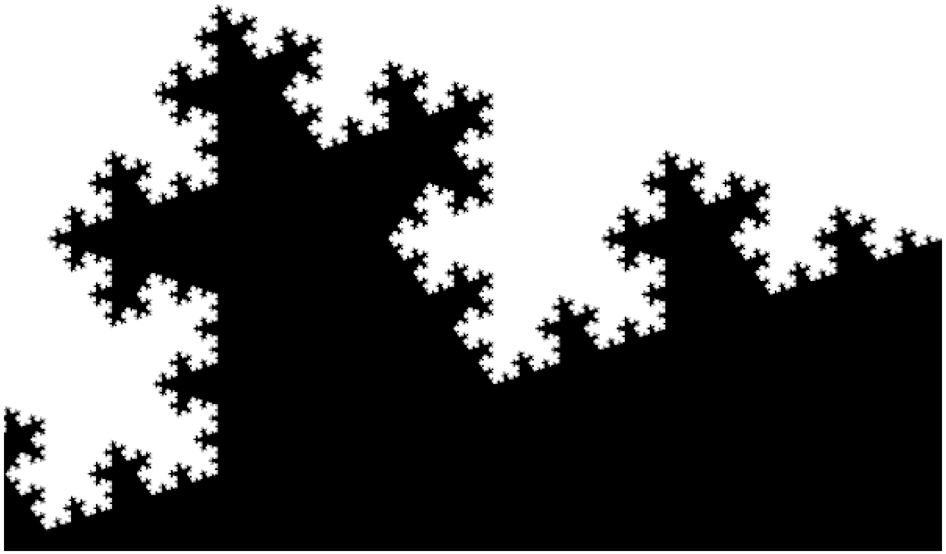
Base de programmation

- BA1 Informatique
  Johan Depréter – johan.depreter@heh.be

La récursivité

- La récursivité est une démarche qui fait référence à l’objet même de la démarche pendant son processus

