

Algorithmique et Programmation 2

Recherche d'éléments et tri de liste

Licences Informatique et Mathématiques
1ère année

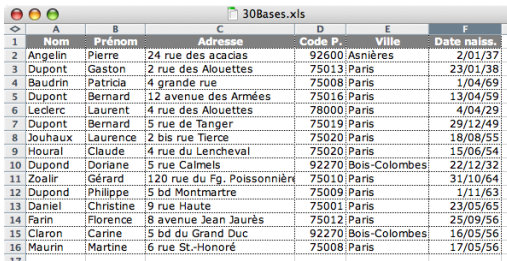
Semestre 2

../../../latex/UPEM-IGM-V1_300dp

Introduction

Certains calculs sur les listes sont très fréquents :

- ▶ la recherche d'informations
- ▶ le tri des informations



	A	B	C	D	E	F
1	Nom	Prénom	Adresse	Code P.	Ville	Date naiss.
2	Angelin	Pierre	24 rue des acacias	92600	Asnières	2/01/37
3	Dupont	Gaston	2 rue des Alouettes	75013	Paris	23/01/38
4	Baudrin	Patricia	4 grande rue	75008	Paris	1/04/69
5	Dupont	Bernard	12 avenue des Armées	75016	Paris	13/04/59
6	Leclerc	Laurent	4 rue des Alouettes	78000	Paris	4/04/29
7	Dupont	Bernard	5 rue de Tanger	75019	Paris	29/12/49
8	Jouhaux	Laurence	2 bis rue Tierce	75020	Paris	18/08/55
9	Houral	Claude	4 rue du Lencheval	75020	Paris	15/06/54
10	Dupond	Doriane	5 rue Calmels	92270	Bois-Colombes	22/12/32
11	Zoalir	Gérard	120 rue du Fg. Poissonnière	75010	Paris	31/10/64
12	Dupond	Philippe	5 bd Montmartre	75009	Paris	1/11/63
13	Daniel	Christine	9 rue Haute	75001	Paris	23/05/65
14	Farin	Florence	8 avenue Jean Jaurès	75012	Paris	25/09/56
15	Claron	Carine	5 bd du Grand Duc	92270	Bois-Colombes	16/05/56
16	Maurin	Martine	6 rue St.-Honoré	75008	Paris	17/05/56

Ces opérations sont pré-définies en Python :

- ▶ `e in lst` ou `i = lst.index(e)`
- ▶ `sorted(lst)` ou `lst.sort()`

Comment sont-elles implémentées ?

Section 1

Recherche d'élément

Recherche d'un élément

Problème (recherche)

Données : une liste `lst`, un élément `x`

Question : est-ce que `lst` contient `x` ?

Recherche d'un élément

Problème (recherche)

Données : une liste `lst`, un élément `x`

Question : est-ce que `lst` contient `x` ?

Algorithme simple : recherche exhaustive

- ▶ Pour chaque élément `elem` de `lst`
 - ▶ Si `elem` vaut `x`, répondre `True`
- ▶ À la fin du parcours, répondre `False`

Recherche exhaustive (itérative)

- ▶ Pour chaque élément successif elem de lst
 - ▶ Si elem vaut x, répondre **True**
- ▶ À la fin du parcours, répondre **False**

```
def rech_exh_iter(lst, x):  
    for elem in lst:  
        if elem == x:  
            return True  
    return False
```

Recherche exhaustive (itérative)

- ▶ Pour chaque élément `elem` de `lst`
 - ▶ Si `elem` vaut `x`, répondre `True`
- ▶ À la fin du parcours, répondre `False`

Complexité :

- ▶ 1 comparaison au mieux, `len(lst)` au pire
Total : $O(\text{len}(lst))$ même avec opérations cachées
- ▶ Coût supplémentaire en espace : $O(1)$

Recherche exhaustive (récursive)

Version récursive :

- ▶ S'il n'y a plus d'élément non vu, répondre **False**
- ▶ Sinon si le premier élément non vu vaut x répondre **True**
- ▶ Sinon, rechercher x parmi les éléments suivants

Recherche exhaustive (récursive)

Version récursive :

- ▶ S'il n'y a plus d'élément non vu, répondre **False**
- ▶ Sinon si le premier élément non vu vaut x répondre **True**
- ▶ Sinon, rechercher x parmi les éléments suivants

```
def rech_exh_rec(lst, x, i=0):  
    if i >= len(lst):  
        return False  
    if lst[i] == x:  
        return True  
    return rech_exh_rec(lst, x, i+1)
```

Recherche exhaustive (récursive)

Version récursive (d : indice de début de recherche) :

- ▶ S'il n'y a plus d'élément non vu, répondre **False**
- ▶ Sinon si le premier élément non vu vaut x répondre **True**
- ▶ Sinon, rechercher x parmi les éléments suivants

Complexité :

- ▶ Coût en opérations : $O(\text{len}(\text{lst}))$
- ▶ **Attention** : coût en espace $O(\text{len}(\text{lst}))$!

