Chapitre 3 : Tris.

MP Lycée Camille Jullian

mars 2022

Dans ce dernier chapitre (déjà !), nous allons étudier en détail les divers algorithmes de tris de listes, en insistant en particulier sur la complexité de ces différents algorithmes. La problématique du tri de listes est centrale en informatique, cela fait donc un certain temps que des algorithmes optimaux d'un point de vue complexité ont été conçus. Mais ces derniers ne sont pas forcément les plus intuitifs, c'est pourquoi nous commencerons notre étude par celle des algorithmes de tri dits « naïfs », avant de nous intéresser à ceux qui permettent vraiment de gagner en efficacité du point de vue complexité. De plus, on s'intéressera également un peu à la complexité spatiale de nos algorithmes, qui sera mesurée ici via un critère très simple : un algorithme de tri est dit **en place** s'il effectue le tri à l'intérieur de la liste donnée en paramètre, autrement dit sans nécessiter de créer des variables auxiliaires de type liste (on pourra par contre utiliser des variables de type flottant si nécessaire). À une exception notable près (le tri fusion), tous les tris que nous allons voir dans ce chapitre peuvent être programmés en place (quitte à avoir un algorithme moins intuitif et donc moins lisible pour certains, mais sans perte en termes de complexité), nous donnerons donc les deux versions à chaque fois.

1 Algorithmes naïfs de tri.

Avant de nous lancer dans une liste des différentes possibilités, précisons bien quels types d'algorithmes nous allons comparer dans ce chapitre : un algorithme de tri **par comparaison** est un algorithme dans lequel le tri sera effectué en procédant à des comparaisons entre éléments de la liste (et la complexéité obtenue en comptant le nombre de ces comparaisons, les autres opérations étant habituellement considérées comme négligeables en termes de temps d'exécution). Il existe en fait d'autres algorithmes de tri, dont certains peuvent d'ailleurs être plus efficaces que ceux que nous allons étudier, mais ceux-ci ne fonctionnent en général que sur des cas particuliers (listes contenant certains types de données). Notons d'ailleurs que, pour pouvoir trier une liste en appliquant un tri par comparaison, il faut bien entendu que les éléments de la liste soient comparables, ce qui est de toute façon un préalable logique pour espérer trier la liste (en pratique, on pourra donc trier des listes dont tous les éléments sont numériques, ou dont tous les éléments sont des chaînes de caractères, la comparaison s'effectuant alors suivant l'ordre alphabétique). Nous allons commencer par évoquer un tri de ce genre, avant de revenir à l'objet principal de notre étude.

1.1 Tri par comptage.

Comme on va le voir plus bas, les algorithmes les plus efficaces pour efectuer le tri complet d'une liste de longueur n ont une complexité sous-linéaire (en $O(n\ln(n))$). Il existe pourtant des possibilités de faire un tri en temps linéaire dans certains cas très particuliers. Prenons l'exemple caricatural suivant : on dispose d'une liste de n nombres entiers dont on sait qu'ils sont tous compris entre 0 et 9 (donc avec probablement beaucoup de répétitions si n n'est pas lui-même très petit). La meilleure façon de trier la liste est tout bêtement de compter le nombre de fois où apparait chaque entier dans la liste, ce qui ne nécessite qu'un seul parcours de la liste et s'effectue donc en temps linéaire. On parle alors de tri par comptage (on se contente de compter, sans faire la moindre comparaison). Ce principe peut être exploité pour améliorer significativement les algorithmes de tris

de listes de chaînes de caractères par exemple : on commence par faire un tri par comptage sur la première lettre des chaînes de caractères (on crée donc 26 sous-listes contenant respectivement les éléments commençant par un 'a', un 'b' etc), puis on trie chacune des sous-listes. On peut même imaginer, pour des listes très grandes, de faire un premier tri par comptage sur les deux premières lettres, mais c'est plus gourmand en termes d'occupation de la mémoire (on a besoin de 26^2 « casiers » pour ranger nos éléments pour l'étape initiale). On ne donnera pas d'exemple de programmation de ce genre de chose en Python puisque ce n'est pas l'objet principal de notre étude.

