De nombreux algorithmes nécessitent de savoir déterminer si un élément appartient à un ensemble important de données. Comme nous l'avons vu précédemment, une méthode efficace de recherche dans un tableau est la méthode dichotomique, qui ne fonctionne que dans le cas où les valeurs du tableau ont été triées au préalable. Il faut donc pour cela pouvoir trier un tableau, ou une liste, avec une complexité telle que le gain obtenu par la recherche dichotomique ne soit pas totalement perdu lors du tri.
Une deuxième contrainte, dans le cas où le nombre de données à trier est très important, est de ne pas avoir besoin d'un espace mémoire plus grand que celui occupé par le tableau initial : lorsque le tri s'effectue en modifiant l'ordre des éléments du tableau initial, sans avoir besoin de créer un autre tableau de « stockage », on parle de tri sur place.

Le programme d'I.P.T indique qu'il y a trois algorithmes à connaître par cœur : tri par insertion, tri fusion, tri rapide.

Pour tester les différents algorithme de tri, on utilisera la fonction suivante qui crée une liste de n entiers, tous entre 0 et 100, de façon aléatoire :

```
import random as rd

def CreerListe(n) :
    L = []
    for _ in range(n) :
        L.append(rd.randint(0,100))
    return L
```

Pour comparer les différents algorithmes de tri introduits dans ce chapitre, on comptera le nombre de comparaisons effectuées par ces algorithmes lorsqu'ils sont appliqués à des listes de longueur n.

I Algorithmes quadratiques

1. Un tri bien naïf

Une première idée pour trier une liste est la suivante : on cherche la position du plus petit élément de la liste, on le sort de la liste et on recommence à trier la liste restante ; dans le même temps, on crée une autre liste dans laquelle on range les éléments dans l'ordre (croissant).
La fonction suivante renvoie l'indice du plus petit élément de la liste :

```
def IndiceMin(L) :
    ind = 0
    for i in range(1,len(L)) :
        if L[ind] > L[i] :
            ind = i
    return ind
```

Et celle ci trie la liste avec la méthode exposée ci-dessus :

```
def TriNaif(L) :
    T = []
    for i in range(len(L)) :
        ind = IndiceMin(L)
        T.append(L[ind])
        L = L[:ind]+L[ind+1:]
    return T
```

Si on s'intéresse au nombre de comparaisons à effectuer lors du tri pour en déterminer la complexité, il faut $k-1$ comparaisons lors de l'utilisation de la fonction `IndiceMin` sur une liste de longueur k. De plus, cette fonction sera appelée n fois pour une liste de longueur n par la fonction `TriNaif`, aves des listes de longueurs n, puis $n-1$, ...,1. Au total, le nombre de comparaisons pour une liste de longueur n sera $C(n) = \dfrac{n(n-1)}{2}$. La complexité est donc $O\left(n^2\right)$.
En outre cette méthode nécessite de construire une deuxième liste, ce qui risque de poser des problèmes de place en mémoire si le nombre de données à trier est très important. Il est à noter que la liste `L` n'est pas modifiée lors de l'utilisation de la fonction `TriNaif` car la commande `L = L[:ind]+L[ind+1:]` crée une variable local de même nom ; c'est elle qui est modifiée.

2. Le tri par sélection

Pour contourner le deuxième problème du tri précédent, on peut trier une liste `L` en plaçant le plus petit élément de la liste à la première place, avant de continuer à trier la liste à partir du rang 1 (la liste est indicée à partir de 0) : il suffit pour cela, quand on a trouvé le plus petit élément, de l'échanger avec le premier élément de la liste.
Un programme utilisant cette idée est le suivant :

```
def TriSelection(L) :
    for i in range(len(L)-1) : # pas de permutation à faire quand il reste un seul
        élément
        ind = i + IndiceMin(L[i:]) # la fonction IndiceMin compte ici à partir de i
        L[i],L[ind] = L[ind],L[i] # on échange les éléments
```

Le nombre de comparaisons à effectuer par `IndiceMin` est $n-i-1$ pour chaque valeur de l'entier `i`, qui varie entre 0 et $n-2$. La complexité est donc $C(n) = \sum_{i=0}^{n-2}(n-i-1)$ qui reste donc du même ordre $O\left(n^2\right)$, mais on a gagné en espace de stockage.