- Exemples :
  - Le calcul d’une factorielle
  - La suite de fibonacci

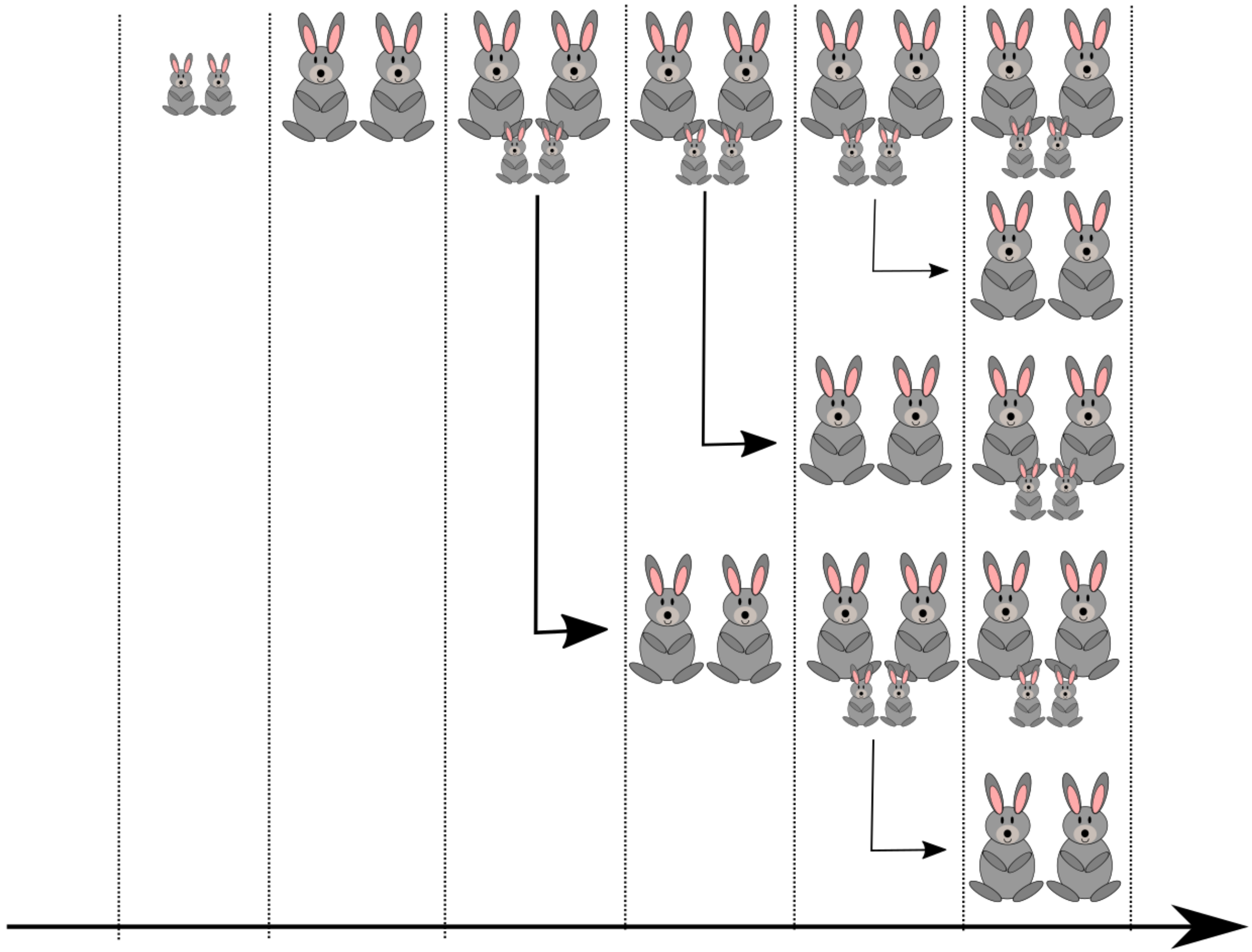


Problème des lapins

- Le problème posé est le suivant :

« Quelqu'un a déposé un couple de lapins dans un certain lieu, clos de toutes parts, pour savoir combien de couples seraient issus de cette paire en une année, car il est dans leur nature de générer un autre couple en un seul mois, et qu'ils enfantent dans le second mois après leur naissance. »

Problème des lapins



Problème des lapins

- Formulation de la classe du problème :

  En sachant qu'un couple de lapins génère un nouveau couple de lapin chaque mois à partir de leur 2ème mois d'existence, après <u>x</u> mois combien aurais-je de couple de lapins ?

- Spécifications du problème :

  Entrée a : nombre de mois

  Pré-condition : a réel positif

  Sortie m : nombre de couples de lapins

  Post-condition : ?

Suite de Fibonacci

0	1	2	3	4	5	6	7	8
0	1	1	2	3	5	8	13	21

- Cas général :

  Un élément est égal à la somme des deux éléments qui le précèdent

- Cas de base :

  L’élément 0 vaut 0 et l’élément 1 vaut 1

Suite de Fibonacci

```python
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```

Contexte d'exécution

- Pile d'exécution

```
fibonacci(5)
    ├─ fibonacci(4)
    │     ├─ fibonacci(3)
    │     │     ├─ fibonacci(2)
    │     │     │     ├─ fibonacci(1)
    │     │     │     └─ fibonacci(0)
    │     │     └─ fibonacci(1)
    │     └─ fibonacci(2)
    │           ├─ fibonacci(1)
    │           └─ fibonacci(0)
    └─ fibonacci(3)
          ├─ fibonacci(2)
          │     ├─ fibonacci(1)
          │     └─ fibonacci(0)
          └─ fibonacci(1)
```

Les preuves

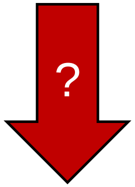
1. Preuve d'arrêt
   - Bien fondé
   - Appel avec des paramètres de valeurs inférieurs

2. Preuve de validité
   - Correction partielle
   - Démontrer que si l'algorithme fonctionne pour n-1, alors il fonctionne pour n

Dérécursivation

- Passer d’un algorithme récursif à itératif

