Un peu d’algorithmique...

- Des tris (à bulle, par insertion, rapide, par fusion, par tas)
- Du backtracking (le problème des 8 reines)

Logique et approche mathématique de la programmation, cours 3
M. Rigo

Bubble sort – tri à bulles – tri par propagation

- Comparer répétitivement les éléments d'une liste
- Lorsque deux éléments consécutifs sont mal triés, les permuter
- On recommence jusqu'à ce que la liste soit triée

Wikipédia : "*Le tri à bulles est souvent enseigné en tant qu'exemple algorithmique, car son principe est simple. Mais c'est le plus lent des algorithmes de tri communément enseignés, et il n'est donc guère utilisé en pratique.*"

Première passe

(**5 1** 4 2 8) → (**1 5** 4 2 8), échange
(1 **5 4** 2 8) → (1 **4 5** 2 8), échange
(1 4 **5 2** 8) → (1 4 **2 5** 8), échange
(1 4 2 **5 8**) → (1 4 2 **5 8**),

Après *k* passes,
les *k* derniers éléments
sont bien triés.

Deuxième passe

(**1 4** 2 5 8) → (**1 4** 2 5 8),
(1 **4 2** 5 8) → (1 **2 4** 5 8), échange
(1 2 **4 5** 8) → (1 2 **4 5** 8),
(1 2 4 **5 8**) → (1 2 4 **5 8**),

Troisième passe : plus rien n'est échangé, fin de l'algorithme

En python, cela peut ressembler à ...

```
def bubblesort(liste):
    swap = True # retient s'il y a eu échange
    while swap:
        swap = False
        for i in range(0,len(liste)-1):
            if liste[i]>liste[i+1]:
                liste[i], liste[i+1] = liste[i+1], liste[i]
                swap = True
    return(liste)
```

Ajout d'une varibale local pour compter
le nombre total de passages dans la boucle.
On compte donc le nombre global d'opérations effectuées.

```
def bubblesort(liste):
    swap = True # retient s'il y a eu échange
    nb_op = 0   # estimer la complexité
    while swap:
        swap = False
        for i in range(0,len(liste)-1):
            nb_op += 1
            if liste[i]>liste[i+1]:
                liste[i], liste[i+1] = liste[i+1], liste[i]
                swap = True
    print(nb_op)
    return(liste)
```

Un programme “complet” :

```
import random

# générer une liste "aléatoire"
l = []
for x in range(101):
    l.append(random.randint(1,501))

def bubblesort(liste):
    swap = True # retient s'il y a eu échange
    nb_op = 0   # estimer la complexité
    while swap:
        swap = False
        for i in range(0,len(liste)-1):
            nb_op += 1
            if liste[i]>liste[i+1]:
                liste[i], liste[i+1] = liste[i+1], liste[i]
                swap = True
    print(nb_op)
    return(liste)

print(l)
print(bubblesort(l))
```

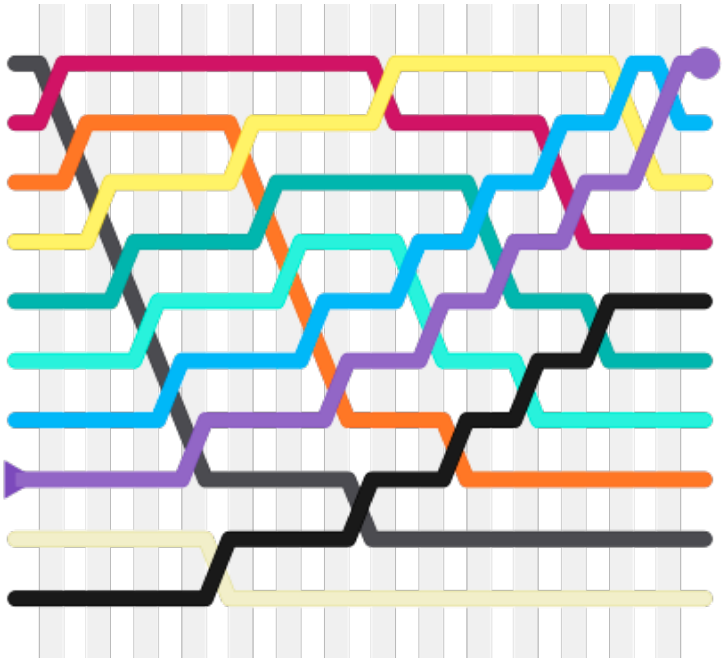
On optimise “un peu”, inutile de parcourir à chaque passe, la liste dans son entièreté → ajout d’une variable *fin*

```
def bubblesort2(liste):
    swap = True # retient s'il y a eu échange
    fin = len(liste)-1 # jusqu'où parcourt-on la liste
    while swap:
        swap = False
        for i in range(0,fin):
            if liste[i]>liste[i+1]:
                liste[i], liste[i+1] = liste[i+1], liste[i]
                swap = True
        fin -= 1
    return(liste)
```

