# Algorithmique et Programmation 2

## Recherche d'éléments et tri de liste

Licences Informatique et Mathématiques
1ère année

Semestre 2

# Rappel du cours précédent

Chercher un élément dans une liste non triée de taille $n$ peut se faire par *recherche exhaustive*

- Complexité $O(n)$
- Sans autre hypothèse on ne peut pas faire mieux

# Rappel du cours précédent

Chercher un élément dans une liste *triée* de taille $n$ peut se faire par *recherche binaire* (ou *dichotomie*)

- Complexité $O(\log n)$ (exponentiellement plus rapide !!)
- Sans autre hypothèse on ne peut pas faire mieux

# Rappel du cours précédent

Conclusion : si on a besoin de chercher fréquemment, on a intérêt à trier (ou à maintenir les données triées)

- Si ce n'est pas trop coûteux !
- D'où l'intérêt d'algorithmes *efficaces*

Section 1

# Intermède : nombre de chiffres d'un nombre

# Nombre de chiffres d'un nombre

Soit $n$ un entier positif écrit en base 10.
Combien de chiffres a-t-il ?

- Un nombre à $k$ chiffres est compris entre $10^{k-1}$ et $10^k - 1$
- Le nombre de chiffres de $n$ est donc le plus petit $k$ tel que $10^{k-1} \leq n \leq 10^k - 1$
- Comment calculer ce nombre ?

# Nombre de chiffres d'un nombre

Soit $n$ un entier positif écrit en base 10.
Combien de chiffres a-t-il ?

- Un nombre à $k$ chiffres est compris entre $10^{k-1}$ et $10^k - 1$
- Le nombre de chiffres de $n$ est donc le plus petit $k$ tel que $10^{k-1} \leq n \leq 10^k - 1$
- Comment calculer ce nombre ?
  - On essaie tous les $k$ un par un, ou bien...

# Nombre de chiffres d'un nombre

Soit $n$ un entier positif écrit en base 10.
Combien de chiffres a-t-il ?

- Un nombre à $k$ chiffres est compris entre $10^{k-1}$ et $10^k - 1$
- Le nombre de chiffres de $n$ est donc le plus petit $k$ tel que $10^{k-1} \leq n \leq 10^k - 1$
- Comment calculer ce nombre ?
  - On essaie tous les $k$ un par un, ou bien...
  - On utilise la fonction $\log_{10}$ (logarithme en base 10)

Proposition : Tout nombre entier positif $n$ en base 10 a exactement $\lfloor \log_{10}(n) \rfloor + 1$ chiffres

# Nombre de chiffres d'un nombre

Soit $n$ un entier positif écrit en base 10.
Combien de chiffres a-t-il ?

- Un nombre à $k$ chiffres est compris entre $10^{k-1}$ et $10^k - 1$
- Le nombre de chiffres de $n$ est donc le plus petit $k$ tel que $10^{k-1} \leq n \leq 10^k - 1$
- Comment calculer ce nombre ?
  - On essaie tous les $k$ un par un, ou bien...
  - On utilise la fonction $\log_{10}$ (logarithme en base 10)

Proposition : Tout nombre entier positif $n$ en base 10 a exactement $\lfloor \log_{10}(n) \rfloor + 1$ chiffres

Proposition (bis) : Tout nombre entier positif $n$ peut être divisé au plus $\lfloor \log_{10}(n) \rfloor + 1$ fois par $10$ avant d'obtenir $0$

# Conversion en binaire

**Exercice / rappel :** Écrire une fonction (itérative ou récursive) `binaire(n)` recevant un entier positif n et renvoyant son écriture en binaire sous la forme d'une liste de `0` et de `1`.
Quelle est la complexité de la fonction ?

# Conversion en binaire

## Binaire, version itérative

```
    def binaire(n):
        res = []
        while n != 0:
            n, r = n // 2, n % 2
            res.append(r)
        res.reverse()
        return res
```

# Conversion en binaire

## Binaire, version récursive

```
    def binaire(n):
        if n == 0:
            return []
        else:
            n, r = n // 2, n % 2
            res = binaire(n)
            res.append(r)
            return res
```

# Nombre de chiffres d'un nombre

Soit $n$ un entier positif écrit en base $b$.
Combien de chiffres a-t-il ?

- Un nombre à $k$ chiffres est compris entre $b^{k-1}$ et $b^k - 1$
- Le nombre de chiffres de $n$ est donc le plus petit $k$ tel que $b^{k-1} \leq n \leq b^k - 1$

Proposition : Tout nombre entier positif $n$ en base $b$ a exactement $\lfloor \log_b(n) \rfloor + 1$ chiffres

# Nombre de chiffres d'un nombre

Soit $n$ un entier positif écrit en base $b$.
Combien de chiffres a-t-il ?

- Un nombre à $k$ chiffres est compris entre $b^{k-1}$ et $b^k - 1$
- Le nombre de chiffres de $n$ est donc le plus petit $k$ tel que $b^{k-1} \leq n \leq b^k - 1$

Proposition : Tout nombre entier positif $n$ en base $b$ a exactement $\lfloor \log_b(n) \rfloor + 1$ chiffres

Proposition (bis) : Tout nombre entier positif $n$ peut être divisé au plus $\lfloor \log_b(n) \rfloor + 1$ fois par $b$ avant d'obtenir $0$

Section 2

# Tri de listes

# Motivation

- Comme on l'a vu avec la recherche, il est important d'organiser les données d'une certaine manière
- On va maintenant voir comment trier des listes

## Problème (tri)

**Données :** une liste `lst`.
**Objectif :** ordonner les éléments de `lst` de manière croissante.

Remarques :

1. On suppose que les éléments de `lst` sont comparables
2. On travaille ici sur des listes croissantes d'entiers, mais le raisonnement reste le même pour d'autres types

