# L3 SID APU
# Cours 2
# Méthodes de conception d'algorithmes
# Application aux algorithmes de Tri

Thomas Pellegrini, équipe SAMoVA, IRIT,
thomas.pellegrini@irit.fr

IRIT - UPS

2016-2017

# Méthodes de conception d'algorithmes

## Quelques approches génériques pour concevoir vos algorithmes :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Approche diviser-pour-régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original. Fait appel la plupart du temps à une solution récursive
- Approche gloutonne : construire une solution de manière incrémentale, en optimisant de manière aveugle un critère local
- Approche par programmation dynamique : une ou la solution optimale est trouvée en combinant des solutions optimales d'une série de sous-problèmes (*pas au programme*)

# Méthodes de conception d'algorithmes

## Quelques approches génériques pour concevoir vos algorithmes :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Approche diviser-pour-régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original. Fait appel la plupart du temps à une solution récursive
- Approche gloutonne : construire une solution de manière incrémentale, en optimisant de manière aveugle un critère local
- Approche par programmation dynamique : une ou la solution optimale est trouvée en combinant des solutions optimales d'une série de sous-problèmes (*pas au programme*)

# Méthodes de conception d'algorithmes

## Quelques approches génériques pour concevoir vos algorithmes :

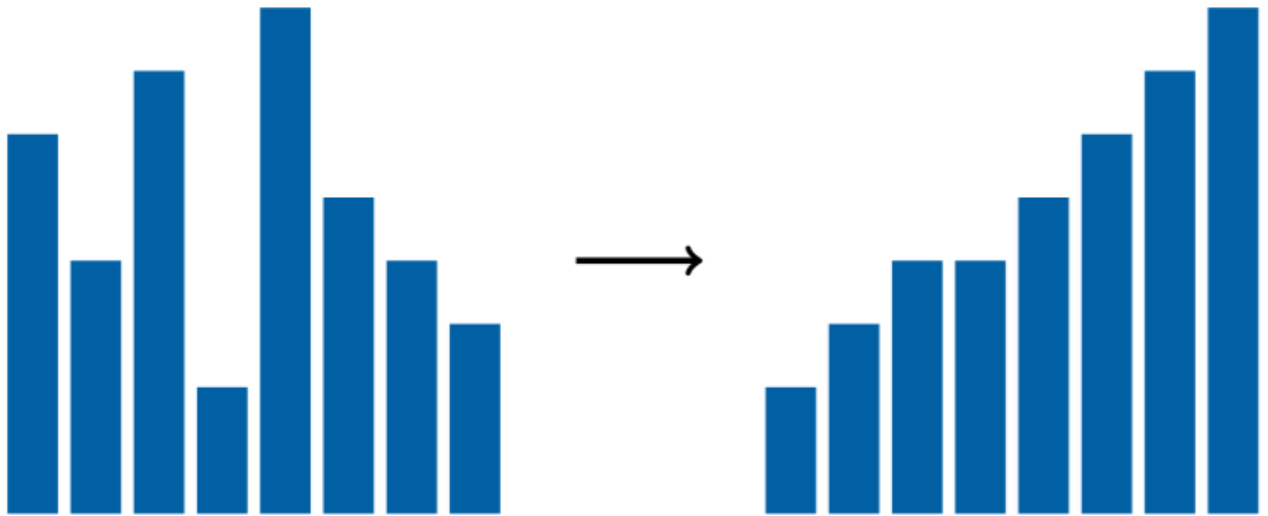
- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Approche diviser-pour-régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original. Fait appel la plupart du temps à une solution récursive
- Approche gloutonne : construire une solution de manière incrémentale, en optimisant de manière aveugle un critère local
- Approche par programmation dynamique : une ou la solution optimale est trouvée en combinant des solutions optimales d'une série de sous-problèmes (*pas au programme*)

# Méthodes de conception d'algorithmes

## Quelques approches génériques pour concevoir vos algorithmes :

- Approche par force brute : résoudre directement le problème, à partir de sa définition ou par une recherche exhaustive
- Approche diviser-pour-régner : diviser le problème en sous-problèmes, les résoudre, fusionner les solutions pour obtenir une solution au problème original. Fait appel la plupart du temps à une solution récursive
- Approche gloutonne : construire une solution de manière incrémentale, en optimisant de manière aveugle un critère local
- Approche par programmation dynamique : une ou la solution optimale est trouvée en combinant des solutions optimales d'une série de sous-problèmes (*pas au programme*)

# Le problème du tri



# Le problème du tri

- Un des problèmes algorithmiques les plus fondamentaux
- En général, on veut trier des enregistrements avec une clé et des données attachées
- Ici, on va ignorer ces données et se focaliser sur les algorithmes de tri des clés

# Le problème du tri

## Applications innombrables

- Tri des mails selon leur ancienneté
- Tri des résultats de requête dans un moteur de recherche
- Tri des facettes des objets pour l'affichage dans les jeux 3D
- Gestion des opérations bancaires
- ...