Recherche exhaustive : discussion

Peut-on faire mieux (moins d'accès à la liste) ?

- ▶ Soit f une recherche en moins de $\text{len}(\text{lst})-1$ accès
- ▶ Soit lst une liste quelconque d'entiers positifs
- ▶ $f(\text{lst}, -1)$ renvoie forcément **False**
- ▶ Soit i l'indice d'un élément non vu
- ▶ Si l'on avait eu -1 dans $\text{lst}[i]$ on aurait aussi eu **False**
→ fonction **incorrecte** !

Conclusion : si on ne sait rien sur la liste, on ne peut pas faire mieux que $\text{len}(\text{lst})$ comparaisons !

Recherche sur une liste triée

Et sur une liste triée (par ex. croissante) ?

Échauffement : je pense à un nombre entre 1 et 1000000.
Devinez-le.

- ▶ Vous avez le droit de proposer n'importe quel nombre
- ▶ Je réponds “trouvé”, “plus petit” ou “plus grand”

En combien de questions êtes-vous sûrs de gagner ?

Recherche sur une liste triée

Et sur une liste triée (par ex. croissante) ? Procéder par **dichotomie**

- ▶ Si on voit un élément trop petit, inutile de chercher avant
- ▶ Si on voit un élément trop grand, inutile de chercher après
- ▶ Pour éliminer le plus de cas possible on regarde toujours au milieu de l'intervalle

Analogies :

- ▶ Jeu de la devinette
- ▶ Recherche d'un nom dans l'annuaire

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

g																			d
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

g									m											d
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757	

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

g									m											d
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757	

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

g										d									
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

										g					m					d
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757	

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

										g					m						d
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19		
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757		

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

										g										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757	

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

										g	m								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

										g											d				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19						
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757						

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

												g	d						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

												g	d						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757
												m							

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

												g	d						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

													g						
													d						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757

La recherche dichotomique en action

- ▶ Avant de présenter l'algorithme, illustrons son action sur un exemple simple :

Exemple

Recherchons l'élément 112 :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	3	3	4	7	19	47	47	53	65	66	81	86	112	137	157	163	188	413	757

Recherche par dichotomie (itérative)

Soient `lst` une liste croissante, `x` un élément, `g` et `d` deux indices

- ▶ Tant que l'intervalle `[g, d]` contient au moins un élément :
 - ▶ Calculer `m = (g + d) // 2`
 - ▶ Si `lst[m] == x`, répondre `True`
 - ▶ Si `lst[m] < x`, fixer `g` à `m+1`
 - ▶ Si `lst[m] > x`, fixer `d` à `m-1`

Recherche par dichotomie (itérative)

```
def dichot_iter(lst, x):  
    g, d = 0, len(lst) - 1  
    while g <= d: # au moins un élément non vu  
        # on examine l'élément du milieu  
        m = (g + d) // 2  
        if lst[m] == x:  
            return True  
        elif lst[m] > x:  
            d = m - 1 # on élimine les trop grands  
        else:  
            g = m + 1 # on élimine les trop petits  
    return False # pas trouvé
```

Complexité de la recherche dichotomique

- ▶ Chaque tour de boucle prend un temps $O(1)$
- ▶ Au début du premier tour de boucle, on cherche parmi $\text{len}(\text{lst})$ éléments ($d - g$ vaut $\text{len}(\text{lst}) - 1$)
- ▶ À chaque tour, on élimine une possibilité plus la moitié de ce qui reste ($d - g$ décrémente et divisé par deux)
- ▶ Si $d - g \leq 1$, la boucle s'arrête à la fin du tour

Complexité de la recherche dichotomique

Parmi combien de nombres peut-on trouver à coup sûr en k tours de boucle ?

k	1	2	3	4	5	6	...
n	1	3	7	15	31	63	...