# Le problème du tri

- Le problème du tri peut être résolu par plusieurs algorithmes d'efficacités diverses
- On va voir trois algorithmes basiques permettant de résoudre ce problème :
  1. le tri à bulle
  2. le tri par sélection
  3. le tri par insertion
- Puis quelques algorithmes plus efficaces :
  1. le tri par pivot
  2. le tri par fusion

# Le tri à bulle

**Algorithme : tri à bulle de `lst`**

Pour chaque indice `i` de 0 à `len(lst)-1` :

- on parcourt les `n-i` derniers éléments depuis la fin
- on intervertit les éléments voisins mal ordonnés

Pourquoi ça marche :

- Après l'étape `i`, le plus petit élément parmi les `n-i` derniers remonte en position `i`
- Donc après l'étape `i`, les `i` premiers éléments de `lst` sont les plus petits et sont triés *(par récurrence)*
- Donc après la dernière étape la liste entière est triée

# Le tri à bulle

```
def tri_bulle(lst):
    n = len(lst)
    # on fait croître la portion triée
    for i in range(0, n-1):
        # on fait remonter la bulle
        # dans la portion non triée
        for j in range(n-1, i, -1):
            if lst[j-1] > lst[j]:
                # échange de voisins mal ordonnés
                lst[j-1], lst[j] = lst[j], lst[j-1]
```

# Complexité du tri à bulle

- Pour chaque valeur de `i` entre 0 et n-2 :
  - `n-i-1` comparaisons dans tous les cas
  - `n-i-1` échanges au pire (0 au mieux)
- Nombre total de comparaisons dans tous les cas :

$$\sum_{i=0}^{n-2} n - i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$

- Nombre total d'échanges : 0 au mieux, $O(n^2)$ au pire
- Pire cas : liste décroissante, meilleur cas : liste croissante
- Amélioration possible : arrêter les comparaisons à la dernière position d'échange (Cf. exercices)

# Le tri par sélection

Idée : on peut aussi échanger des éléments non adjacents !

**Algorithme : tri par sélection**

Pour chaque `i` entre 0 et `len(lst)-2` :

1. on cherche le plus petit élément à partir de l'indice `i`
2. on échange `lst[i]` avec cet élément

Pourquoi ça marche :

- Après l'étape `i`, on a dans les `i` premières positions de `lst` les `i` plus petits éléments de `lst` dans l'ordre
- Après la dernière étape, la liste est bien triée

# Le tri par sélection

```
def indice_min(lst, i):
    position = i
    minimum = lst[i]
    for p in range(i + 1, len(lst)):
        if lst[p] < minimum:
            minimum = lst[p]
            position = p
    return position
```

```
def tri_selection(lst):
    for i in range(len(lst)-1):
        p = indice_min(lst, i)
        lst[i], lst[p] = lst[p], lst[i]
```

# Complexité du tri par sélection

- Pour chaque valeur de `i` entre 0 et n-2 :
  - `n-i-1` comparaisons dans `indice_min`
  - 1 échange dans tous les cas
- Nombre total de comparaisons dans tous les cas :

$$\sum_{i=0}^{n-2} n - i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$

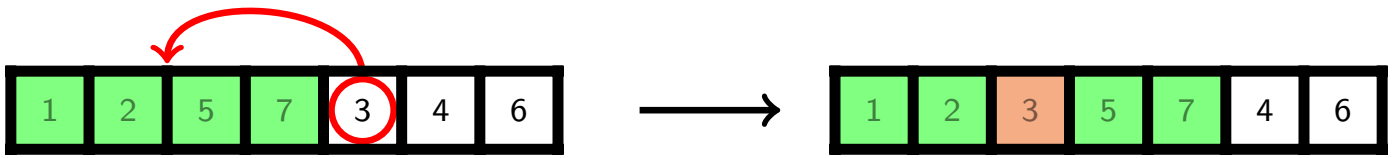
- Nombre total d'échanges : $n - 1$ dans tous les cas
- Pas de pire ni de meilleur cas
  (complexité indépendante du contenu de la liste)

# Le tri par insertion

- Au lieu d'échanger deux éléments, on pourrait tout simplement déplacer ceux qui sont "mal placés"
- Pour chaque indice `i` entre 1 et `len(lst)-1`, on insère `lst[i]` "à la bonne place" parmi les `i` premiers éléments
- Après l'étape `i`, les `i+1` premiers éléments sont triés
- Lorsque l'algorithme se termine, la liste est bien triée

# Insertion d'un élément

- Pour mettre en œuvre le tri par insertion, il faut donc savoir comment effectuer les réinsertions




- On peut procéder en trois étapes :
  1. sauvegarder l'élément e à déplacer
  2. décaler les éléments d'une position de manière à créer une place libre à la destination
  3. affecter la valeur de e à la destination

# Insertion d'un élément

```
def insertion(lst, i):
    # sauvegarder l'élément à déplacer
    e = lst[i]
    # décaler les éléments plus grands
    k = i
    while k > 0 and lst[k-1] > e:
        lst[k] = lst[k-1]
        k = k - 1
    # insérer e en position k
    lst[k] = e
```

```
def tri_insertion(lst):
    for i in range(1, len(lst)):
        insertion(lst, i)
```

# Complexité du tri par insertion

- Pour chaque valeur de `i` entre 1 et n−1 :
  - Entre 1 et `i`−1 comparaisons
  - Entre 0 et `i` affectations
- Nombre de comparaisons au pire (liste décroissante) :

$$\sum_{i=0}^{n-2} n - i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$

- Nombre d'affectations au pire : $O(n^2)$ (calcul similaire)
- Meilleur cas : $O(n)$ sur liste croissante (aucun décalage)
- Amélioration possible : chercher la position finale de chaque élément par dichotomie sur le début de la liste (ne change pas le nombre d'affectations nécessaire)