1.2 Tri par sélection.

Il s'agit probablement du tri le plus intuitif à mettre en place : on recherche d'abord le plus petit (ou le plus grand) élément de la liste, puis le deuxième plus petit etc. Pour chacune de ces recherches on effectuera une recherche de minimum classique sur une liste de plus en plus petite (puisqu'on élimine bien sûr à chaque étape l'élément précédemment sélectionné. Une version de cette procédure :

```
def triselection(L) :
    L2=[]
    for i in range(len(L)) :
        m,j=L[0],0
        for k in range(1,len(L)) :
            if L[k]<m :
                m,j=L[k],k
        L2.append(m)
        del(L[j])
    return L2
```

On peut facilement créer une version récursive de ce programme, on calcule simplement le minimum puis on applique la fonction à ce qui reste dans la liste. Mais ce qui serait encore mieux, ce serait de créer une version en place de cet algorithme. C'est en fait très simple, il suffit d'effectuer un échange au sein de la liste après avoir trouvé le minimum :

```
def triselectionenplace(L) :
    n=len(L)
    for i in range(n-1) :
        m=i
        for j in range(i+1,n) :
            if L[j]<L[m]
                m=j
        L[i],L[m]=L[m],L[i]
    return
```

La complexité de cet algorithme est facile à calculer : on effectue $n-1$ comparaisons lors de la recherche du premier minimum, $n-2$ pour le second etc, soit au total $\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$ comparaison. Autrement dit, la complexité est toujours en $O(n^2)$ pour cet algorithme.

1.3 Tri par insertion.

Encore un algorithme extrêmement intuitif : on tri les éléments « dans l'ordre où on les croise » : on commence par comparer le premier élément et le deuxième, on les échange si besoin. Ensuite, on prend le troisième élément et on essaie de le placer au bon endroit par rapport aux deux premiers, ce qui nécessaite une ou deux comparaisons (s'il est plus petit que le premier, pas besoin de le comparer au deuxième), on fait pareil avec le quatrième etc. Un exemple d'implémentation en Python :

```
def triinsertion(L) :
    n,L2=len(L),[]
    for i in range(n) :
        j=0
        while j<len(L2) and L[i]>L2[j] :
            j+=1
        L2.insert(j,L[i])
    return L2
```

Est-il possible d'effectuer ce tri en place ? Oui, ce n'est même pas très compliqué : après l'étape numéro i, par hypothèse, la sous-liste constituée des i premiers éléments de L est triée par ordre croissant. On part alors, pour l'étape suivante, de l'élément $L[i+1]$, puis on l'échange avec les éléments qui le précèdent tant qu'ils sont plus grands que lui. La seule légère complication est l'obligation de faire une boucle while à indices décroissants :

```
def triinsertionenplace(L) :
    n=len(L)
    for i in range(1,n) :
        j,x=i,L[i]
        while j>0 and L[i-1]>x :
            L[j],L[j-1]=L[j-1],L[j]
            j-=1
    return
```

Quelle est la complexité de cet algorithme ? Eh bien ça dépend ! Pour la première fois, nous croisons un algorithme qui n'aura pas la même complexité dans le cas moyen que dans le meilleur cas. En effet, si on a beaucoup de chance, on a une liste qui est déjà triée. Dans ce cas, on n'aura besoin que d'une seule comparaison à chaque étape pour se rendre compte que l'élément i est déjà à sa place, ce qui donne une complexité linéaire (remarquons que, dans notre première version non en place de l'algorithme, c'est au contraire quand la liste est triée par ordre décroissant qu'on sera dans le meilleur cas, le cas de la liste croissante étant au contraire le pire cas possible). Dans le pire cas (liste en ordre décroissant), on fera le même nombre de comparaisons, donc $\frac{n(n-1)}{2}$, que pour un tri par sélection (une comparaison pour le deuxième élément, deux pour le troisième, jusqu'à $n-1$ pour le dernier à placer). Mais en moyenne, on n'aura bien sûr pas besoin de remonter jusqu'au début de la liste pour remettre l'élément à sa place. Il est assez simple de se convaincre qu'on aura en moyenne environ deux fois moins de comparaison que dans le pire cas (puisqu'on devrait remonter jusqu'au milieu de la liste), soit un nombre de comparaisons plutôt équivalent à $\frac{n^2}{4}$. On conserve donc un algorithme de complexité quadratique, mais avec une efficacité probablement supérieure à celle du tri par sélection en pratique.