https://en.wikipedia.org/wiki/Bubble_sort

https://fr.wikipedia.org/wiki/Tri_à_bulles

https://www.youtube.com/watch?v=lyZQPjUT5B4



Prouver qu'un algorithme est correct

- Prouver que l'algorithme s'achève (le problème de l'arrêt est indécidable)
- S'il s'achève, c'est avec le bon résultat

Complexité de l'algorithme

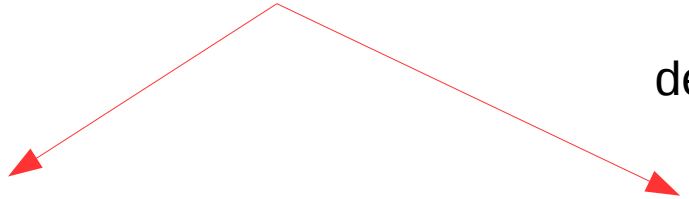
- Mémoire consommée (*complexité spatiale*)
- Nombre d'opérations "élémentaires" réalisées (*complexité temporelle*)

Complexité dans le pire cas (worst case)
Complexité en moyenne (plus difficile à estimer)

Pour le bubble sort : $O(n^2)$

Effet de bord en Python, passage d'une liste (objet muable)

Il faut déterminer si les fonctions modifient ou non la liste qui a été transmise...
Quel comportement veut-on pour notre fonction ?
On peut aussi travailler avec une copie (attention si longue liste)



```
def func1(list):
    print list
    list = [47,11]
    print list

fib = [0,1,1,2,3,5,8]

func1(fib)
```

```
[0, 1, 1, 2, 3, 5, 8]
[47, 11]

    print(fib)
[0, 1, 1, 2, 3, 5, 8]
```

Nouvelle affectation,
une variable locale
a été créée !

```
def func2(list):
    print list
    list += [47,11]
    print list

fib = [0,1,1,2,3,5,8]

func2(fib)
```

```
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 1, 2, 3, 5, 8, 47, 11]

    print(fib)
[0, 1, 1, 2, 3, 5, 8, 47, 11]
```

On a transmis “l'adresse” / l'alias
de la liste, Python l'utilise !

Effet de bord en Python, passage d'une liste (objet muable)

Il faut déterminer si les fonctions modifient ou non la liste qui a été transmise...
Quel comportement veut-on pour notre fonction ?
On peut aussi travailler avec une copie (attention si longue liste)

```
l=[7,9,2,3,5]
bubblesort(l)
print(l) # l a été modifié

l=[7,9,2,3,5]
copie=l[:]
bubblesort(copie)
print(l) # l est intact
print(copie)
```