# Autres algorithmes de tris

- Les algorithmes de tri déjà présentés ont une complexité prohibitive
- On va voir maintenant des algorithmes plus performants
  1. le tri rapide
  2. le tri fusion

# Echauffement : le tri pair / impair

- A titre d'échauffement, essayons de résoudre le problème suivant :

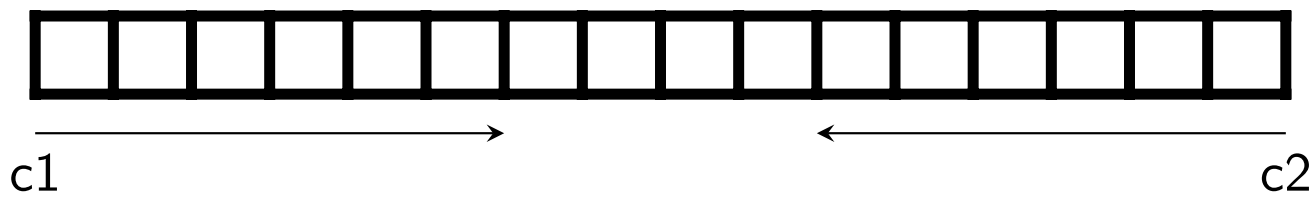
## Problème (tri pair / impair)

**Données :** une liste `lst` de naturels
**Objectif :** mettre les éléments pairs de `lst` au début et les impairs à la fin

- Comment résoudre ce problème de manière efficace ?

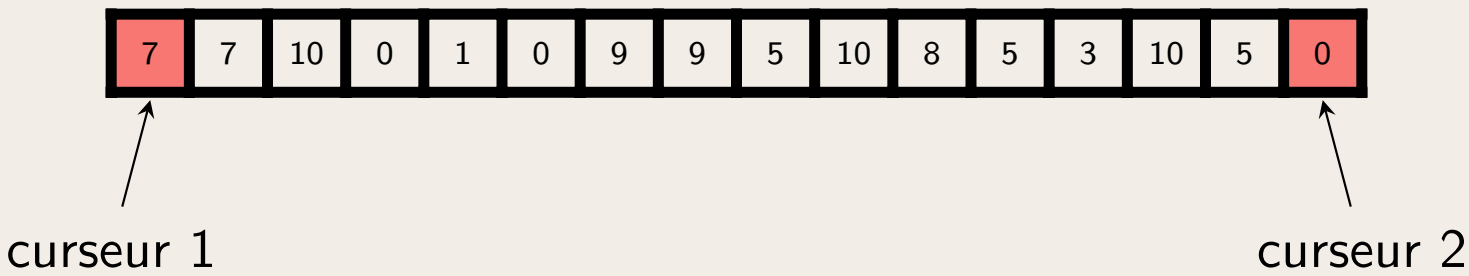
# Echauffement : le tri pair / impair

- Une solution possible : parcourir la liste simultanément avec deux curseurs c1 et c2
- Les deux curseurs progressent l'un vers l'autre au départ des extrémités
- A chaque fois qu'on tombe sur un couple (`lst[c1]`, `lst[c2]`) d'éléments mal placés, on les échange