1.4 Tri à bulles.

Le tri à bulles, outre le fait qu'il a un nom complètement ridicule, représente l'un des tris les moins efficaces qui soient régulièrement utilisés en pratique. Son principe est en tout cas très simple : on parcourt toute la liste et on compare chaque élément (sauf le dernier) avec l'élément qui le suit. S'ils ne sont pas « dans le bon ordre », on les échange. On recommence ces parcours jusqu'à avoir une liste entièrement triée, ce qui nécessaite au maximum $n-1$ parcours de la liste. En pratique, après le premier parcours, le plus grand élément de la liste sera situé en fin de liste (puisqu'on va l'échanger avec tous ceux qui lui sont inférieurs) puis le deuxième plus grand sera positionné correctement après le deuxième passage, et ainsi de suite. On peut donc arrêter les comparaisons de plus en plus tôt pour améliorer légèrement la complexité. Une version Python de cet algorithme qui pour le coup se fait très naturellement en place :

```
def tribulles(L) :
    n=len(L)
    fini=False
    while not fini :
        fini=True
        for i in range(n-1) :
            if L[i]<L[i+1] :
                L[i],L[i+1]=L[i+1],L[i]
                fini=False
        n-=1
    return
```

Le nombre de comparaisons dans le pire cas (la liste est triée en ordre décroissant) sera une fois de plus de $\frac{n(n-1)}{2}$ ($n-1$ au premier passage, $n-2$ au suivant, et on a besoin de $n-1$ passages), dans le meilleur cas (liste déjà triée) on n'aura besoin que de $n-1$ comparaisons (un passage pour vérifier que la liste est triée, et on sort de la boucle while). En moyenne, on sera en fait très proche du pire cas, puisqu'on ne gagnera la plupart du temps que les quelques derniers passages qui sont ceux qui effectuent le moins de comparaisons. Surtout, le tri à bulles effectue énormément d'échanges qui ralentissent en pratique l'exécution, d'où son statut de « mauvais » algorithme de tri (cf les comparaisons effectuées plus bas).

1.5 Complexité optimale d'un tri par comparaison.

Jusqu'ici, toutes nos tentatives se sont soldées par la création de tris en complexité moyenne quadratique. Peut-on espérer mieux, et si oui, quelle est la meilleure complexité possible pour un algorithme de tri par comparaison ? Le résultat théorique suivant nous fournit la réponse :

Théorème 1. La complexité moyenne d'un algorithme de tri par comparaison est au mieux en $O(n\ln(n))$.

Démonstration. C'est plus une idée de preuve qu'une preuve vraiment rigoureuse qu'on va donner. Le principe d'un tri par comparaison est le suivant : à chaque fois qu'un effectue une comparaison, on élimine parmi tous les ordres possibles sur les n éléments de la liste un certain nombre de possibilités, jusqu'à n'en avoir plus qu'une. Par exemple, si $n=3$, on a initialement six ordes possibles sur les éléments de la liste (qu'on notera a, b et c) : abc, acb, bac, bca, cab et cba. Si on commence par exemple par comparer a et b et qu'on trouve que $b<a$, il ne reste plus que les trois possibilités bac, bca et cba. Si ensuite on compare a et c et que $a<c$, on n'a plus qu'une possibilité (bac), on peut arrêter l'algorithme. De façon plus générale, on aura $n!$ ordres possibles, et chaque comparaison pourra au mieux diviser par 2 le nombre de cas encore possibles (dans le cas moyen du moins), ce qui nécessitera donc un nombre de comparaisons égal à $\log_2(n!)$. Un petit coup de formule de Stirling montre alors que ce dernier terme est équivalent à $n\log_2(n)$, qui est donc une borne inférieure pour la complexité de nos algorithmes. □

Comme on va désormais le voir, cette borne n'est pas seulement théorique puisqu'il existe plusieurs algorithmes de tris de complexité sous-linéaire.

2 Algorithmes optimisés de tri par comparaison.

Les deux algorithmes de tri que nous allons étudier dans cettte dernière partie ont une complexité sous-linéaire en $O(n\ln(n))$ dans le cas moyen, ce qui correspond à ce qu'on peut faire de mieux. On verra pas contre qu'ils ne se comportent pas de la même façon dans le pire cas. Ils sont tous les

deux basés sur le principe « diviser pour régner » très fréquemment exploité pour faire apparaître des complexités logarithmiques : il est plus rapide de trier deux listes de longueur égales à la moitié de notre liste initiale que de trier directement la liste complète.