```python
def fibo(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n - 1) + fibo(n - 2)
```




```python
def fiboI(n):
    a, b = 0, 1
    for i in range(0, n):
        a, b = b, a+b
    return a
```

Pourquoi ?

- Avantages de la récursivité
  - Simple à comprendre
  - Simple à lire

➡ Plus intuitif

- Inconvénients de la récursivité
  - Utilisation accrue de la mémoire
  - Utilisation accrue du CPU

➡ Moins efficace

Chapitre 2

Exercices

Factorielle

- Problème :

  Calculer la factorielle d'un nombre réel positif <u>*p*</u>

$$n! = 1 * 2 * 3 * \cdots * (n-1) * n$$
$$= (n-1)! * n$$

- Rédiger le code python de la version récursive de l'algorithme

Correction

```
def factR(n):
    if n <= 1:
        return 1
    else:
        return factR(n-1)*n
```

Correction

```python
def factI(n):
    a = 1
    for i in range(1, n+1):
        a = a*i
    return a
```

Tours de Hanoï

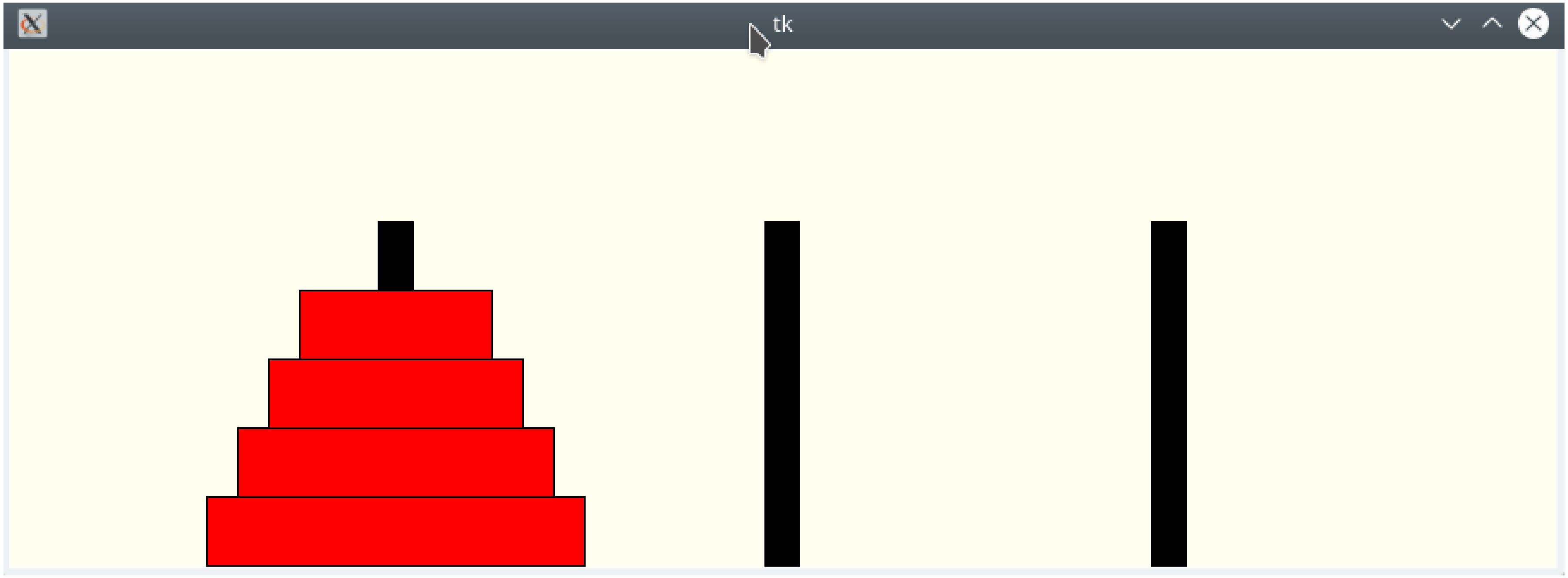
- Problème :

  On a 7 disques de diamètres différents qui forment une tour. On souhaite déplacer ces disques vers une nouvelle tour en suivant les règles suivantes :

  - On ne peut pas déplacer plus d'un disque à la fois
  - On ne peut placer un disque que sur un disque plus grand (ou sur un emplacement vide)

  Trouver comment résoudre ce problème avec le moins de déplacement possible, et en utilisant une tour intermédiaire.

Tours de Hanoï



Tours de Hanoï

- Rédiger les spécifications de la classe du problème
- Formaliser le problème
- Ecrire, en Python, la fonction hanoi() qui permet de calculer le nombre de coup minimum nécessaire.
- Est-il possible de généraliser la classe du problème. Si oui, comment ?

Correction

- Spécification de la classe du problème :

  Entrée n – nombre de disques

  Entrée D – Tour de départ

  Entrée I – Tour intermédiaire

  Entrée A – Tour d'arrivée

  Pré-condition – n > 0

  Sortie z – nombre de « coups »

  Post-condition – $z = 2^n - 1$

  (Respect des règles du jeu)

Correction

- Formalisation

Etape 1 : Démarrer

Etape 2 : Lire D, I, A et n

Etape 3 : Déplacer (n-1) disques de la source vers l'intermédiaire

Etape 4 : Déplacer le disque n de la source vers la destination

Etape 5 : Déplacer (n-1) disques de l'intermédiaire vers la destination

Etape 6 : Recommencer les étapes 3 à 5 avec n-1

Etape 7 : Stop

Correction