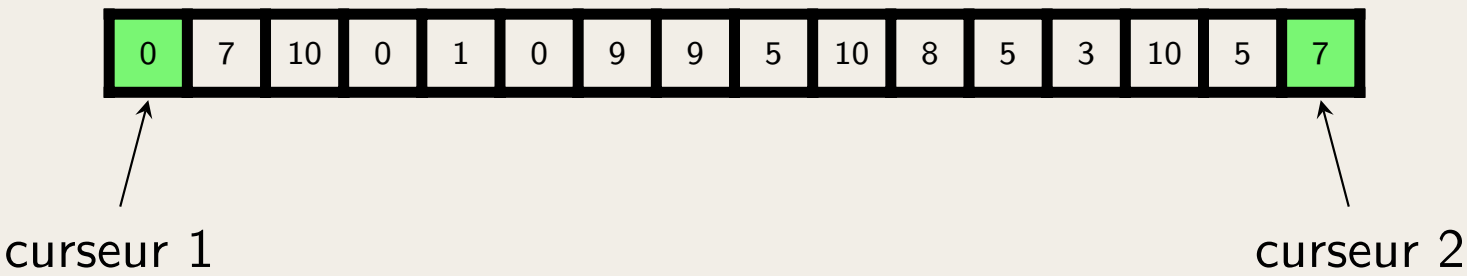
# Le tri pair / impair en action

## Exemple



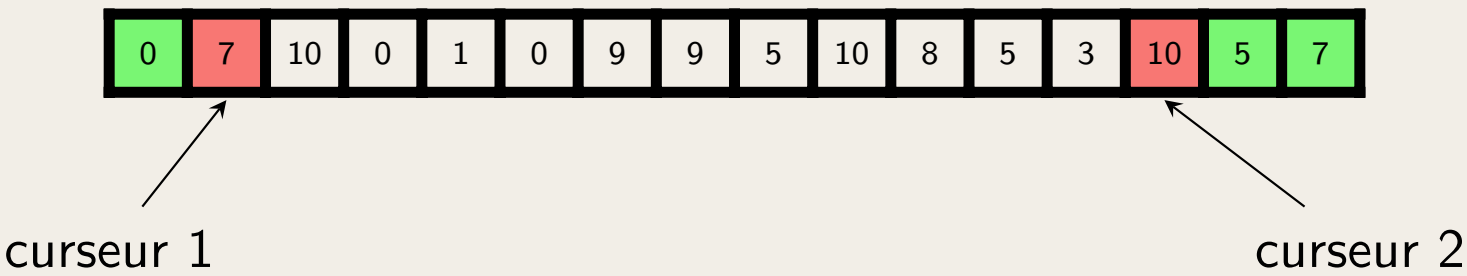
# Le tri pair / impair en action

## Exemple



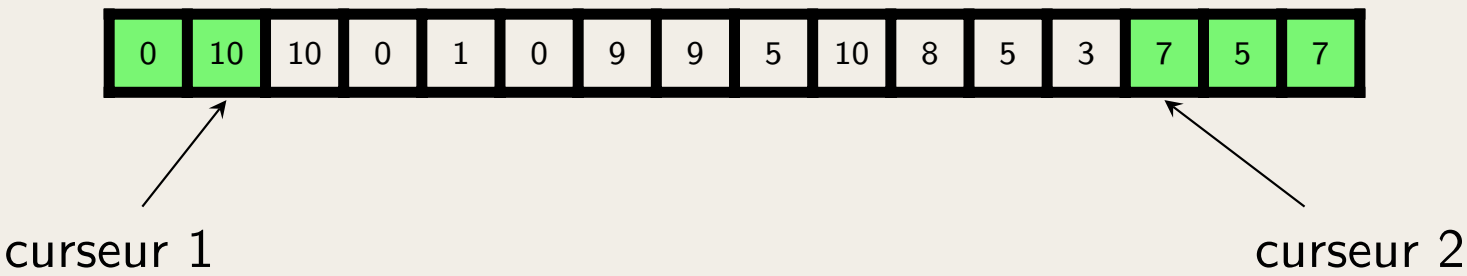
# Le tri pair / impair en action

## Exemple



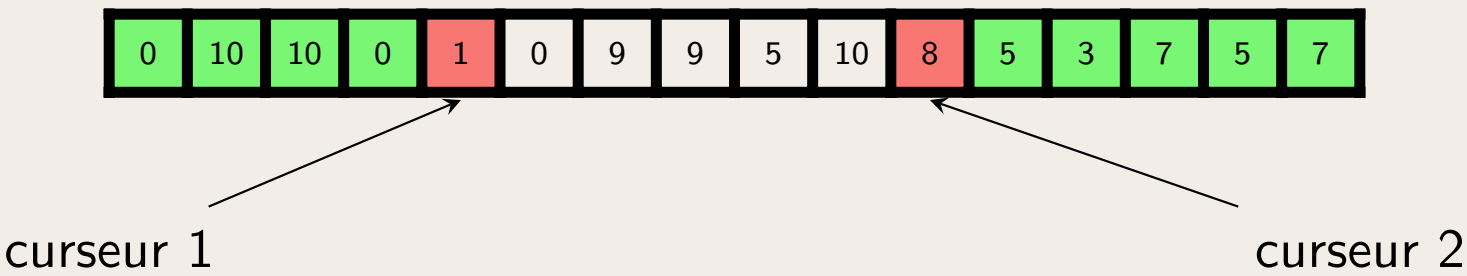
# Le tri pair / impair en action

## Exemple



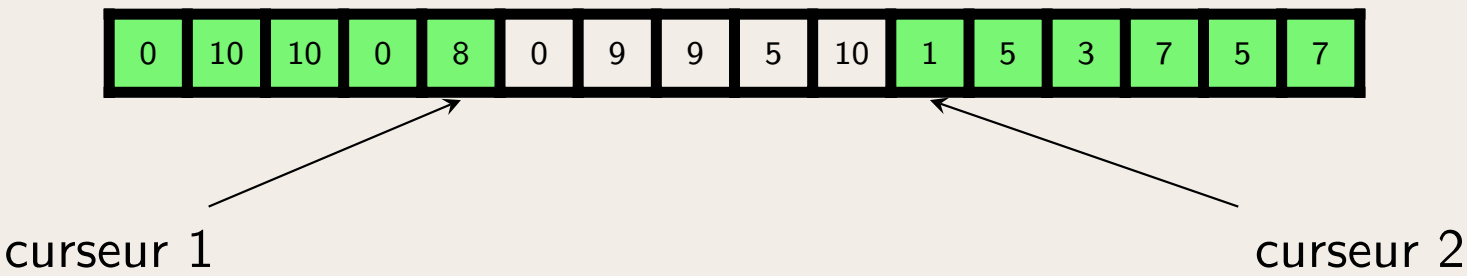
# Le tri pair / impair en action

## Exemple



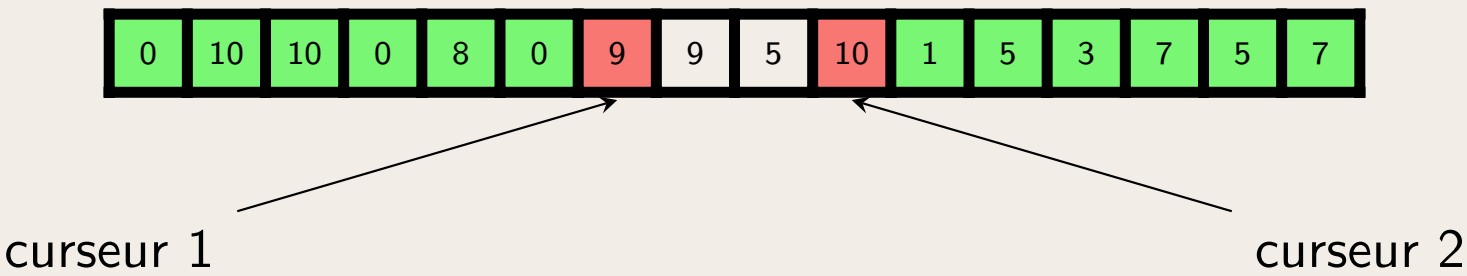
# Le tri pair / impair en action

## Exemple



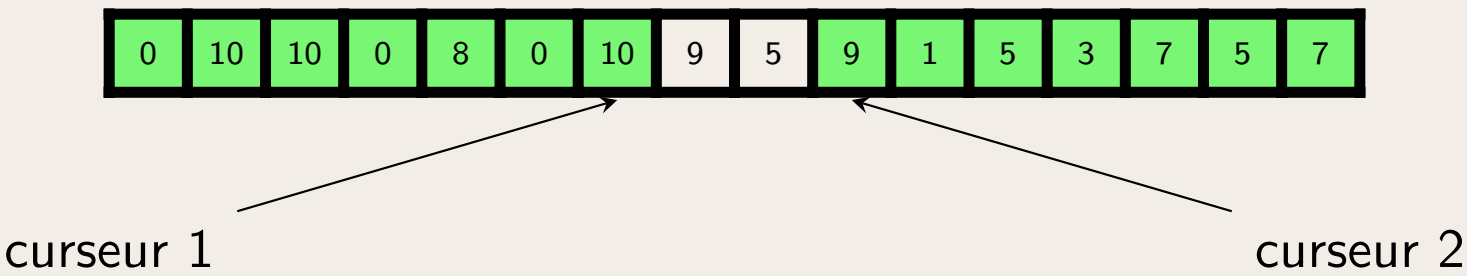
# Le tri pair / impair en action

## Exemple



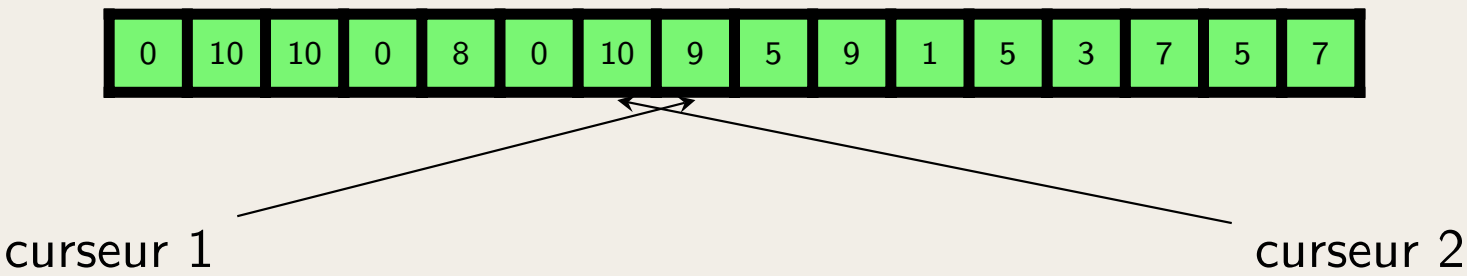
# Le tri pair / impair en action

## Exemple



# Le tri pair / impair en action

## Exemple




- L'algorithme s'arrête quand les curseurs se croisent

# Le tri pair / impair en Python

```
def pair_impair(lst):
    curseur1, curseur2 = 0, len(lst) - 1
    while curseur1 <= curseur2:
        # repérer le premier élément impair en partant du début
        while curseur1 <= curseur2 and lst[curseur1] % 2 == 0:
            curseur1 += 1

        # repérer le premier élément pair en partant de la fin
        while curseur1 <= curseur2 and lst[curseur2] % 2 != 0:
            curseur2 -= 1