## Le tri sert aussi de brique de base / pré-traitement pour d'autres algos

- Recherche binaire dans un tableau trié
- Recherche des éléments dupliqués dans un tableau
- Recherche du k*ième* élément le plus grand dans une liste
- ...

# Le problème du tri

## Formulation

- Entrée : une suite de $n$ nombres
- Sortie : une permutation (réorganisation) $a'_1, a'_2, \ldots, a'_n$ de façon que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

## Exemple

- Entrée : [8, 4, 2, 5, 2]
- Sortie : [2, 2, 4, 5, 8]

# Approches par force brute pour le tri

## Exemples communs

- Tri à bulle — *bubble sort* : parcourir le tableau de gauche à droite en échangeant toutes les paires d'éléments consécutifs nécessaires
- Tri par sélection : trouver le minimum du tableau, l'échanger avec le premier élément et répéter pour trier le reste du tableau
- Tri par insertion : tri de jeu de cartes dans sa main

# Approches par force brute pour le tri

## Exemples communs

- Tri à bulle — *bubble sort* : parcourir le tableau de gauche à droite en échangeant toutes les paires d'éléments consécutifs nécessaires
- Tri par sélection : trouver le minimum du tableau, l'échanger avec le premier élément et répéter pour trier le reste du tableau
- Tri par insertion : tri de jeu de cartes dans sa main

# Approches par force brute pour le tri

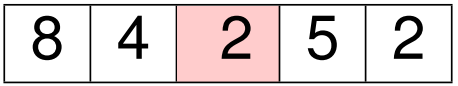
## Exemples communs

- Tri à bulle — *bubble sort* : parcourir le tableau de gauche à droite en échangeant toutes les paires d'éléments consécutifs nécessaires
- Tri par sélection : trouver le minimum du tableau, l'échanger avec le premier élément et répéter pour trier le reste du tableau
- Tri par insertion : tri de jeu de cartes dans sa main

# Tri par sélection

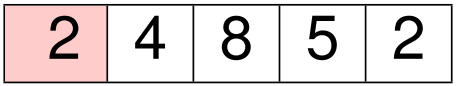
| 8 | 4 | 2 | 5 | 2 |
|---|---|---|---|---|

# Tri par sélection



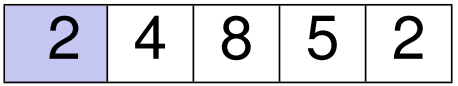

- Trouver le minimum du tableau en le parcourant

# Tri par sélection



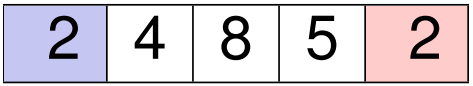

- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau

# Tri par sélection



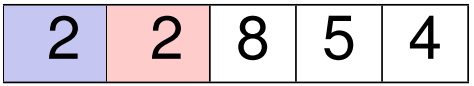

- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection



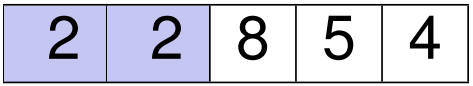

- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection



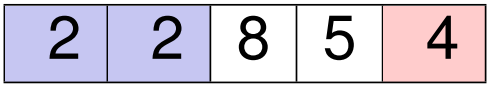

- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection



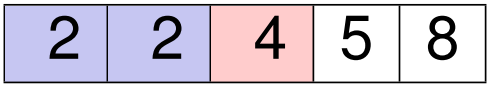

- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection




- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection




- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection


- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection


- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection


- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Tri par sélection


- Trouver le minimum du tableau en le parcourant
- Echanger ce plus petit élément avec le premier élément du tableau
- Répéter avec le reste du tableau

# Apparté : comment échanger deux variables a et b ?

En pseudo-code, si a et b sont des entiers :

1: temp : Entier $\leftarrow$ a
2: a $\leftarrow$ b
3: b $\leftarrow$ temp

Pour simplifier la lecture des algos par la suite, on utilisera l'expression "échanger(a, b)" pour signifier ces trois lignes de pseudo-code.