```python
def hanoi(n, D, A, I):
    global count
    count += 1
    if n == 1:
        print("Déplacer disque 1 du départ", D, "vers la destination", A)
        return
    hanoi(n-1, D, I, A)
    print("Déplacer disque", n, "du départ", D, "vers la destination", A)
    hanoi(n-1, I, A, D)
```

Chapitre 6

Les algorithmes de tri

Introduction

- A quoi ca sert de trier ?
- Tri en place / non en place
- Tri stable / non stable

Complexité

- Permet de mesurer la performance

- Complexité temporelle
  - Permet de quantifier la vitesse d’exécution

- Complexité spatiale
  - Permet de quantifier l’utilisation de la mémoire

Complexité temporelle

- Compter le nombre d'opérations élémentaires

- La taille des données, notée ***n***

- La donnée en question
  - Calcul dans le meilleur des cas
  - Calcul dans le pire des cas
  - Calcul dans le cas moyen

Tri à bulles

8 5 3 1 4 7 9

1. Parcourir la liste

2. A chaque élément elle va comparer avec le suivant et changer leur position

Tri à bulles

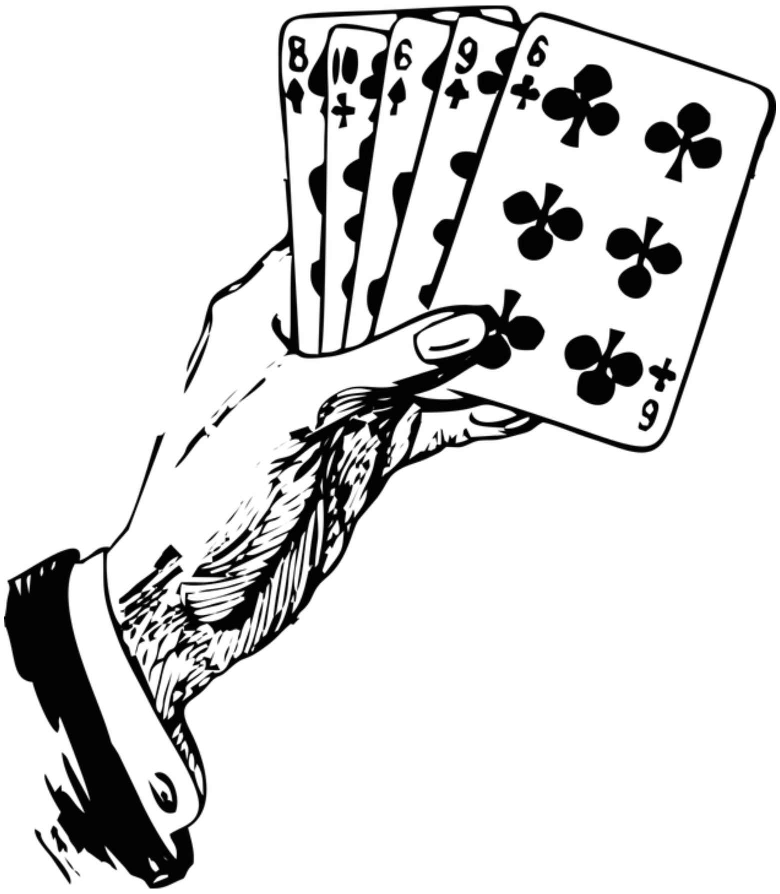
8	**5**	**3**	**1**	**4**	**7**	**9**
5	8	3	1	4	7	9
5	3	8	1	4	7	9
5	3	1	8	4	7	9
5	3	1	4	8	7	9
5	3	1	4	7	8	9
3	5	1	4	7	8	9
3	1	5	4	7	8	9

→ 1 3 4 5 7 8 9

Tri à bulles

Complexité temporelle	
Meilleur	O(n)
Pire	O(n^2)
Moyenne	O(n^2)
Complexité spatiale	O(1)
Stabilité	Oui