        # si nécessaire, procéder à l'échange
        if curseur1 < curseur2:
            lst[curseur1], lst[curseur2] = lst[curseur2], lst[curseur1]
            curseur1 += 1
            curseur2 -= 1
```

- Complexité ?

# Le tri pair / impair en Python

```
def pair_impair(lst):
    curseur1, curseur2 = 0, len(lst) - 1
    while curseur1 <= curseur2:
        # repérer le premier élément impair en partant du début
        while curseur1 <= curseur2 and lst[curseur1] % 2 == 0:
            curseur1 += 1

        # repérer le premier élément pair en partant de la fin
        while curseur1 <= curseur2 and lst[curseur2] % 2 != 0:
            curseur2 -= 1

        # si nécessaire, procéder à l'échange
        if curseur1 < curseur2:
            lst[curseur1], lst[curseur2] = lst[curseur2], lst[curseur1]
            curseur1 += 1
            curseur2 -= 1
```

- Complexité ? Temps $O(n)$, espace $O(1)$

# Généralisation

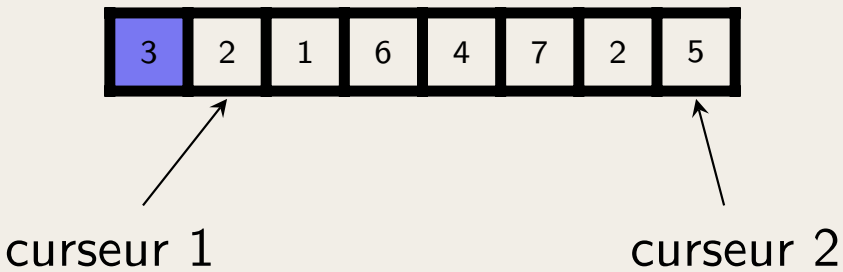
- On peut utiliser ce principe pour trier la liste
- Au lieu de se contenter de mettre les pairs au début et les impairs à la fin, on peut mettre les éléments inférieurs à une certaine valeur avant elle et les autres après elle
- Ce principe est à la base du tri rapide