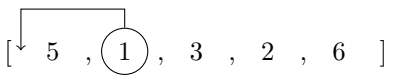
3. Tri par insertion (au programme)

C'est le tri du joueur de cartes. L'idée du tri par insertion est le suivant : on compare les deux premiers éléments de la liste, s'ils ne sont pas dans le bon ordre, on les permute, sinon, on les laisse tels quels. On considère ensuite le troisième élément ; on va l'insérer à sa place : on le met au début s'il est plus petit que les deux autres (il faut alors décaler les deux autres pour faire de la place), on l'insère au milieu des deux autres en décalant le second ou on le laisse à la fin s'il est plus grand que les deux autres. Après cette étape, les trois premiers éléments seront dans l'ordre croissant et on continue ainsi de suite.
Un code réalisant ce tri (dans l'ordre croissant) est le suivant :

```
def TriInsertion(L) :
    for i in range(1,len(L)) :
        rang_ins = i
        elt = L[i]
        while rang_ins > 0 and  L[rang_ins-1] > elt :
            L[rang_ins] = L[rang_ins-1]
            rang_ins -=1
        L[rang_ins] = elt
```

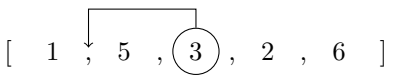
Un exemple d'exécution : si on trie la liste `[5,1,3,2,6]`

— `i = 1` : on déplace le `1`



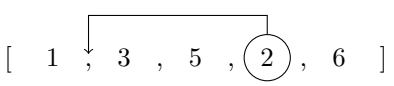
rang_ins	elt	L[rang_ins-1]	L
1	1	5	[5,5,3,2,6]
0	1		[1,5,3,2,6]

— `i = 2` : on déplace le `3`



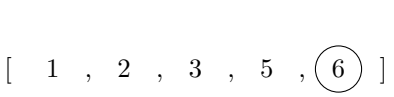
rang_ins	elt	L[rang_ins-1]	L
2	3	5	[1,5,5,2,6]
1	3	1	[1,3,5,2,6]

— `i = 3` : on déplace le `2`



rang_ins	elt	L[rang_ins-1]	L
3	2	5	[1,3,5,5,6]
2	2	3	[1,3,3,5,6]
1	2	1	[1,2,3,5,6]

— `i = 4` : le `6` est bien placé



rang_ins	elt	L[rang_ins-1]	L
4	6	5	[1,2,3,5,6]

Preuves :

— La terminaison de l'algorithme est assurée pour les raisons suivantes : la boucle `for` se termine forcément et la boucle `while` se termine en remarquant que `rang_ins` est un variant de boucle.

— Pour prouver la correction de l'algorithme, on peut vérifier que `L[0]`$\leqslant$`L[1]`$\leqslant \ldots \leqslant$`L[i]` est un invariant de boucle.

Calculs de complexité :

— Pour une liste de longueur n, la complexité, dans le pire des cas (qui correspond à une liste initiale rangée dans l'ordre décroissant) est

$$C_{\text{pire}}(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

ce qui donne à nouveau un $O\left(n^2\right)$.

— Dans le meilleur des cas, où la liste est déjà triée dans le bon ordre, seule la boucle `for` est exécutée et la complexité est $C_{\text{meilleur}}(n) = n$.

— Pour le calcul de la complexité en moyenne, on peut raisonner de la façon suivante : pour trier une liste de n éléments, il faut d'abord trier une liste de $n-1$ éléments puis insérer le dernier élément à sa place. Le nombre moyen de comparaisons à effectuer pour placer le dernier élément est $\frac{n-1}{2}$ donc $C_{\text{moyen}}(n) = \frac{n-1}{2} + C_{\text{moyen}}(n-1)$.