En python et seulement en python, il est possible de le faire en une seule ligne de code :

```
a, b = b, a
```

# Tri par sélection

Soit *A* le nom du tableau pris en entrée par notre pseudo-code

**Algorithme** TRI-SELECTION(A)

```
1: POUR i de 1 à n FAIRE
2:     minIndice ← i
3:     POUR j de i+1 à n FAIRE
4:         SI A[j] < A[minIndice] ALORS
5:             minIndice ← j
6:     échanger(A[i], A[minIndice])
```

# Tri par sélection : en python

```
A = [8, 4, 2, 5, 2]
print("avant tri :", A)
n=len(A)
for i in range(n):
    minIndice = i
    for j in range(i+1, n):
        if A[j] < A[minIndice]:
            minIndice = j
    A[i], A[minIndice] = A[minIndice], A[i]
print("apres tri :", A)
```

# Tri par sélection : en python, avec une fonction

```
def triSelection(tab):
    n=len(tab)
    for i in range(n):
        minInd = i
        for j in range(i+1, n):
            if tab[j] < tab[minInd]:
                minInd = j
        tab[i], tab[minInd] = tab[minInd], tab[i]
    return tab

A = [8, 4, 2, 5, 2]
print("avant tri :", A)
A_tri = triSelection(A)
print("apres tri :", A_tri)
```

# Trois questions récurrentes à se poser sur un algo

1. Mon algorithme est-il correct ?
2. Quelle est sa vitesse ou temps d'exécution ?
3. Peut-on faire mieux ?

Dans notre exemple de tri par sélection :

1. Oui, la correction d'un algorithme se démontre par la technique des invariants (hors-sujet pour ce cours)
2. Cet algo est en $O(n^2) \rightarrow$ analyse de la complexité de l'algo
3. Oui, nous verrons un algo plus rapide, en $O(n \log n)$

# Analyse de complexité

- Analyser la complexité d'un algorithme c'est estimer son temps d'exécution qui correspond à la durée des opérations effectuées par l'ordinateur
- En général ce temps croît avec la $\ll$ taille $\gg$ de l'entrée : tri de trois nombres versus tri de mille nombres ?

$\Rightarrow$ On estime ce temps en fonction de la taille de l'entrée : nombre d'éléments du tableau à trier, nombre de sommets et d'arcs d'un arbre à parcourir, etc.
$\Rightarrow$ Cette analyse permet de comparer l'efficacité de deux algos entre eux, de manière indépendante de l'ordinateur utilisé

## Parti pris

- Chaque ligne de pseudo-code prend un temps constant : la ième ligne prend un temps constant $c_i$

# Complexité de TRI-SELECTION

| **Algorithme** TRI-SELECTION(A) | Coût | fois |
|---|---|---|
| 1: POUR i de 1 à n FAIRE | $c_1$ | $n$ |
| 2: minIndice $\leftarrow$ i | $c_2$ | $n$ |
| 3: POUR j de i+1 à n FAIRE | $c_3$ | $\sum_{i=1}^{n} n-i$ |
| 4: SI A[j] $<$ A[minIndice] ALORS | $c_4$ | $\sum_{i=1}^{n} n-i$ |
| 5: minIndice $\leftarrow$ j | $c_5$ | $\sum_{i=1}^{n} n-i$ |
| 6: échanger(A[i], A[minIndice]) | $c_6$ | $n$ |

Pour obtenir le coût total $T(n)$, on additionne les produits des colones *coût* et *fois* :

$$T(n) = c_1 n + c_2 n + c_3 \sum_{i=1}^{n} n-i + c_4 \sum_{i=1}^{n} n-i + c_5 \sum_{i=1}^{n} n-i + c_6 n$$

# Complexité de TRI-SELECTION

$$T(n) = c_1 n + c_2 n + c_3 \sum_{i=1}^{n} n - i + c_4 \sum_{i=1}^{n} n - i + c_5 \sum_{i=1}^{n} n - i + c_6 n$$

$$= (c_1 + c_2 + c_6)n + (c_3 + c_4 + c_5) \sum_{i=1}^{n} n - i$$

On a

$$\sum_{i=1}^{n} n - i = n * n - \sum_{i=1}^{n} i$$

$$= n^2 - \frac{n(n+1)}{2}$$

$$= \frac{n^2}{2} - \frac{n}{2}$$

# Complexité de TRI-SELECTION

$$T(n) = (c_1 + c_2 + c_6)n + (c_3 + c_4 + c_5)(\frac{n^2}{2} - \frac{n}{2})$$
$$= \frac{(c_3 + c_4 + c_5)}{2} n^2 + (c_1 + c_2 + c_6 - \frac{c_3 + c_4 + c_5}{2})n$$

Le temps d'exécution s'exprime donc sous la forme $an^2 + bn$, avec $a, b$ des constantes qui dépendent des coûts $c_i$
$\Rightarrow$ C'est donc une **fonction quadratique** de n
Nous dirons :
« la complexité de TRI-SELECTION est en « grand o » de $n^2$ »,
car le terme quadratique $n^2$ domine le terme linéaire $n$
$\Rightarrow$ Notation de Landau : $T(n) = O(n^2)$

# Complexité de TRI-SELECTION

Le temps d'exécution de TRI-SELECTION sera-t-il différent si en entrée on donne : un tableau déjà trié ou un tableau aléatoire, tous les deux de même taille ?