# Le tri rapide

- Principe du tri rapide :
  1. Sélectionner un pivot, c'est-à-dire une valeur p de référence dans la liste (par exemple la première)
  2. Partitionner la liste :
     - Toutes les valeurs inférieures à p se retrouvent avant p
     - Toutes les valeurs supérieures à p se retrouvent après p
  3. Recommencer récursivement sur les deux parties de liste (avant et après le pivot)

# Le tri rapide

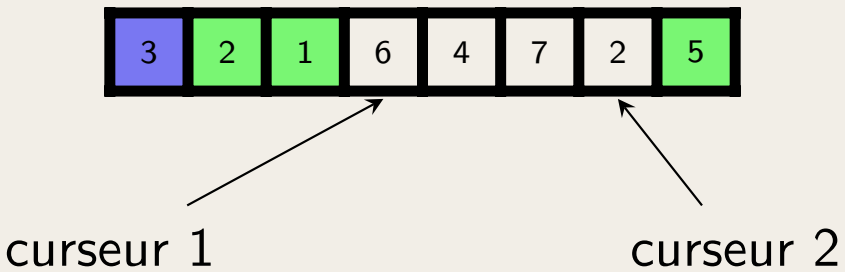
## Exemple




1. On partitionne avec le premier élément comme pivot

# Le tri rapide

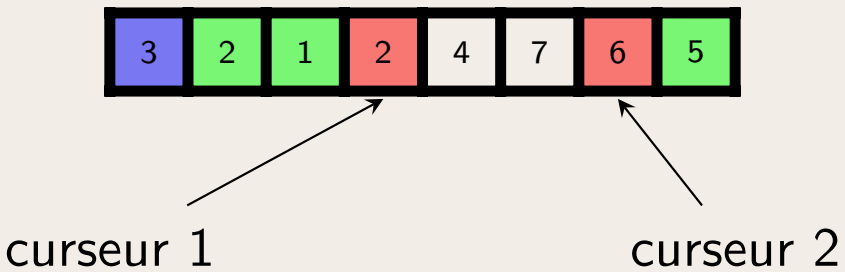
## Exemple




1. On partitionne avec le premier élément comme pivot

# Le tri rapide

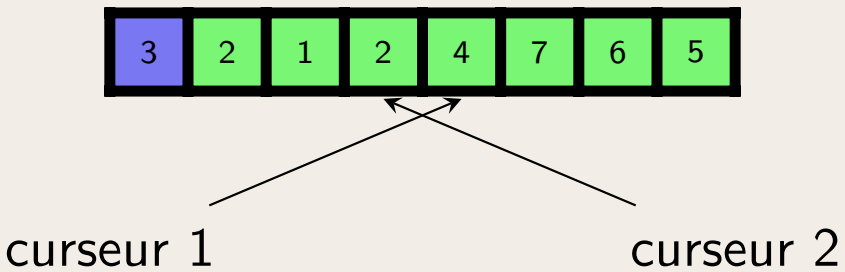
## Exemple




1. On partitionne avec le premier élément comme pivot

# Le tri rapide

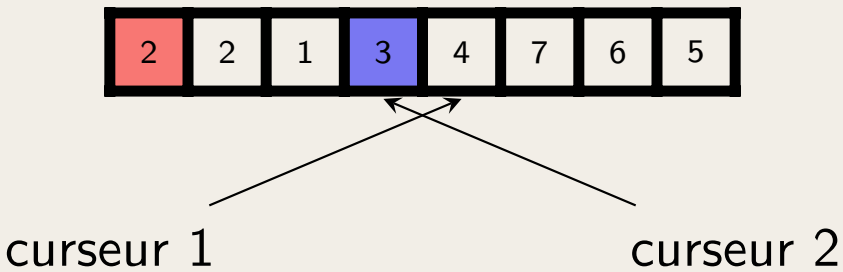
## Exemple




1. On partitionne avec le premier élément comme pivot

# Le tri rapide

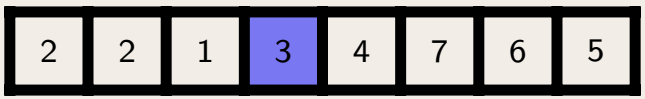
## Exemple




1. On partitionne avec le premier élément comme pivot
2. On place le pivot à la “frontière”

# Le tri rapide

## Exemple




1. On partitionne avec le premier élément comme pivot
2. On place le pivot à la “frontière”
3. On trie les deux zones récursivement par le même principe

# Calcul de la partition

- L'algorithme est très similaire au tri pair / impair ;
  - avant : pairs à gauche, impairs à droite ;
  - maintenant : $x \leq$ p à gauche, $y >$ p à droite ;

```
def partition(lst, debut, fin):
    curseur1, curseur2 = debut + 1, fin

    while curseur1 <= curseur2:
        while curseur1 <= curseur2 and lst[curseur1] <= lst[debut]:
            curseur1 += 1

        while lst[curseur2] > lst[debut]:
            curseur2 -= 1

        if curseur1 < curseur2:  # si nécessaire, procéder à l'échange
            lst[curseur1], lst[curseur2] = lst[curseur2], lst[curseur1]
            curseur1 += 1
            curseur2 -= 1

    lst[debut], lst[curseur2] = lst[curseur2], lst[debut]
    return curseur2  # la position finale du pivot servira plus tard
```

# Le tri rapide en Python

```
def tri_rapide(lst, debut=0, fin=None):
    if fin is None:
        fin = len(lst) - 1

    if debut < fin:
        # partition entre debut et fin
        pivot = partition(lst, debut, fin)

        # tri de la zone entre debut et pivot
        tri_rapide(lst, debut, pivot - 1)

        # tri de la zone entre pivot et fin
        tri_rapide(lst, pivot + 1, fin)
```