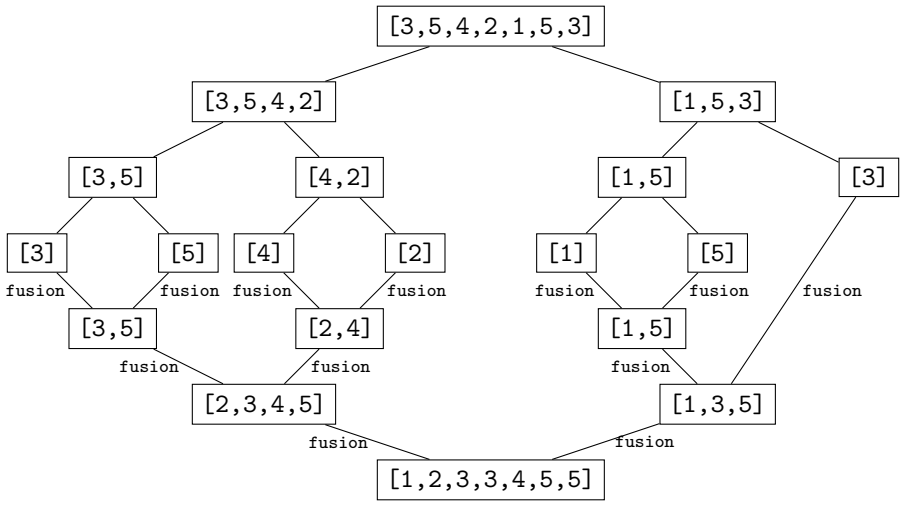
On trouve alors $C_{\text{moyen}}(n) = \frac{n(n-1)}{4}$; la complexité en moyenne est donc, comme dans le pire des cas $O\left(n^2\right)$.

Toutes les modifications se font sur la liste `L` elle-même donc c'est une procédure de tri sur place.

II « Diviser pour mieux régner »

1. Tri fusion (au programme)

Le principe du tri fusion est le suivant : on divise la liste en deux sous-listes (de même longueur si la liste initiale est de longueur paire), on trie chacune des ses sous-listes puis on fusionne les deux sous-listes triées en insérant les éléments de la deuxième liste dans la première à la bonne place.



On peut commencer par programmer une fonction qui insère une liste `L2` triée, dans une liste `L1` triée elle aussi.
Voici un code itératif de la procédure de fusion : on rajoute à une liste auxiliaire le plus petit des éléments présent dans `L1` ou `L2` jusqu'à vider une des listes, il reste ensuite à rajouter les éléments de la liste qui n'a pas été vidée.

```
def Fusion(L1,L2) :
    fus = []
    i,j = 0,0
    while i < len(L1) and j < len(L2) :
        # L1 et L2 n'ont pas été explorées en entier
        if L1[i] < L2[j] :
            fus.append(L1[i])
            i += 1
        else :
            fus.append(L2[j])
            j += 1
    if i == len(L1) :
        # L1 a été totalement rajoutée
        fus += L2[j:]
    else :
        # alors c'était L2
        fus += L1[i:]
    return fus
```

Preuves :
- La terminaison de cette procédure est assurée par l'incrémentation de `i` ou de `j` (un variant de boucle est par exemple `len(L1)+len(L2)-(i+j)`).
- La correction de l'algorithme peut être prouvée en remarquant que `fus[0]`$\leqslant$`fus[1]`$\leqslant\cdots\leqslant$`fus[k]` est un invariant de boucle, où `k` est le nombre de boucles `while` effectuées ; la concaténation finale éventuelle conservera bien sûr l'ordre des éléments de la liste.

Calculs de complexité :
- La complexité de cette procédure est, pour une liste `L1` à p éléments et une liste `L2` à q éléments, dans le pire des cas (la boucle `while` épuise une liste entière et laisse un seul élément dans la deuxième) $F_{\text{pire}}(p,q) = p+q-1$
- Dans le meilleur des cas (la plus grande des deux listes est laissée intacte par la boucle `while`), la complexité est $F_{\text{meilleur}}(p,q) = \min(p,q)$.

A l'aide de cette procédure `Fusion_rec`, on peut facilement programmer le tri fusion, de façon récursive :

```
def TriFusion_rec(L) :
    if len(L) <= 1 :
        return L
    else :
        n = len(L)//2
        L1,L2 = L[:n],L[n:]
        return Fusion(TriFusion_rec(L1),TriFusion_rec(L2))
```

Preuves :
- Ici la terminaison est assurée par le variant de boucle `n`.
- Une récurrence (forte) sur la longueur de la liste à trier assure la correction de l'algorithme.

