1) *Création d'un tableau aléatoire*

```
from random import randrange

def tab_alea(n,v_max):
    return [randrange(v_max) for k in range(n)]
```

Une fantaisie pythonnesque à éviter : `[randrange(v_max)]*n` remplira le tableau avec la même valeur partout, puisque `randrange` sera appelé une seule fois avant duplication !

2) *Tri par dénombrement*

a) Suivant l'indication de l'énoncé :

- je crée un tableau *occur* de taille val_max, initialement rempli de zéros, destiné à contenir le nombre d'occurrences dans t de chaque valeur possible (complexité $O(val\_max)$)
- je parcours t en incrémentant le nombre d'occurrences de chaque valeur rencontrée (complexité $O(n)$)
- je crée le tableau trié tt en partant d'un tableau vide et en l'augmentant, pour chaque valeur possible v, allant de 0 à $val\_max - 1$, de ladite valeur recopiée autant de fois que son nombre d'occurrences (complexité $O(n)$, car la somme du nombre d'occurrences vaut n ; si l'on compte la gestion du compteur de boucle, la complexité devient un $O(val\_max + n)$)
- je renvoie le tableau tt.

J'obtiens bien par construction les valeurs du tableau initial triées par ordre croissant, avec une complexité globale $O(val\_max + n)$.

b) J'applique l'algorithme décrit au **a)** :

```
def tri_den(t,val_max):
    occur=[0]*val_max
    for v in t:
        occur[v]+=1
    tt=[]
    for v in range(val_max):
        tt.extend([v]*occur[v])
    return tt
```

J'ai choisi de transmettre aussi val_max comme paramètre, pour insister sur le fait que cet algorithme est réservé aux tableaux contenant des valeurs cantonnées à un intervalle donné, ici $[0, val\_max[$. Il est dangereux d'envoyer un tableau quelconque et de déterminer son maximum en guise de val_max, qui risque d'être énorme. C'est cette limitation qui permet d'obtenir un algorithme de tri en $O(n)$ (pour val_max fixé !).

3) *Tri par sélection*

a) Voici une solution possible :

```
def indice_min(t,g,d):
    j=g
    for i in range(g+1,d):
        if t[i]<t[j]:
            j=i
    return j
```

```
def tri_sel(t):
    n=len(t)
    for k in range(n-1):
        j=indice_min(t,k,n)
        if k<j:
            t[k],t[j]=t[j],t[k]
    return t
```

Justification :

* la terminaison est assurée par les boucles inconditionnelles (on ne modifie pas le compteur dans le corps de la boucle !)
* la correction de *indice_min* s'obtient en prouvant par récurrence sur i l'invariant de boucle $\mathcal{P}_i$ suivant : "après l'exécution du corps de la boucle pour la valeur i, $t[j]$ est la plus petite des valeurs de la tranche $t\,[g:i]$"
* cela prouvé, la correction de *tri_sel* s'obtient en prouvant par récurrence sur k l'invariant de boucle $\mathcal{Q}_k$ suivant : "après l'exécution du corps de la boucle pour la valeur k, $t[0],...,t[k]$ sont les $k+1$ plus petites valeurs de t en ordre croissant" (où l'on voit qu'il suffit bien de s'arrêter à $k=n-2$, puisqu'alors $t\,[n-1]$ est nécessairement la plus grande valeur !)

Complexité : le corps de la boucle principale de la fonction *tri_sel* est exécuté $n-1$ fois et chaque appel à *indice_min* a une complexité en $O\,(n)$, d'où une complexité totale en $O\left(n^2\right)$.

On peut ici faire le calcul exact du nombre de comparaisons entre éléments du tableau. Il ne dépend pas du tableau initial et vaut : $\sum_{k=0}^{n-2}(n-k-1)=\dfrac{(n-1)\,n}{2}$

(en effet l'appel $indice\_min(t,g,d)$ se solde par $d-g-1$ comparaisons).

b) L'algorithme précédent se prête bien à une vision récursive : j'écris une fonction `tri_sel(t,k)` qui trie la tranche $t\,[k:]$:

* si $k=len\,(t)-1$, il n'y a rien à faire ;
* sinon, je détermine un indice j tel que $t\,[j]$ soit le plus petit élément de la tranche $t\,[k:]$, j'échange $t\,[k]$ et $t\,[j]$ et je trie (récursivement !)la tranche $t\,[k+1:]$.