2.1 Tri fusion.

Le principe de cet algorithme est très simple : on découpe en deux la liste initiale, on trie les deux moitiés obtenues (en appliquant la procédure de façon récursive pour optimiser la complexité) puis on « fusionne » les deux listes obtenues pour retrouver la liste complère triée dans l'ordre croissant. On a donc besoin d'un programme auxiliaire effctuant cette opération de fusion, qui va prendre comme paramètres deux listes supposées triées par ordre croissant, et doit retourner une liste contenant tous les éléments de ces deux listes classés par ordre croissant. Cette fusion est assez simple à réaliser d'un point de vue intuitif : on compare le premier élément de chaque liste, on place le plus petit des deux en tête de la liste finale. Tout en faisant cela, on déplace le curseur au niveau du deuxième élément dans la liste dont on a extrait l'élément le plus petit, puis on compare ce deuxième élément avec le premier de l'autre liste, on place le plus petit des deux dans la liste finale tout en déplaçant le curseur dans la liste où se trouvait cet élément, etc. On peut arrêter les comparaisons une fois que le curseur a atteint la fin d'une des deux listes et finir en complétant notre liste fusionnée avec la fin de la liste qu'on n'a pas épuisée. Par exemple, si on veut fusionner les deux listes $l1 = [3, 5, 7, 8]$ et $l2 = [1, 2, 4, 6]$, en notant i et j les curseurs correspondant respectivement à $l1$ et $l2$, et L la liste finale, on effectuera les étapes suivantes :

- $i = 0$, $j = 0$, $l1[0] > l2[0]$, donc on insère la valeur 1 dans L et on effectue la modification $j = 1$ (pour l'instant, $L = [1]$).
- $i = 0$, $j = 1$, $l1[0] > l2[1]$, donc on insère la valeur 2 dans L et on effectue la modification $j = 2$ (pour l'instant, $L = [1, 2]$).
- $i = 0$, $j = 2$, $l1[0] < l2[2]$, donc on insère la valeur 3 dans L et on effectue la modification $i = 1$ (pour l'instant, $L = [1, 2, 3]$).
- $i = 1$, $j = 2$, $l1[1] > l2[2]$, donc on insère la valeur 4 dans L et on effectue la modification $j = 3$ (pour l'instant, $L = [1, 2, 3, 4]$).
- $i = 1$, $j = 3$, $l1[1] < l2[3]$, donc on insère la valeur 5 dans L et on effectue la modification $i = 2$ (pour l'instant, $L = [1, 2, 3, 4, 5]$).
- $i = 2$, $j = 3$, $l1[2] > l2[3]$, donc on insère la valeur 6 dans L et on effectue la modification $j = 4$ (pour l'instant, $L = [1, 2, 3, 4, 5, 6]$).
- j est égal à la longueur de $l2$, on arrête les comparaisons et on complète L avec les termes de $l1$ d'indice supérieur ou égal à i, ce qui donne finalement $L = [1, 2, 3, 4, 5, 6, 7, 8]$ comme souhaité.

Et un joli programme Python effectuant ce travail (on peut l'écrire de façon récursive si on le souhaite) :

```
def fusion(l1,l2) :
    n,p,i,j,L=len(l1),len(l2),0,0,[]
    while i<n and j<p :
        if l1[i]<l2[j] :
            L.append(l1[i])
            i+=1
        else :
            L.append(l2[j])
            j+=1
    if i==n :
        return L+l2[j :]
    return L+L1[i :]
```

Le nombre de comparaisons effectué par ce programme est au maximum égal à $n+p-1$, et dans le meilleur cas est égal à $\min(n,p)$ (atteint si tous les éléments de $l1$ sont plus petits que tout ceux de $l2$, ou le contraire, auquel cas L sera une simple concaténation des deux listes), ce qui donne une complexité dans le cas moyen égale à du $O(n+p)$. En particulier, si les deux listes ont une même taille égale à n, on aura un algorithme linéaire en $O(n)$. Une fois la procédure de fusion programmée, le tri fusion lui-même est extrêmement simple à programmer récursivement :

```
def trifusion(L) :
    if len(L)<2 :
        return L
    m=len(L)//2
    return fusion(trifusion(L[ :m]),trifusion(L[m :]))
```

En notant $C(n)$ la complexité du tri fusion pour une liste à n éléments, on aura par construction, dans le cas où n est pair, $C(n) = 2C\left(\frac{n}{2}\right) + O(n)$ (le dernier terme correspondant à la complexité de la fusion des deux sous-listes triées). Via les astuces de calcul classique, on peut alors écrire $C(2^k) = 2C(2^{k-1}) + O(2^k)$, donc $C(2^k) = 2C(2^{k-1}) + O(2^k)$, ou encore $\frac{C(2^k)}{2^k} = \frac{C(2^{k-1})}{2^{k-1}} + O(1)$. On en déduit facilement $\frac{C(2^k)}{2^k} = O(k)$, soit $C(2^k) = O(k2^k)$, ce qui correspond bien à $C(n) = O(n\log_2(n))$.