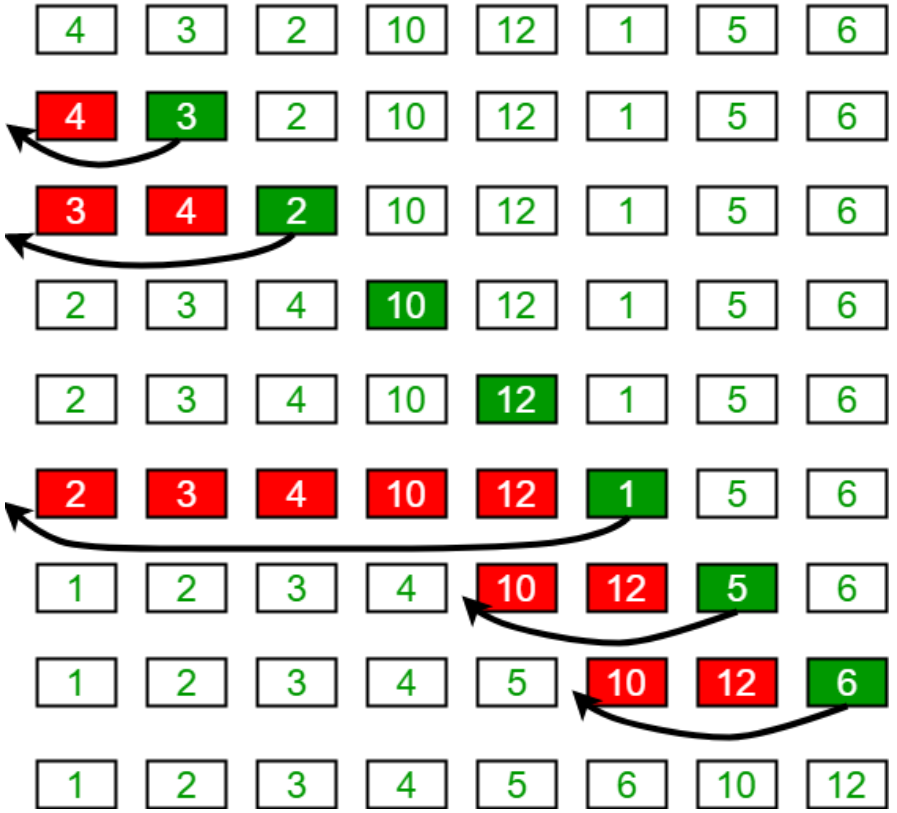
On pourrait aussi travailler avec une copie au sein de la fonction
(on évite ainsi les effets de bord)

```
def bubblesort(liste):
    copie=liste[:]
    ...
```

Tri par insertion

- On parcourt la liste à trier du début à la fin.
- Quand on considère le *i*-ème élément, les éléments qui le précèdent sont déjà triés.
- Si cet élément est < que le précédent, l'insérer à la bonne position parmi les *i* premiers.

Wikipédia : "*En général, le tri par insertion est beaucoup plus lent que d'autres algorithmes. Il est cependant considéré comme le tri le plus efficace sur des entrées de petite taille. Il est aussi très rapide lorsque les données sont déjà presque triées. Pour ces raisons, il est utilisé en pratique en combinaison avec d'autres méthodes.*"



```python
def insertionsort(liste):
    # remarque : on commence avec i=1 (et pas 0)
    for i in range(1, len(liste)):
            element = liste[i] # on mémorise l'élément
            # déplacer, un à un, les éléments de liste[0:i-1]
            # qui sont plus grands que "element"
            j = i-1
            while j >= 0 and element < liste[j]:
                liste[j + 1] = liste[j]
                j -= 1
            liste[j + 1] = element
    return(liste)
```

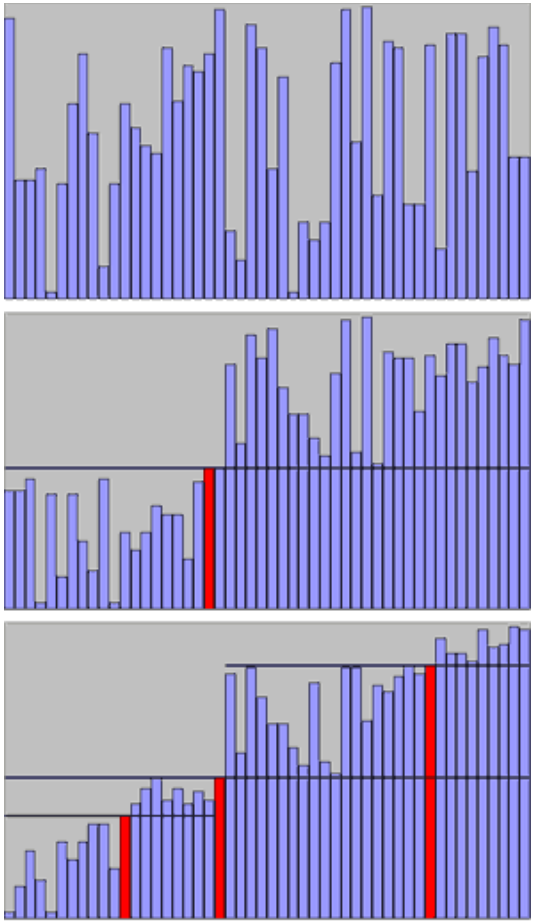
https://fr.wikipedia.org/wiki/Tri_par_insertion

https://en.wikipedia.org/wiki/Insertion_sort

https://www.youtube.com/watch?v=ROalU379l3U

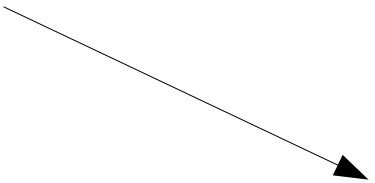
Quick sort – Tri rapide – diviser pour régner

- Partitionnement : Placer un élément de la liste (appelé *pivot*) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite.
- Pour chaque sous-liste, on définit un nouveau pivot et on répète le partitionnement.
- Répéter récursivement, jusqu'à ce que l'ensemble des éléments soit trié.