Calculs de complexité : La complexité T d'une telle procédure est donnée par

$$T(n) = T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + T\left(\left\lceil\frac{n}{2}\right\rceil\right) + F\left(\left\lfloor\frac{n}{2}\right\rfloor, \left\lceil\frac{n}{2}\right\rceil\right)$$

- Si $2^p \leqslant n < 2^{p+1}$, on a, dans le pire des cas

$$T_{\text{pire}}(n) \leqslant T_{\text{pire}}\left(2^{p+1}\right) = 2T_{\text{pire}}\left(2^p\right) + 2^{p+1}$$

  On en déduit alors $T_{\text{pire}}\left(2^p\right) = \left(T_{\text{pire}}(1)+p\right)\times 2^p$ ce qui donne $T_{\text{pire}}(n) = O\left(n\log_2(n)\right)$.
- Un calcul similaire donne $T_{\text{meilleur}} = \Theta(n\log_2(n))$.
- Les complexités dans le meilleur et dans le pire des cas étant du même ordre de grandeur, la complexité en moyenne est elle aussi en $O(n\log_2(n))$.

Les défauts d'une telle procédure sont les suivants : la programmation récursive qui limite la taille des listes que l'ont veut trier et surtout l'utilisation d'une liste annexe pour la fusion qui est très gourmande en espace mémoire dans le cas d'une grande liste (ce n'est pas une procédure de tri sur place).
Ces deux défauts peuvent être corrigé en
- programmant des fonctions de fusion et de tri itératives : en fait on peut partir de sous-listes, toutes de taille 1, et les fusionner deux par deux jusqu'à ce que la taille de la liste obtenue soit égale à la taille de la liste de départ.
- programmer une fusion sur place : si on suppose que deux sous-listes `L[p,q]` et `L[q,r]` sont triées, on peut les fusionner sur la liste `L[p,r]` en décalant les termes de la liste `L[p,q]` lorsqu'on a besoin d'y insérer des termes de l'autre sous-liste comme on l'a déjà vu dans la procédure de tri par insertion.

2. Tri rapide (au programme)

L'idée du tri rapide est la suivante : on choisit un pivot dans la liste à trier (le premier élément de la liste par exemple), on divise la liste en deux sous-listes, la première constituée des éléments inférieurs ou égaux au pivot, la seconde constituée des éléments strictement plus grands que le pivot, et on trie récursivement les deux sous-listes. L'avantage de ce tri est de trier sur place : on crée les deux sous-listes en déplaçant les éléments de la liste.

On commence par programmer une fonction `Partition(L,gauche,droite)` qui réarrange les éléments de la liste `L` d'indices compris entre `gauche` et `droite`, en prenant comme pivot `L[gauche]` ; à la sortie, on récupère la liste `L` réarrangée avec les éléments inférieurs ou égaux au pivot placés aux indices entre `gauche` et `indice_inf`.

```
def Partition(L,gauche,droite) :
    pivot = L[gauche]
    dernier_inf = gauche # indice du dernier élément inférieur au pivot
    for i in range(gauche+1,droite+1) :
        if L[i] < pivot :
            dernier_inf += 1
            L[i],L[dernier_inf] = L[dernier_inf],L[i]
```

```
    # on met le pivot entre les deux sous-listes
    L[gauche],L[dernier_inf] = L[dernier_inf],L[gauche]
    # on renvoie l'indice final du pivot
    return dernier_inf
```

Un exemple d'exécution : si on applique cette fonction à la liste `L=[3,5,4,2,1,5,3]` avec `gauche=0` et `droite=6` (ce qui sera la première étape du tri rapide programmé plus loin)

pivot	i	L[i]	dernier_inf	L
3	0	3	0	[3,5,4,2,1,5,3]
	1	5	0	[3,5,4,2,1,5,3]
	2	4	0	[3,5,4,2,1,5,3]
	3	2	1	[3,2,4,5,1,5,3]
	4	1	2	[3,2,1,5,4,5,3]
	5	5	2	[3,2,1,5,4,5,3]
	6	3	2	[3,2,1,5,4,5,3]
			2	[1,2,3,5,4,5,3]