2.2 Tri rapide (Quick Sort).

Comme son nom l'indique, le tri rapide a été inventé (au début des années 1960) pour créer un tri extrêmement efficace. Il repose sur le même principe de dichotomie que le tri fusion, mais peut être programmé en place et sera légèrement plus rapide dans le cas moyen. Il aura par contre une complexité quadratique dans le pire cas. Le principe en est le suivant : on prend un élément aléatoire de la liste L à trier, qu'on appellera pivot, puis on sépare les éléments restants en deux catégories : ceux qui sont plus petits que le pivot, et ceux qui sont plus grands. Il ne reste plus alors qu'à appliquer récursivement l'algorithme aux deux listes obtenues. Si on ne se force pas à programmer le tri en place, c'est assez facile :

```
def decoupe(L,x) :
    L1,L2=[],[]
    for i in L :
        if i<x :
            L1.append(i)
        else :
            L2.append(i)
    return L1,L2
```

Ce premier programme effectue la découpe d'une liste en deux morceaux, sans inclure le pivot x dans l'une des deux listes. Pour effectuer le tri rapide, on va donc prendre comme pivot le premier élément de la liste (c'est un choix aussi bon que n'importe quel autre) et appliquer la fonction decoupe au reste de la liste, puis faire un appel récursif :

```
def trirapide(L) :
    if len(L)<2 :
        return L
    x=L[0]
    L1,L2=decoupe(L[1 :],x)
    return trirapide(L1)+[x]+trirapide(L2)
```

Si on souhaite effectuer un tri en place, il faut réussir à découper notre liste sans utiliser de liste auxiliaire. Pour cela, on va créer un programme auxiliaire **partition** qui prend comme arguments

une liste l, une borne gauche g (correspondant à un indice de la liste) et une borne droite d et qui va séparer les éléments de $l[g:d]$ en fonction de leur position par rapport au pivot $l[g]$ (comme on va appliquer la procédure récursivement, il faut qu'elle puisse fonctionner sur des bouts de liste et pas seulement sur la liste entière). Pour cela, on procède ainsi : on prend un par un tous les éléments de la liste du numéro $g+1$ jusqu'au $d-1$, et pour chacun d'entre eux, s'il est supérieur au pivot, on le laisse tranquille, s'il est inférieur, on l'échange avec l'élément supérieur situé le plus à gauche de la tranche étudiée, après avoir incrémenté un compteur (pour savoir justement où se trouve l'élément le plus à gauche supérieur à $l[g]$). À la fin, il ne faut pas oublier de déplacer le pivot lui-même. Illustrons tout ça sur un exemple : imaginons que la tranche à partitionner soit $[5,4,7,2,9,8,1]$.

- on initialise un compteur à $n=g$ (ici on peut considérer que $n=0$).
- l'élément $l[1]$ est égal à 4 donc plus petit que le pivot, mais comme $n=0$, on le laisse à sa place (en pratique, dans ma version de l'algorithme, on l'échangera avec lui-même), et on incrémente : $n=1$.
- l'élément $l[2]$ est égal à 7, il est plus grand que le pivot, on le laisse à sa place, on conserve $n=1$.
- l'élément $l[3]$ est égal à 2, il est petit, on incrémente ($n=2$) et on échange $l[2]$ et $l[3]$ pour obtenir $l=[5,4,2,7,9,8,1]$.
- l'élément $l[4]$ est égal à 9, il est plus grand que le pivot, on le laisse à sa place, on conserve $n=2$.
- l'élément $l[5]$ est égal à 8, il est plus grand que le pivot, on le laisse à sa place, on conserve $n=2$.
- l'élément $l[6]$ est égal à 1, il est petit, on incrémente ($n=3$) et on échange $l[3]$ et $l[6]$ pour obtenir $l=[5,4,2,1,9,8,7]$.
- enfin, on échange le pivot avec $l[3]$, et on ressort donc la liste $[1,4,2,5,9,8,7]$.