Une première fonction pour partitionner :

```
# on suppose que le pivot est l'élément liste[sup]
# modifie la liste dans l'intervalle liste[inf],...,liste[sup]
def partition(liste, inf, sup):
    i = inf-1  # indice du dernier élt. plus petit que le pivot
    pivot = liste[sup]
    for j in range(inf, sup):
        if liste[j] <= pivot:
           i = i+1
           liste[i],liste[j] = liste[j],liste[i]
    liste[i+1], liste[sup] = liste[sup], liste[i+1]
    return (i+1) # renvoie la position du pivot

l=[8,9,2,3,4,1,7,5]
print(partition(l,0,7))
print(l)

l=[8,9,2,3,4,1,7,5]
print(partition(l,0,4))
print(l)
```



```
4
[2, 3, 4, 1, 5, 9, 7, 8]
2
[2, 3, 4, 9, 8, 1, 7, 5]
```

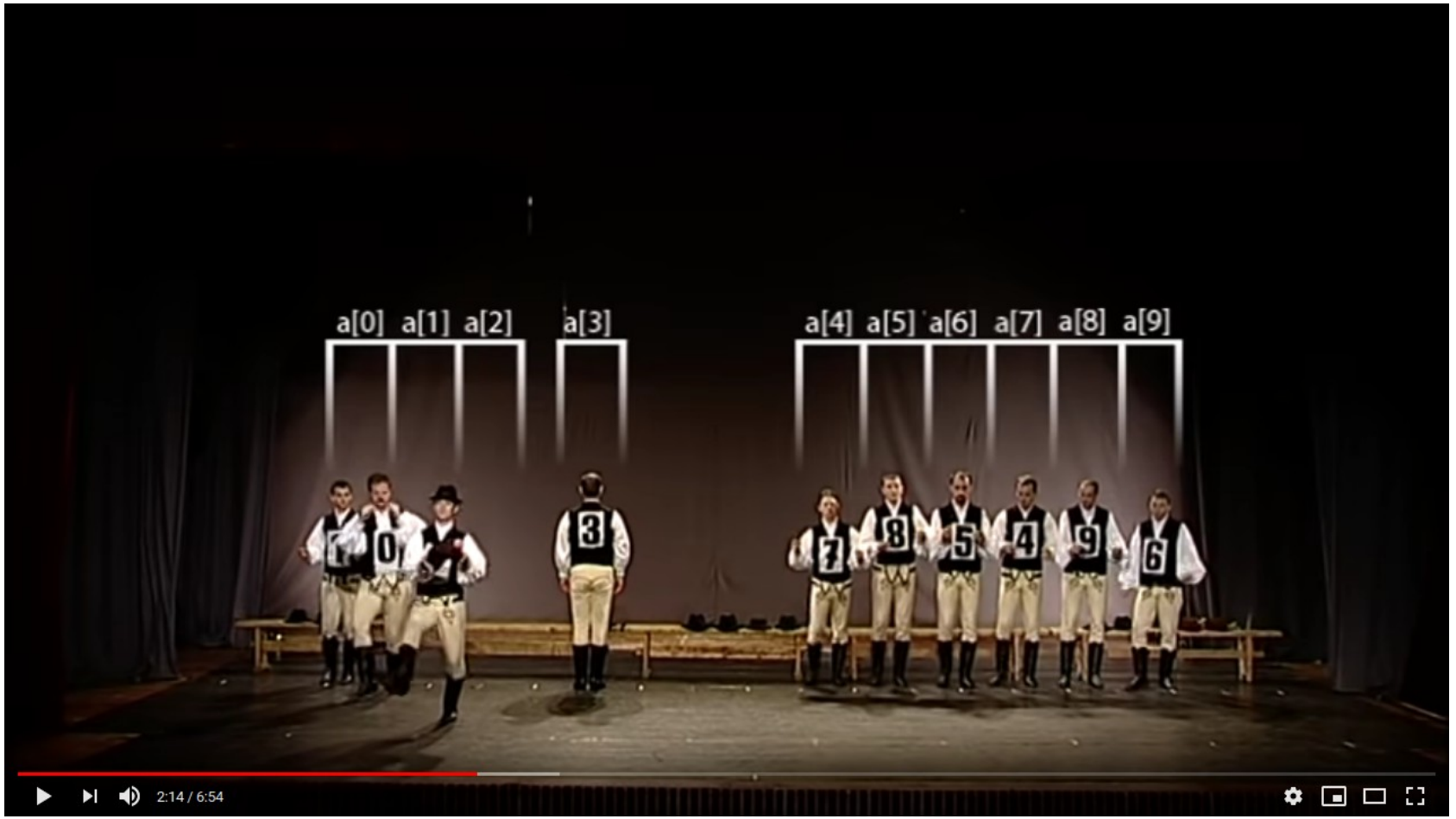
Et maintenant, le tri à proprement parler :

```
def quicksort(liste,inf,sup):
    if inf<sup:
        pi = partition(liste,inf,sup) # liste[pi] est bien positionné
        # on recommence récursivement
        quicksort(liste, inf, pi-1)
        quicksort(liste, pi+1, sup)

l=[8,9,2,3,4,1,7,5]
quicksort(l,0,len(l)-1)
print(l)
```

