])
    return fusion(L1, L2)
```

[4]:
```
def partitionne(L, pivot):
    """
    Retourne les deux sous-listes précédemment mentionnées
    plus une liste des éléments égaux au pivot
    """
    L1 = []
    Lp = []
    L2 = []
    for e in L:
        if e < pivot:
            L1.append(e)
        elif e == pivot:
            Lp.append(e)
        else:
            L2.append(e)
    return L1, Lp, L2

def rassemble(L1, Lp, L2):
    """
    Reconstruit la liste L
    """
```

```
    return L1 + Lp + L2

def choisit_pivot(L):
    """
    Retourne l'indice du pivot choisi.
    """
    return 0

def quicksort(L):
    if len(L) < 2:
        return L
    pivot = L[choisit_pivot(L)]
    L1, Lp, L2 = partitionne(L, pivot)
    L1 = quicksort(L1)
    L2 = quicksort(L2)
    return rassemble(L1, Lp, L2)
```

[5]:
```
import random
```

[7]:
```
%timeit tri_selection([random.random() for _ in range(5000)])
%timeit tri_insertion([random.random() for _ in range(5000)])
%timeit tri_fusion([random.random() for _ in range(5000)])
%timeit quicksort([random.random() for _ in range(5000)])
%timeit [random.random() for _ in range(5000)].sort()
```

```
458 ms ± 3.54 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
467 ms ± 8.04 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
8.69 ms ± 48.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
6.16 ms ± 23.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
837 µs ± 2.44 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```