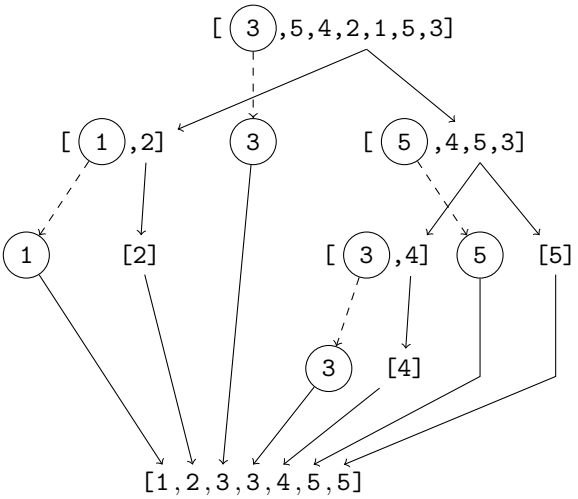
Preuves :
- La terminaison de cette procédure est assurée (boucle `for`)
- La correction est prouvée en utilisant le variant de boucle suivant : si $1 \leqslant$`j`$\leqslant$`dernier_inf` alors `L[j]`$<$`pivot` et si `dernier_inf`$<$`j`$\leqslant$`i` alors `L[j]`$\geqslant$`pivot`.

Calcul de complexité : Le nombre de comparaisons à effectuer pour partitionner une liste de longueur n (`=droite-gauche+1`) est $n-1$ dans tous les cas (donc aussi en moyenne).
On programme alors le tri :

```
def TriRapide_rec(L,gauche,droite) :
    if gauche < droite : # reste au moins 2 éléments
        rang = Partition(L,gauche,droite)
        # on trie ce qui est strictement à gauche du pivot
        TriRapide_rec(L,gauche,rang-1)
        # on trie ce qui est strictement à droite du pivot
        TriRapide_rec(L,rang+1,droite)
```

Pour une liste `L` de 10 éléments, il faudra taper `TriRapide_rec(L,0,9)` et on affiche la liste triée avec `print(L)` après.

Un exemple d'exécution : toujours avec la liste `[3,5,4,2,1,5,3]`, on aura le schéma suivant (tout se fait sur place mais pour plus de visibilité, on représente la liste de façon « éclatée »)



Preuves :
- La terminaison est prouvée par le variant de boucle `droite-gauche`.
- La correction est prouvée par une récurrence (forte) sur la longueur de la liste.

Calculs de complexité :
- Dans le pire des cas, la liste est déjà triée donc le pivot reste toujours à sa place mais après avoir été comparé à tous les autres éléments de la liste. La complexité dans le pire des cas est donc $C_{\text{pire}}(n) = \sum_{i=1}^{n} i - 1 = \dfrac{n(n-1)}{2}$ c'est-à-dire $O\left(n^2\right)$.

— Dans le meilleur des cas, le pivot va, à chaque étape, partager la liste en deux sous-listes de même longueur : il faut $n-1$ comparaisons pour déterminer la position du pivot avant de trier les deux sous listes. On a donc

$$C_{\text{meilleur}}(n) = n - 1 + C_{\text{meilleur}}\left(\left\lfloor \frac{n}{2} \right\rfloor - 1\right) + C_{\text{meilleur}}\left(\left\lceil \frac{n}{2} \right\rceil - 1\right),$$

ce qui donne $C_{\text{meilleur}}(n) = \Theta(n \log_2(n))$.