# Complexité du tri rapide

- Complexité du partitionnement : $O(n)$
- Pire cas : élément minimal ou maximal comme pivot
  - Une partie à 0 éléments, l'autre à $n-1$ éléments
  - Mêmes calculs que pour les tris naïfs : $O(n^2)$
- Cas le plus favorable : partition en deux moitiés égales
  - Nombre d'appels récursifs en $O(\log n)$
  - Au $k^e$ niveau de récursion on partitionne environ $2^k$ listes de taille environ $n/2^k$, coût total $O(n)$
  - Coût total : $O(n \log n)$
- En moyenne : on peut montrer qu'on obtient $O(n \log n)$

# Tri par pivot

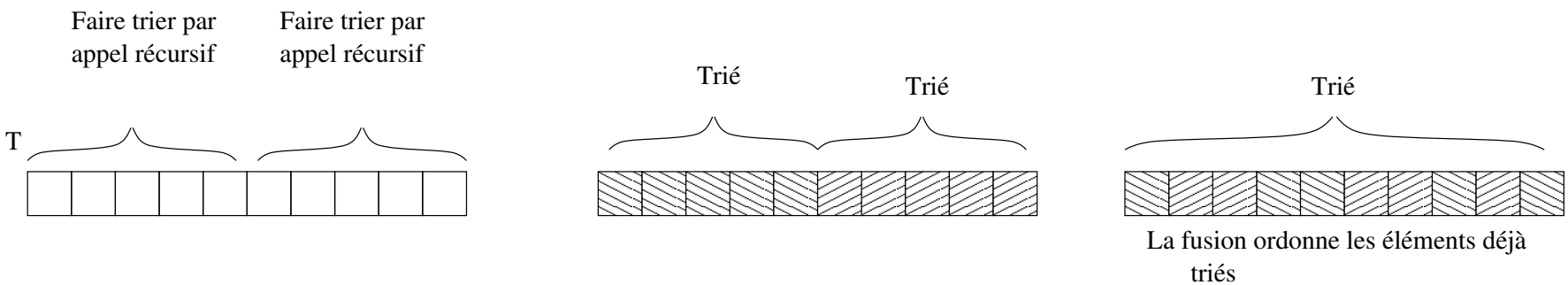
- Pire cas du tri par pivot : liste déjà triée
- Problème : en pratique cas très fréquent !
- (au moins) 2 solutions :
  - Mélanger la liste avant de commencer (`random.shuffle`, détails à suivre)
  - Choisir un pivot au hasard avant de partitionner

# Le tri rapide en Python

```
def tri_rapide(lst, debut=0, fin=None):
    if fin is None:
        fin = len(lst) - 1

    if debut < fin:
        # choix d'un pivot aléatoire et placement en debut
        pivot = random.randint(debut, fin)
        lst[debut], lst[pivot] = lst[pivot], lst[debut]

        # partition entre debut et fin
        pivot = partition(lst, debut, fin)

        # tri de la zone entre debut et pivot
        tri_rapide(lst, debut, pivot - 1)

        # tri de la zone entre pivot et fin
        tri_rapide(lst, pivot + 1, fin)
```

# Le tri fusion

- Ce tri se décrit facilement récursivement :
  - Si la liste contient 0 ou 1 élément, elle est triée
  - Sinon :
    1. On la divise en deux listes de tailles égales
    2. On trie les deux sous-listes
    3. On fusionne le résultat à l'aide d'une liste auxiliaire
    4. On recopie la liste auxiliaire dans la liste d'origine



# La partie fusion

- Algorithme de fusion de listes triées :
  - On utilise deux curseurs parcourant `lst1` et `lst2` en parallèle dans le même sens
  - À chaque étape, on ajoute `min(lst1[c1], lst2[c2])` au résultat
  - Si une des listes est épuisée on recopie la fin de l'autre
- Ici, on va travailler sur deux portions voisines d'une même liste `lst`
  - Il est suffisant de travailler avec des indices
  - Une fois le résultat obtenu, on le recopie sur la portion correspondante de `lst`

# La partie “fusion”

```
def fusionner(lst, debut, milieu, fin):
    aux = []
    curseur1, curseur2 = debut, milieu + 1
    for k in range(debut, fin + 1):
        # si une des deux sous-listes a été copiée, copier l'autre
        if curseur1 > milieu:
            aux.append(lst[curseur2])
            curseur2 += 1
        elif curseur2 > fin:
            aux.append(lst[curseur1])
            curseur1 += 1
        # sinon, copier min(lst[curseur1], lst[curseur2])
        elif lst[curseur1] < lst[curseur2]:
            aux.append(lst[curseur1])
            curseur1 += 1
        else:
            aux.append(lst[curseur2])
            curseur2 += 1
    # on recopie la liste auxiliaire sur la liste à trier
    lst[debut:fin+1] = aux
```

# La partie “tri”

```
def tri_fusion(lst, debut, fin):
    # plus que 0 ou 1 élément à trier
    if debut >= fin:
        return

    # partitionner en deux-sous listes
    milieu = debut + (fin - debut) // 2

    # les trier
    tri_fusion(lst, debut, milieu)
    tri_fusion(lst, milieu+1, fin)

    # les fusionner
    fusionner(lst, debut, milieu, fin)
```

# Complexité

- Complexité de la fusion : $O(n)$
- Même idée que pour le cas favorable du tri par pivot
- Dans tous les cas : partition en deux moitiés égales
  - Profondeur max d'appels récursifs imbriqués : $O(\log n)$
  - Au $k^e$ niveau de récursion on fusionne $2^k$ listes de taille environ $n/2^k$, coût total $O(n)$
  - Coût total : $O(n \log n)$
- Algorithme (asymptotiquement) optimal mais plus de mémoire utilisée que tri par pivot
- Tri stable (des éléments égaux restent dans le même ordre)