Tri par insertion



8 10 6 9 6

1. Parcourir la liste
2. Comparer à l’élément précédent
3. Déplacer l’élément le plus grand pour « faire de la place »

Tri par insertion

8 10 6 9 6

6 8 10 9 6

6 8 9 10 6

6 6 8 9 10

Tri par insertion

Complexité temporelle	
Meilleur	O(n)
Pire	O(n^2)
Moyenne	O(n^2)
Complexité spatiale	O(1)
Stabilité	Oui

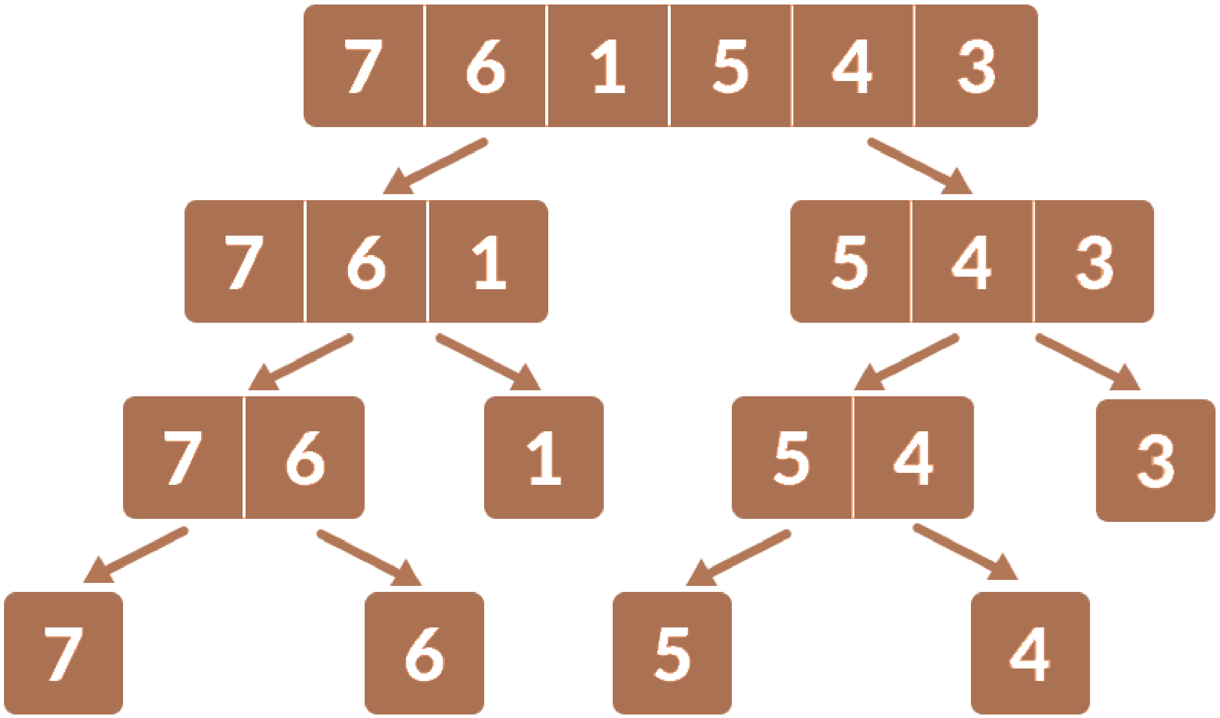
Tri par fusion

- « Divide and Conquer »

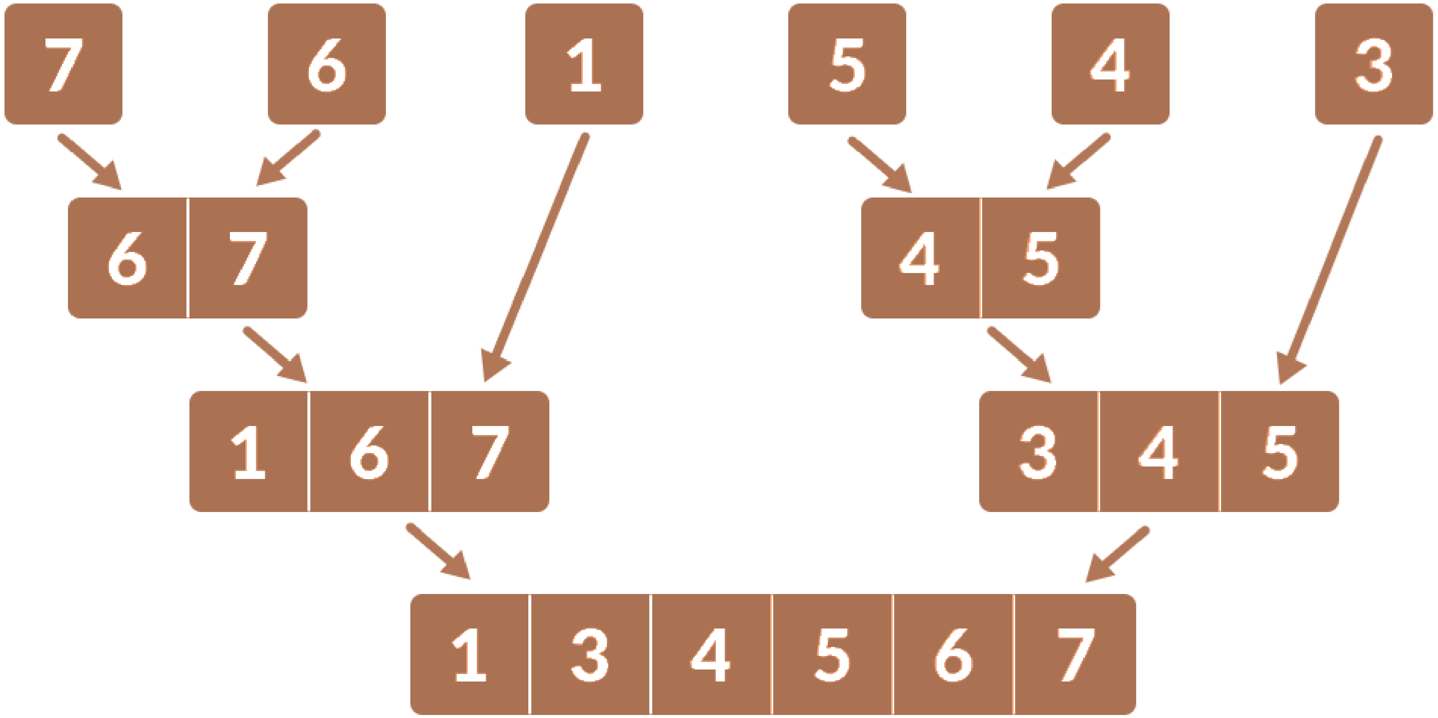
  1. Diviser
  2. Régner
  3. Combiner

- Méthode récursive

Tri par fusion



Tri par fusion



Tri par fusion

- Algorithme :

    1. Si la liste n’a qu’un élément, elle est triée