— Pour la complexité en moyenne, on raisonne ainsi : le coût de la procédure `Partition` est n, la répartition des éléments par rapport au `pivot` étant équiprobable, la probabilité que la première sous-liste soit de longueur i est $\frac{1}{n-1}$, le coût moyen du tri d'une telle liste est $C_{\text{moyen}}(i)$, le coût moyen du tri de la deuxième liste est alors $C_{\text{moyen}}(n-i)$. On a donc, avec $C_{\text{moyen}}(1) = 0$,

$$C_{\text{moyen}}(c) = n + \frac{1}{n-1}\sum_{i=1}^{n-1} C_{\text{moyen}}(i)C_{\text{moyen}}(n-i) = 2 + \frac{n}{n-1}C_{\text{moyen}}(n-1)$$

On en déduit $\frac{1}{n}C_{\text{moyen}}(n) = 2\sum_{k=1}^{n}\frac{1}{k} \sim \ln(n)$ donc $C_{\text{moyen}}(n) = \Theta(n\ln(n))$ est du même ordre de grandeur que la complexité dans le meilleur des cas.

Pour éviter de tomber dans le pire des cas, on peut essayer de choisir le `pivot` de façon plus astucieuse : un moyen simple d'éviter de prendre un `pivot` qui soit le plus petit des éléments de la liste est de choisir comme `pivot` la valeur médiane des trois éléments `L[gauche]`, `L[droite]` et `L[(gauche+droite)//2]`. Ce choix va coûter 3 comparaisons mais permet d'éviter que la taille d'une des deux sous-liste soit $n-1$ (mais elle pourrait quand même être $n-2$).
Ce tri est, contrairement au tri fusion, un tri sur place.

III Choix de la méthode de tri

Les complexités correspondent à un comportement asymptotique pour des listes de grande taille et permettent pour un algorithme donné de prédire l'évolution du temps de calcul quand par exemple on double cette grande taille. Cependant, les constantes multiplicatives, qui n'apparaissent pas dans les calculs d'ordre de grandeur de complexité, peuvent jouer un rôle important dans le choix d'une méthode de tri. Pour les grandes listes, on utilise souvent le tri rapide plutôt que le tri fusion (ce qui justifie son nom). En revanche, pour des listes de taille plus faible, le tri par insertion devient plus rapide.
D'autres notions interviennent comme la stabilité d'une méthode. Pour une liste peu désordonnée, on a par exemple intérêt à utiliser le tri par insertion ou le tri fusion, stables, plutôt que le tri rapide, instable, qui va, lors d'une modification, désordonner certains éléments.
L'animation proposée prend une liste de taille n contenant tous les n premiers entiers, mais initialement désordonnée (il existe des valeurs de i telles que $L[i] \neq i$), et les trie selon la méthode choisie. L'évolution de la liste est représentée sur le graphe où l'indice de l'élément est porté en absisses et la valeur de l'élément en ordonnées. Ainsi, quand la liste est triée, le graphe représente la première bissectrice. On peut alors comparer graphiquement les méthodes et leur stabilité, ainsi que la rapidité du temps de calcul et l'influence de la taille de la liste sur ce temps.

IV Exercice : Tri d'un jeu de cartes

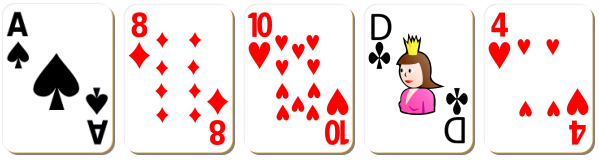
Une carte est caractérisée par sa couleur (C,K,P,T pour respectivement « cœur », « carreau », « pique » et « trèfle ») ainsi que sa valeur (de 2 à 10 puis 11 pour « valet », 12 pour « dame », 13 pour « roi » et 14 pour « as »). Une carte sera donc un doublet ('couleur',valeur). Par exemple l'as de pique sera représenté par `('P',14)`. Un jeu de 52 cartes est donc une liste de 52 cartes. Cette liste sera considérée comme parfaitement ordonnée si elle contient d'abord les cœurs dans l'ordre croissant, puis les carreaux, les piques et enfin les trèfles.

Énoncé

1. Créer une fonction `JeuInit()` qui renvoie un jeu complet parfaitement ordonné.

2. Programmer une fonction `melange(jeu)` qui déplaçe aléatoirement les éléments d'une liste `jeu` fournie en entrée. On utilisera la fonction `randint(0,N)` de la bibliothèque `random` qui renvoie un entier pris au hasard entre 0 et N inclus. On pourra utiliser `L.pop(j)` qui supprime l'élément d'indice `j` de la liste `L`.
On peut maintenant battre les cartes d'un jeu quelconque.