# Le tri de Python

Méthode `sort` : tri sur place de `list`

- Algo hybride entre tri par fusion et tri par insertion (*Timsort*)
- Stable, $O(n \log n)$ en temps (au pire et en moyenne), $O(n)$ en espace
- Utilisé par d'autres langages de programmation (par ex. Java)
- **Attention :** trie sur place, ne renvoie pas de liste !
- Possibilité de trier selon un critère donné (`key=f`), ou à l'envers (`reverse=True`)

Tri sans modification de la liste : fonction `sorted`

# Le tri de Python

```
>>> lst = ["Chennai", "Mumbai", "Kochi", "Delhi", "Calcutta", "Amritsar"]
>>> lst.sort()
>>> print(lst)
['Amritsar', 'Calcutta', 'Chennai', 'Delhi', 'Kochi', 'Mumbai']

>>> lst.sort(reverse=True)
>>> print(lst)
['Mumbai', 'Kochi', 'Delhi', 'Chennai', 'Calcutta', 'Amritsar']

>>> lst.sort(key=len)
>>> print(lst)
['Kochi', 'Delhi', 'Mumbai', 'Chennai', 'Calcutta', 'Amritsar']

>>> def derniere_lettre(s):
...     return s[-1]
...
>>> sorted(lst, key=derniere_lettre)
['Calcutta', 'Kochi', 'Delhi', 'Mumbai', 'Chennai', 'Amritsar']
```

Section 3

# Variations autour du tri

# Mélange de liste

On a vu comment trier une liste, mais comment la mélanger uniformément ?

Première tentative :

```
from random import randrange

def melange(lst):
    for i in range(len(lst)):
        k = randrange(len(lst))
        lst[i], lst[k] = lst[k], lst[i]
```

Exercice : Quelle est la fréquence d'apparition de chaque résultat possible sur une liste de longueur 3 ?

# Mélange de liste

```
from random import randrange
def melange(lst):
    for i in range(len(lst)):
        k = randrange(len(lst))
        lst[i], lst[k] = lst[k], lst[i]
```

Buggé !!!

- Combien de permutations possibles ? $n!$
- Combien d'exécutions possibles de l'algorithme ? $n^n$ (beaucoup plus)
- $n^n$ n'est pas divisible par $n!$ en général $\rightarrow$ répartition égale impossible

# Mélange de liste

Seconde tentative

```
def melange(lst):
    for i in range(len(lst)-1):
        k = randrange(i, len(lst))  # changement ici !
        lst[i], lst[k] = lst[k], lst[i]
```

On peut raisonner à l'envers pour comprendre :

- À chaque étape on "devine" l'élément qui était en position `i` avant de trier
- Les éléments de rang $< i$ sont déjà fixés, donc on le nes considère pas
- Une fois chaque élément fixé, on a fini
- Complexité :

# Mélange de liste

Seconde tentative

```
def melange(lst):
    for i in range(len(lst)-1):
        k = randrange(i, len(lst))  # changement ici !
        lst[i], lst[k] = lst[k], lst[i]
```

On peut raisonner à l'envers pour comprendre :

- À chaque étape on "devine" l'élément qui était en position `i` avant de trier
- Les éléments de rang $<$ `i` sont déjà fixés, donc on le nes considère pas
- Une fois chaque élément fixé, on a fini
- Complexité : $O(n)$

# Recherche de la médiane

**Problème de recherche de la médiane**

Donnée : une liste de $n$ éléments comparables
Résultat : l'élément médian de la liste

(Médiane : autant d'éléments supérieurs que d'éléments inférieurs)

- Algorithme naïf : calculer $\lfloor n/2 \rfloor$ fois le plus petit élément parmi les éléments restants
- Complexité :

# Recherche de la médiane

**Problème de recherche de la médiane**

Donnée : une liste de $n$ éléments comparables
Résultat : l'élément médian de la liste

(Médiane : autant d'éléments supérieurs que d'éléments inférieurs)

- Algorithme naïf : calculer $\lfloor n/2 \rfloor$ fois le plus petit élément parmi les éléments restants
- Complexité : $O(n^2)$. Peut-on faire mieux ?

# Recherche de la médiane

**Problème de recherche de la médiane**

Donnée : une liste de $n$ éléments comparables
Résultat : l'élément médian de la liste

(Médiane : autant d'éléments supérieurs que d'éléments inférieurs)

- Algorithme naïf : calculer $\lfloor n/2 \rfloor$ fois le plus petit élément parmi les éléments restants
- Complexité : $O(n^2)$. Peut-on faire mieux ?
- Idée : si on partitionne selon un pivot, l'élément médian est dans la partie qui contient plus de $n/2$ éléments...

Exercice : écrire une fonction efficace de recherche de la médiane

# Recherche de la médiane

**Problème de recherche du $k$è plus petit**

Donnée : une liste de $n$ éléments comparables
Résultat : le $k$è plus petit élément de la liste

```
def plus_petit(lst, k):
    debut = 0
    fin = len(lst)-1
    while True:
        p = randint(debut, fin)
        p = partitionne(lst, debut, p, fin)
        if p == k:
            return lst[k]
        elif p < k:
            debut = p+1
        else:
            fin = p-1
```

Complexité :

# Recherche de la médiane

## Problème de recherche du $k$è plus petit

Donnée : une liste de $n$ éléments comparables
Résultat : le $k$è plus petit élément de la liste

```
def plus_petit(lst, k):
    debut = 0
    fin = len(lst)-1
    while True:
        p = randint(debut, fin)
        p = partitionne(lst, debut, p, fin)
        if p == k:
            return lst[k]
        elif p < k:
            debut = p+1
        else:
            fin = p-1
```

Complexité : $O(n)$ en temps et $O(1)$ en espace (en moyenne !)