Là encore, la détermination de j peut-être confiée à une fonction auxiliaire, que l'on peut aussi programmer récursivement pour rester dans le ton...

```
def indice_min(t,g,d):
    if g==d-1:
        return g
    else:
        j=indice_min(t,g+1,d)
        if t[g]<t[j]:
            return g
        else:
            return j
```

```
def tri_sel(t,k=0):
    if k<len(t)-1:
        j=indice_min(t,k,len(t))
        t[j],t[k]=t[k],t[j]
        tri_sel(t,k+1)
```

J'ai choisi d'écrire une **procédure** modifiant le tableau t, afin d'éviter les recopies de tranches de tableaux. J'ai profité de la syntaxe `k=0` pour le second paramètre, qui permet d'omettre ledit paramètre lors de l'appel initial : `tri_sel(t)` sera interprété comme `tri_sel(t,0)`...

Comme souvent, la justification des versions récursives est plus élégante : terminaison assurée par la décroissance stricte de la longueur du tableau à chaque appel récursif et correction prouvée par récurrence immédiate.

En contrepartie, ces versions sont plus gourmandes en mémoire, d'autant que la profondeur des appels récursifs est de l'ordre de n... Ainsi, dans Python, la limitation de cette profondeur (à 1 000, par défaut) sera vite atteinte ! On peut toutefois la modifier par la commande `sys.setrecursionlimit`.

Pour les calculs de complexité, on obtient (naturellement !) des relations de récurrence... Comptons les comparaisons entre éléments du tableau à trier. Soit γ_n (resp. c_n) le nombre de telles comparaisons effectuées par $indice\_min\,(t)$ (resp. $tri\_sel\,(t)$) pour t de longueur n :

$$\gamma_1=0 \quad \text{et} \quad \forall n\geq 2 \quad \gamma_n=\gamma_{n-1}+1 \quad ; \quad c_0=0 \quad \text{et} \quad \forall n\in\mathbb{N}^* \quad c_n=\gamma_n+c_{n-1}$$

d'où je déduis immédiatement

$$\forall n\in\mathbb{N} \quad \gamma_n=n-1 \quad \text{et} \quad c_n=\frac{n\,(n-1)}{2}.$$

4) *Version procédurale du tri fusion*

Les procédures `fusion(g,m,d)` et `tri(g,d)` seront encapsulées dans la définition de la procédure `tri_fusion(t)`, qui se limitera ainsi à l'appel `tri(0,len(t))` !

`tri(g,d)` s'écrit immédiatement, selon le principe du tri fusion, en supposant définie la procédure de fusion :

```
def tri(g,d):
    if g<d-1:
        m=(g+d)//2
        tri(g,m)
        tri(m,d)
        fusion(g,m,d)
```

Reste à programmer la fusion. Je reprends le principe exposé en cours, en stockant le résultat de la fusion dans la liste auxiliaire `aux`. Une fois que l'une des deux "moitiés" est épuisée, je profite de la remarque suivante pour économiser des copies redondantes : si c'est la moitié de gauche qui s'épuise en premier, c'est que les dernières valeurs de la moitié de droite sont déjà à leur place ; il est donc inutile de les rajouter à `aux` pour les remettre ensuite au même endroit dans t ! En revanche, si c'est la moitié de droite qui s'épuise en premier (sortie de la boucle `while` avec $j = d$), **il faut** ajouter à `aux` les dernières valeurs de la moitié gauche (les plus grandes) afin de les recopier à leur place correcte dans t.

```
def fusion(g,m,d):
    i,j=g,m
    aux=[]
    while i<m and j<d:
        if t[i]<t[j]:
            aux.append(t[i])
            i+=1
        else:
            aux.append(t[j])
            j+=1
    for k in range(i,m):
        aux.append(t[k])
    k=g
    for x in aux:
        t[k]=x
        k+=1
```

5) *Comptage expérimental du nombre de comparaisons*

Avant tout, il faut bien distinguer :

- le nombre de comparaisons effectuées, qui ne dépend que de l'algorithme tel qu'il est programmé
- le temps d'exécution, qui dépend du langage utilisé, de la version dudit langage, de la machine utilisée, des "occupations" de ladite machine pendant les tests.

Cela dit, un chronométrage peut permettre des comparaisons, si l'on exécute plusieurs programmes dans des conditions identiques.

Ainsi, on constate sans surprise (c'est son gros avantage) que le tri fusion est peu sensible au nombre d'ex-æquo dans le tableau : le nombre de comparaisons et le temps de calcul ne dépendent guère de la valeur maximale des éléments de tableau. On vérifie expérimentalement l'ordre de grandeur $n \log_2 n$ pour le nombre de comparaisons du tri fusion.