Conjecture : en k tours on peut distinguer parmi $2^k - 1$ nombres

- ▶ Preuve : si on sait chercher parmi n en k questions, on peut chercher parmi $2n + 1$ en $k + 1$ questions
- ▶ Suite déjà rencontrée dans le cours sur les tours de Hanoï

Complexité de la recherche dichotomique

Pour n quelconque, on devra donc chercher au plus en k tours où k est l'unique entier tel que :

$$2^{k-1} - 1 < n \leq 2^k - 1$$

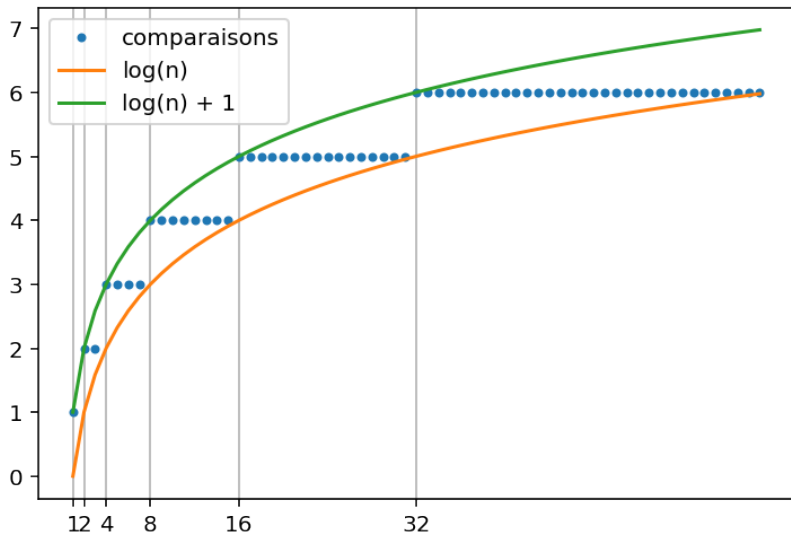
soit $2^{k-1} \leq n < 2^k$

soit $k - 1 \leq \log_2 n < k$

soit $k = \lfloor \log_2 n \rfloor + 1$

On obtient donc une complexité de la recherche dichotomique en $O(\log_2(\text{len}(\text{lst})))$ au pire

Complexité de la recherche dichotomique



Recherche dichotomique (récursive)

```
def dichorec(lst, x, g=0, d=None):  
    if d is None:  
        d = len(lst) - 1  
    if g > d:  
        return False  
    m = (g + d) // 2  
    if lst[m] == x:  
        return True  
    if lst[m] > x:  
        # chercher dans la partie inférieure  
        return dichorec(lst, x, g, m-1)  
    else:  
        # chercher dans la partie supérieure  
        return dichorec(lst, x, m+1, d)
```

Section 2

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

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 3

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 1st.

Objectif : ordonner les éléments de 1st de manière croissante.

Remarques :

1. On suppose que les éléments de 1st sont comparables
2. On travaille ici sur des listes 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 $\text{len}(lst)-2$:

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 :

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

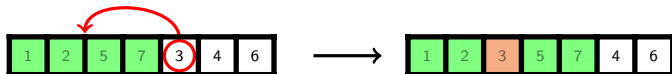
- ▶ Nombre total d'échanges : $n - 1$ dans tous les cas
- ▶ Pas de pire ni de meilleur cas
(complexité indépendante du contenu de la liste)

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 $\text{len}(\text{lst})-1$, on insère $\text{lst}[i]$ "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)

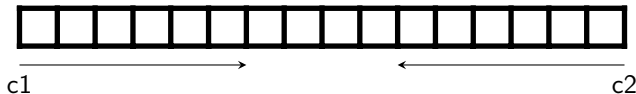
Données : une liste lst de naturels

Objectif : mettre les éléments pairs de lst au début et les impairs à la fin

- ▶ Comment résoudre ce problème de manière efficace ?