# Complexité de TRI-SELECTION

Le temps d'exécution de TRI-SELECTION sera-t-il différent si en entrée on donne : un tableau déjà trié ou un tableau aléatoire, tous les deux de même taille ?

Réponse : dans le cas d'un tableau déjà trié, cas appelé « cas le plus favorable », la condition de la ligne 4 n'est jamais vraie et donc la ligne 5 n'est jamais exécuté. Il n'en reste pas moins que le temps d'exécution reste quadratique et ne dépend donc pas de la particularité du tableau donné en entrée.

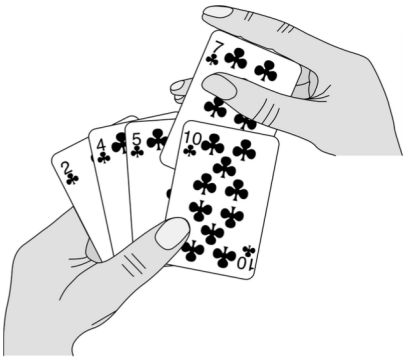
Pour caractériser plus précisément la complexité d'algos de tri, on pourra distinguer :

- cas le plus favorable : tableau déjà trié
- cas moyen : tableau pas spécialement ordonné
- cas le plus défavorable : tableau trié en ordre décroissant

# Complexité de TRI-SELECTION

Démo online : https://www.toptal.com/developers/sorting-algorithms/

# TRI-INSERTION



## Un autre algorithme de tri : TRI-INSERTION

- Principe identique à un joueur tenant ses cartes dans sa main gauche : au début sa main gauche est vide et il prend les cartes sur la table une par une de sa main droite pour les placer correctement classées dans sa main gauche. A tout moment, les cartes tenues par la main gauche sont correctement triées.

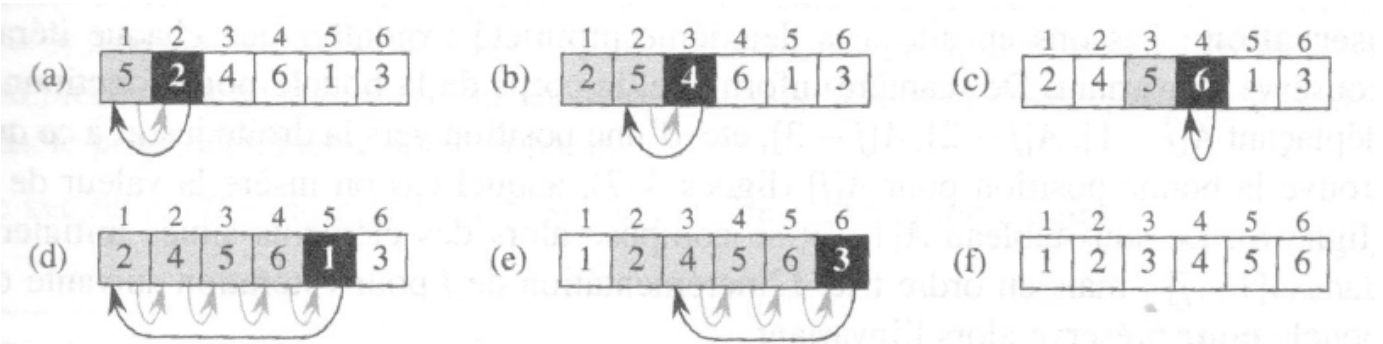
# TRI-INSERTION

Description en langage naturel

- On parcourt le tableau de gauche à droite
- Pour chaque ième élément du tableau : on le place au bon endroit dans la séquence déjà triée composée de (i-1) éléments
- On s'arrête quand le dernier élément a été inséré dans la séquence

Remarque : TRI-INSERTION est un algo de tri dit ≪ **sur place** ≫, au même titre que TRI-SELECTION, dans le sens où le tri est réalisé directement sur le tableau lui-même, sans avoir à créer un tableau supplémentaire temporaire pour stocker certains éléments

# TRI-INSERTION




La suite en TD...