https://en.wikipedia.org/wiki/Quicksort

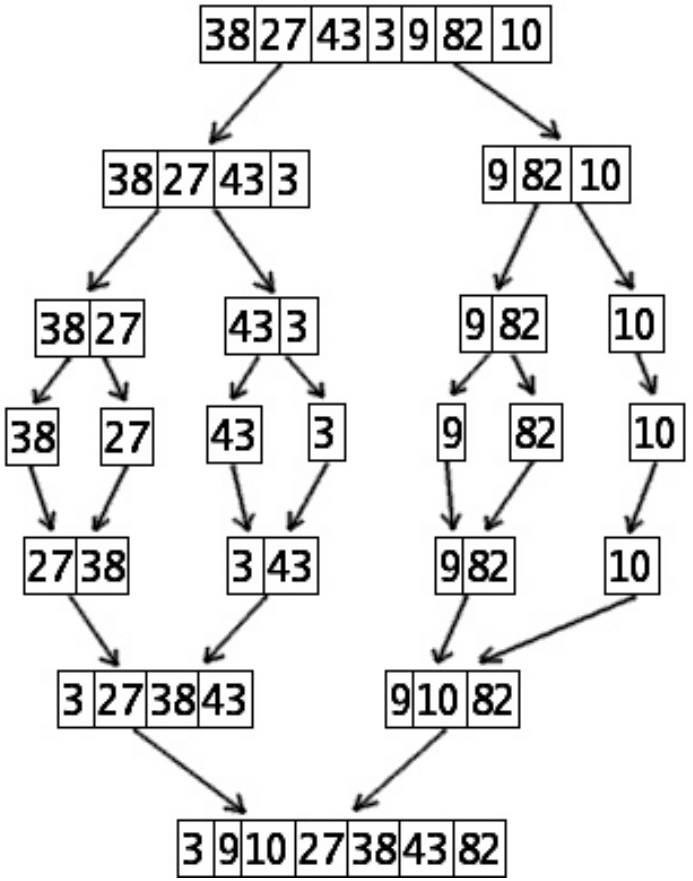
https://fr.wikipedia.org/wiki/Tri_rapide

https://www.youtube.com/watch?v=ywWBy6J5gz8



Merge sort – tri fusion – diviser pour régner

- Si la liste n'a qu'un élément, elle est triée.
- Sinon, séparer le tableau en deux parties “égales”.
- Trier récursivement les deux parties.
- Fusionner les deux listes trieés en une seule liste triée.