Par contre, on constate de grosses différences pour le tri rapide, selon la version utilisée et la structure du tableau :

- *pour des tableaux avec beaucoup d'ex æquo* ; les versions `quicksort1` et `quicksort2` (qui utilisent le nombre d'occurrences du pivot) s'avèrent plus rapides, c'était prévisible et pour de grands tableaux les autres versions atteignent plus vite la `recursionlimit`...
- *pour des tableaux avec peu d'ex æquo* : la version `quicksort1`, qui parcourt le tableau trois fois pour faire la partition, effectue bien plus de comparaisons (et encore notre procédé de décompte ne prend pas en compte la complexité des `count` !), mais son temps d'exécution n'augmente pas en proportion, car les manipulations de listes par Python sont optimisées ; les versions `quicksort3` et `quicksort` sont celles qui effectuent le moins de comparaisons, la plus rapide étant en général `quicksort3`, là encore du fait de la gestion efficace des listes par Python (le tri en place de `quicksort` est pourtant plus rapide avec la plupart des langages...)

On vérifie aussi l'ordre de grandeur $2n \ln n$ pour le nombre de comparaisons des deux dernières versions du tri rapide, avec des tableaux aléatoires contenant peu d'ex æquo.

N.B. : dans les fichiers Python fournis avec ce corrigé, le lecteur remarquera l'usage de l'instruction `try:...except...: else:...` qui permet d'*intercepter* une erreur prévisible (typiquement ici le débordement de la pile des appels récursifs). Hors programme mais parfois utile...

6) *Tri de Shell et question des seuils*

a) Voici une implémentation du tri de Shell :

```
1  def tri_Shell(t):
2      pas=[p for p in [1,4,10,23,57,132,301,701] if len(t)>2*p]
3      p=1612
4      while len(t)>2*p:
5          pas.append(p)
6          p=int(p*2.3)
7      pas.reverse()
8      for p in pas:
9          for k in range(p):
10             i=k+p
11             while i<len(t):
12                 temp=t[i]
13                 j=i
14                 while j-p>=k and t[j-p]>temp:
15                     t[j]=t[j-p]
16                     j-=p
17                 t[j]=temp
18                 i+=p
```

- ∗ La première phase (lignes 2 à 7) consiste à déterminer la liste des pas à utiliser. Je compare la longueur du tableau à $2p$ (afin qu'il y ait au moins deux valeurs à comparer pour la première valeur de p !). Pour éviter une boucle, je me permets d'utiliser la méthode `reverse`, qui remplace une liste par son *image miroir*.
- ∗ Ensuite, pour chaque valeur p du pas, je trie les sous-tableaux définis dans l'énoncé : $k \in [\![0, p-1]\!]$ désigne l'indice de la première case du sous-tableau auquel j'applique le tri par insertion (lignes 10 à 18).

b) Pour combiner l'algorithme précédent à celui du tri rapide sans recopier de tranche, j'en écris une version "encadrée" `Shell(t,g,d)`, qui applique le tri de Shell à la tranche $t\,[g, d]$, lorsque $d - g$ est inférieur à un certain seuil.

Comme les seuils "raisonnables" sont de l'ordre de quelques centaines, je peux simplifier la construction de la liste des pas.

Quelques tests avec Python montrent que, pour de grands tableaux, le tri de Shell du **a)** est moins rapide que le tri rapide, mais toutefois concurrentiel, avec une complexité qui semble de l'ordre de $n \ln n \ldots$

Toujours avec Python, le programme hybride `quickShell` ci-dessous s'avère plus rapide que le tri de Shell purement itératif, avec des choix "raisonnables" du seuil, mais reste un peu plus lent que tri rapide, sans doute parce que les boucles ne s'exécutent pas très vite avec un langage *interprété* comme Python...

Noter qu'avec Pascal (langage *compilé* !), on obtient la hiérarchie attendue : sur un exemple de tableau de 10 millions de valeurs, le tri de Shell s'effectue en 12,9s, le tri rapide en 2,8s et tri hybride en 2,6s.

```
def Shell(t,g,d):
    pas=[p for p in [701,301,132,57,23,10,4,1] if d-g>2*p]
    for p in pas:
        for k in range(p):
            i=g+k+p
            while i<d:
                temp=t[i]
                j=i
                while j-p>=g+k and t[j-p]>temp:
                    t[j]=t[j-p]
                    j-=p
                t[j]=temp
                i+=p

def partition(t,g,d):
    p=g
    pivot=t[g]
    for k in range(g+1,d):
        if t[k]<pivot:
            p+=1
            t[p],t[k]=t[k],t[p]
    t[p],t[g]=t[g],t[p]
    return p

def quickShell(t,seuil=100):
    def tri(g,d):
        if d-g<seuil:
            Shell(t,g,d)
        else:
            p=partition(t,g,d)
            tri(g,p)
            tri(p+1,d)

    tri(0,len(t))
    return t
```