3. Programmer maintenant le tri par insertion (sur place), puis l'appliquer au jeu complet « battu ». Appliquer également cette méthode à la main suivante :



4. Donner les différentes étapes permettant de trier la main précédente par le tri fusion.
5. Donner les différentes étapes permettant de trier la main précédente par le tri rapide.

Corrigé

1.

```
def JeuInit():
    jeu = []
    for elt in ['C','K','P','T']:
        jeu += [(elt,i) for i in range(2,15)]
    return jeu
```

2.

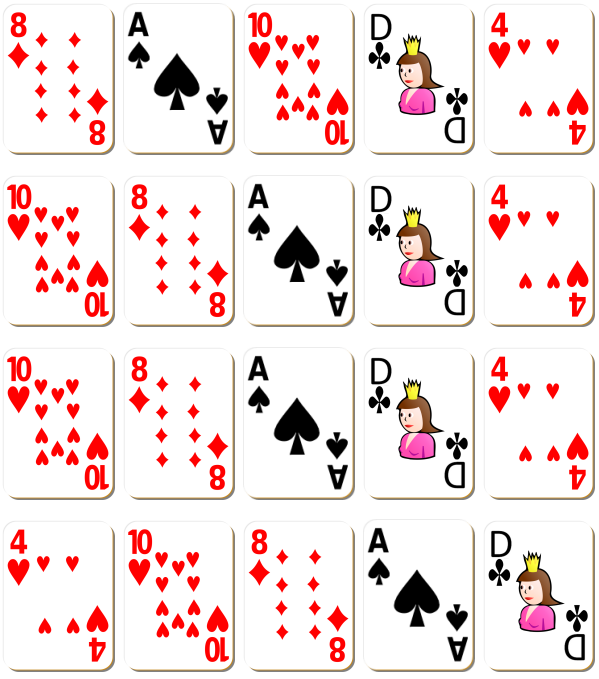
```
import random as rd

def melange(jeu):
    temp = jeu[:] # il faut créer une copie du jeu
    for i in range(len(jeu)):
        j = rd.randint(0,len(temp)-1)
        jeu[i] = temp[j]
        temp.pop(j) # on enlève l'élément déjà tiré au sort
```

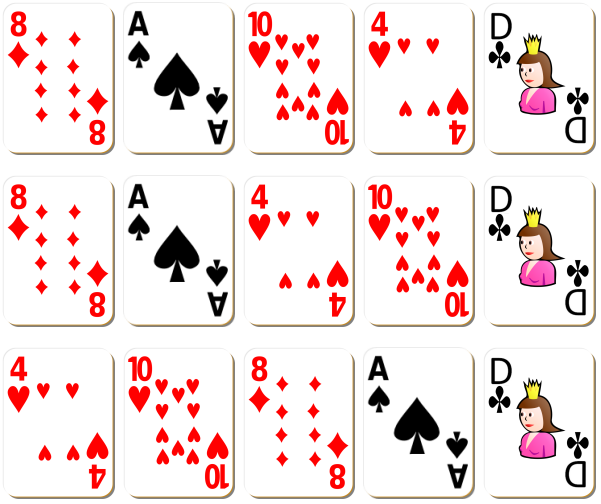
3.

```
def TriInsertion(L):
    for i in range(1,len(L)):
        elt = L[i]
        rang_ins = i
        while elt < L[rang_ins-1] and rang_ins > 0:
            L[rang_ins] = L[rang_ins-1]
            rang_ins -= 1
        L[rang_ins] = elt
```

On donne ci-dessous les étapes correspondant à la boucle `for` :



4. Les étapes lors du dépilage sont :



5. Pour la première partition, l'as de pique est le pivot :

On trie de la même façon ce qu'il y a à gauche et à droite du pivot. Pour la partie de gauche, Le 4 de cœur est le pivot : rien n'est modifié. La partie de droite est déjà triée.

Il reste à à trier le huit de carreau et le 10 de cœur :