Il ne reste plus qu'à écrire la version Python :

```
def partition(L,g,d) :
    x=L[g]
    n=g
    for i in range(g+1,d) :
        if L[i]<x :
            n+=1
            L[i],L[n]=L[n],L[i]
    L[n],L[g]=L[g],L[n]
    return n
```

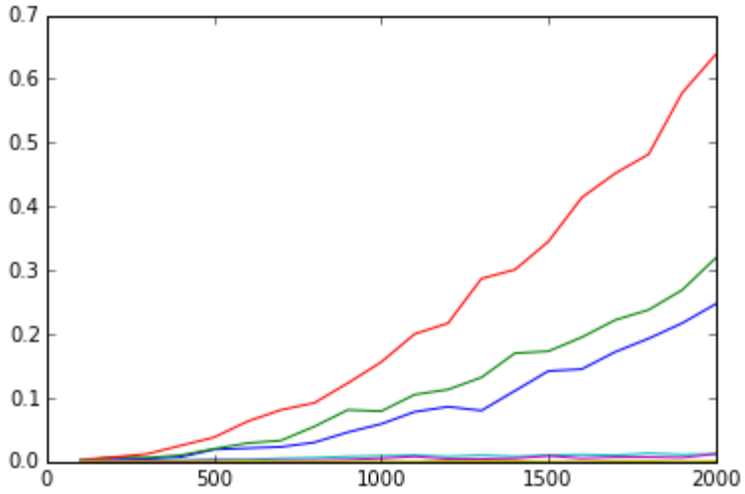
On retourne la valeur de n où se trouve le pivot en fin de partition pour savoir où couper la liste pour les appels récursifs. Il ne reste plus qu'à écrire l'algorithme de tri qui va toutefois nécessiter un programme auxiliaire :

```
def trirapide(L) :
    def aux(g,d) :
        if g<d-1 :
            n=partition(L,g,d)
        aux(g,m)
        aux(m+1,d)
    aux(0,len(L))
```

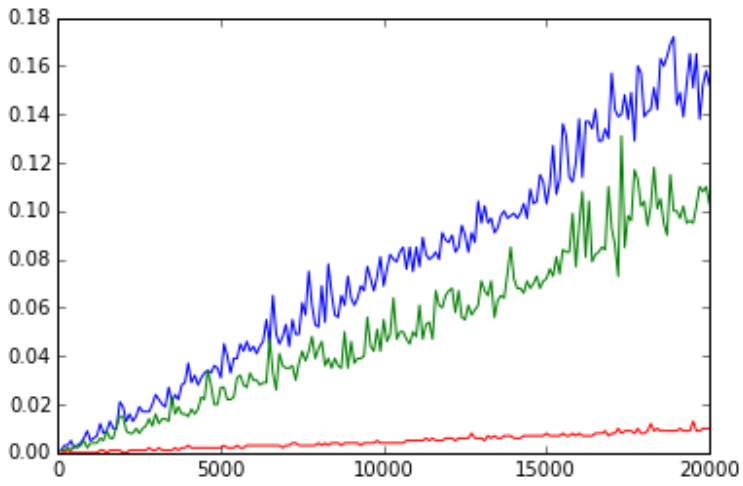
Complexité un peu pénible à calculer, mais sous-linéaire, avec un facteur meilleur que le trifusion (cf les simulations). Dans le pire cas, $C(n)=C(n-1)+O(n)$, ce qui donne assez clairement du $O(n^2)$ (ce sera le cas par exemple si on teste l'algorithme sur une liste déjà triée en prenant systématiquement le premier élément comme pivot), mais un calcul précis et probabiliste montre qu'on est très proche du meilleur cas en pratique.