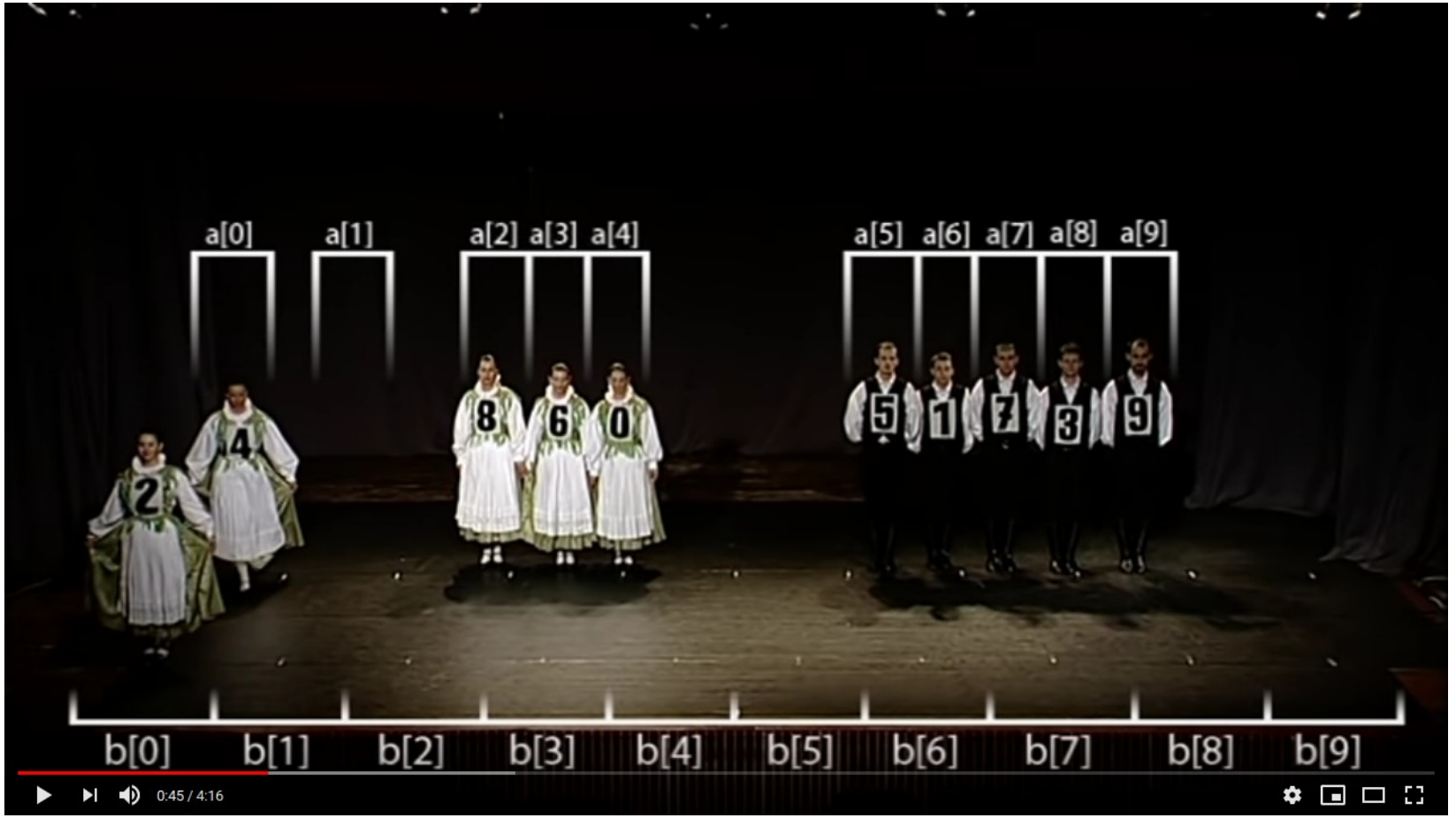


```python
def mergesort(liste):
    if len(liste) >1:
        milieu = len(liste)//2
        G = liste[:milieu]   # on travaille sur des copies
        D = liste[milieu:]   # on a coupé liste en 2
        mergesort(G)
        mergesort(D)

        # procéder à la fusion au sein de liste
        i = j = k = 0
        while i < len(G) and j < len(D):
            if G[i] < D[j]:
                liste[k] = G[i]
                i+=1
            else:
                liste[k] = D[j]
                j+=1
            k+=1

        # Quand on sort de la boucle, on n'a pas
        # nécessairement parcouru tout G ou L,
        while i < len(G):
            liste[k] = G[i]
            i+=1
            k+=1
        while j < len(D):
            liste[k] = D[j]
            j+=1
            k+=1
```

https://en.wikipedia.org/wiki/Merge_sort

https://fr.wikipedia.org/wiki/Tri_fusion

https://www.youtube.com/watch?v=XaqR3G_NVoo



Heap sort – tri par tas – voir la liste comme un arbre binaire

-

Le problème des 8 reines...

- Peut-on placer 8 reines sur un échiquier de telle sorte qu'aucune reine ne puisse en prendre une autre ?

https://www.youtube.com/watch?v=R8bM6pxlrLY