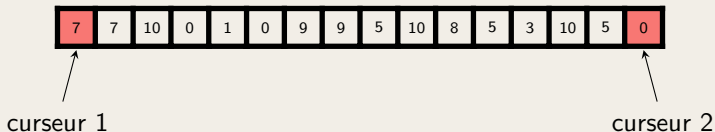
Echauffement : le tri pair / impair

- ▶ 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



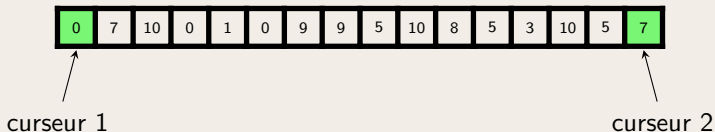
Le tri pair / impair en action

Exemple



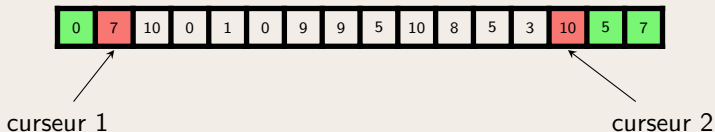
Le tri pair / impair en action

Exemple



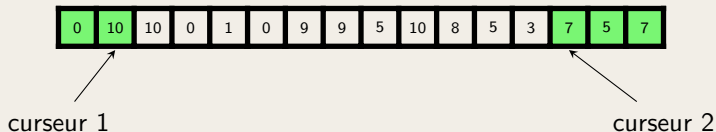
Le tri pair / impair en action

Exemple



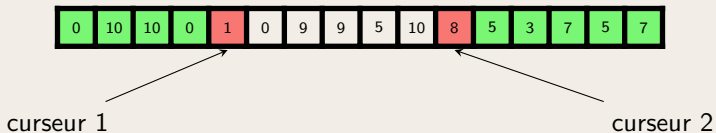
Le tri pair / impair en action

Exemple



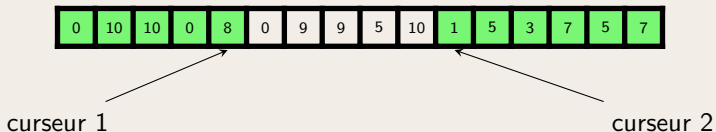
Le tri pair / impair en action

Exemple



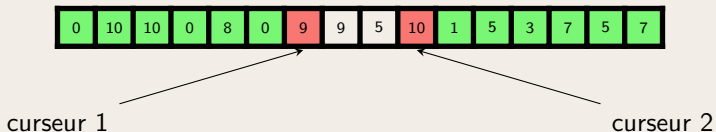
Le tri pair / impair en action

Exemple



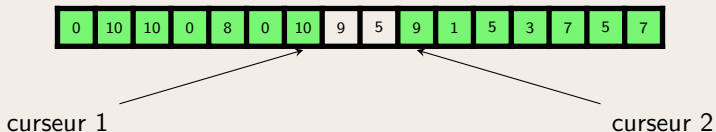
Le tri pair / impair en action

Exemple



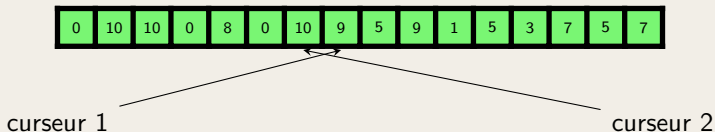
Le tri pair / impair en action

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

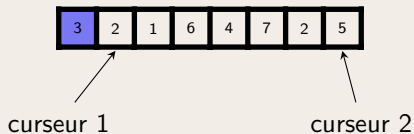
- ▶ On peut utiliser ce principe pour **trier** la liste
- ▶ Au lieu de se contenter de mettre les pairs au début et les impairs à la fin, on peut mettre les éléments inférieurs à une certaine valeur avant elle et les autres après elle
- ▶ Ce principe est à la base du **tri rapide**

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

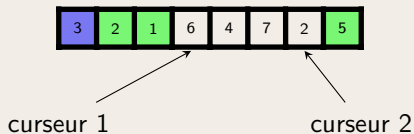
Exemple



1. On partitionne avec le premier élément comme pivot

Le tri rapide

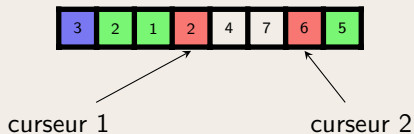
Exemple



1. On partitionne avec le premier élément comme pivot

Le tri rapide

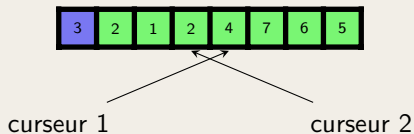
Exemple



1. On partitionne avec le premier élément comme pivot

Le tri rapide

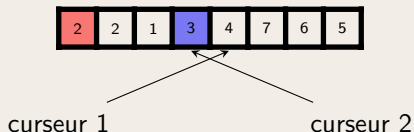
Exemple



1. On partitionne avec le premier élément comme pivot

Le tri rapide

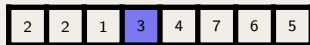
Exemple



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 récurifs 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]

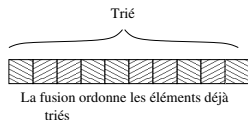
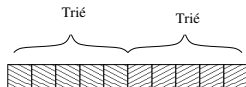
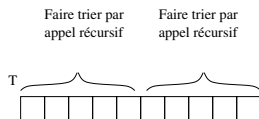
        # partition entre debut et fin
        pivot = partition(lst, debut, fin)

        # 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 4

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 ne les 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 ne les 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 $k^{\text{ème}}$ plus petit

Donnée : une liste de n éléments comparables

Résultat : le $k^{\text{ème}}$ 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 $k^{\text{ème}}$ plus petit

Donnée : une liste de n éléments comparables

Résultat : le $k^{\text{ème}}$ 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!)