2.3 Comparaison pratique de l'efficacité des différents tris.

Pour tester très concrètement les différents tris que nous avons programmés dans ce chapitre, j'ai simplement fait la chose suivante : pour chaque valeur de n comprise entre 100 et 2 000 avec un pas de 100, création d'une liste aléatoire de n éléments, qu'on trie avec chacune des procédures (en pratique, je trie en fait des listes différentes, car le fait qu'on effectue le tri en place pour la plupart va nous mettre régulièrement dans les pires cas si on les applique l'une après l'autre avec une même liste d'origine). On stocke le temps mis par chaque algorithme pour effectuer le tri, puis on trace les courbes correspondantes avec matplotlib (je vous laisse regarder le détail dans le fichier Python si vous le souhaitez). Sans surprise, les trois algorithmes naïfs sont battus à plate couture par les algos efficaces (le tri à bulles étant le moins bon de tous), et le QuickSort est légèrement plus rapide que le tri fusion. Si on met le sort de Python dans la comparaison, il est encore à peu près dix fois plus rapide que le Quicksort ! Le graphique ci-dessous représente les courbes obtenues (tri à bulles en rouge, tri insertion en vert, tri par sélection en bleu, on voit à peine les autres) :



Et un deuxième graphique pour comparer uniquement les deux tris optimisés avec le sort de Python, avec des valeurs de n allant cette fois jusqu'à 20 000 (en bleu, le tri fusion, en vert le tri rapide et en rouge le sort de Python, pour une liste de 20 000 éléments, on a un temps d'exécution proche d'un centième de seconde pour ce dernier, un dixième de seconde pour le tri rapide et 1.5 dixième de seconde pour le tri fusion) :



2.4 Application du tri rapide au calcul de médiane

On problème classique pouvant être résolu de façon efficace en exploitant l'algorithme de tri rapide est celui du calcul de la médiane d'une liste. Rappelons que la médiane est la valeur située « au milieu » des valeurs de la liste : si L contient un nombre impair n d'éléments, ce sera donc le $\frac{n+1}{2}$-ème élément dans l'ordre croissant, et si la liste en contient un nombre pair, on peut choisir le $\frac{n}{2}$-ème ou le suivant pour jouer le rôle de médiane. Une méthode évidente pour obtenir la valeur de cette médiane est de trier la liste, puis d'en extraire le bon élément. D'après ce qu'on a étudié dans ce chapitre, on aurait alors une complexité en $O(n\log_2(n))$ pour trouver la médiane. Peut-on faire mieux ? Intuitivement, ça devrait être le cas : pour obtenir le plus petit ou le plus grand élément d'une liste, ou même par exemple le troisième élément dans l'ordre croissant, on peut très bien procéder en temps linéaire (il s'agit simplement de calculer un ou plusieurs minimums), pourquoi ne serait-ce pas le cas pour la médiane ? De fait, on peut gagner du temps par rapport à un tri complet de la liste en effectuant uniquement le tri des morceaux de liste dans lesquels peut se trouver la médiane. On obtient l'algorithme suivant :

- on effectue sur la liste L la première partition initiant le tri rapide, et on note n la position où va se retrouver notre premier pivot dans la liste, et m la position de la médiane.
- si $m = n$, gros coup de pot, notre pivot était exactement égal à la médiane, on peut arrêter immédiatement l'algorithme.
- si $m < n$, on recherche la médiane, ou plutôt l'élément en position m (qui n'est plus médian dans cette sous-liste) dans $L[:n]$, sans se préocupper de la moitié droite de la partition.
- si $m > n$, au contraire, on va rechercher l'élément en position $m - n + 1$ (on ne tient pas compte du pivot) dans la liste $L[n+1:]$.
- on applique bien sûr récursivement l'algorithme si besoin, sachant que trouver la médiane dans une liste de longueur 1 est trivial.

La complexité de cet algorithme est loin d'être évidente à calculer (comme pour le tri rapide, cela dépend fortement de la position des pivots choisis aléatoirement), mais on peut prouver qu'il est bel et bien linéaire dans le cas moyen (le nombre moyen de comparaisons effectuées est majoré par $4n$). Une programmation possible en Python permettant de calculer plus généralement le k-ème plus petit élément d'une liste donnée et reprenant la fonction partition programmée plus haut (on appliquera bien sûr le programme avec $k = len(L)//2$ pour obtenir la médiane) :

```
def medianeplus(L,k) :
    if len(L)==1 :
        return L[0]
    n=partition(L,0,len(L))
    if n==k :
        return L[n]
    elif k<n :
        return medianeplus(L[ :n],k)
    else :
        return medianeplus(L[n+1 :],k-n-1)
```

Notez bien que si les appels récursifs mènent à appliquer la fonction à une liste de longueur 1 (cas d'initialisation de notre fonction récursive) alors on la valeur de k sera forcément devenue égale à 0 (sauf bien sûr si initialement on a pris un k plus grand que $len(L)$).