    2. Séparer la liste en 2 listes +/- égales
    3. Trier chacune des listes selon le tri par fusion
    4. Fusionner les 2 listes en une seule liste triée

Tri par fusion

Complexité temporelle	
Meilleur	$O(n \log n)$
Pire	$O(n \log n)$
Moyenne	$O(n \log n)$
Complexité spatiale	$O(n)$
Stabilité	Oui

Tri rapide

- Principe similaire au tri par fusion
- Une des méthodes les plus utilisées

Tri rapide

- Utilisation d’un pivot pour séparer la liste en 2 sous listes

- Différents choix possibles du pivot :
  - 1er élément
  - Dernier élément
  - Élément aléatoire
  - Élément médian

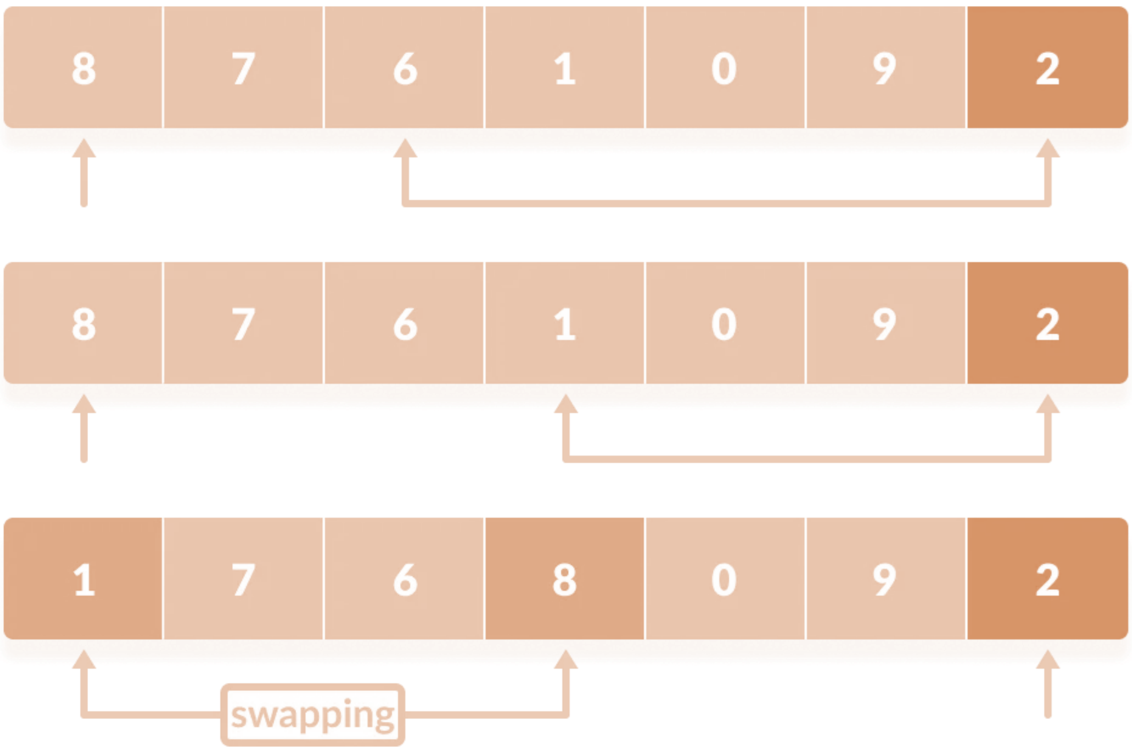
Tri rapide

8	7	6	1	0	9	2

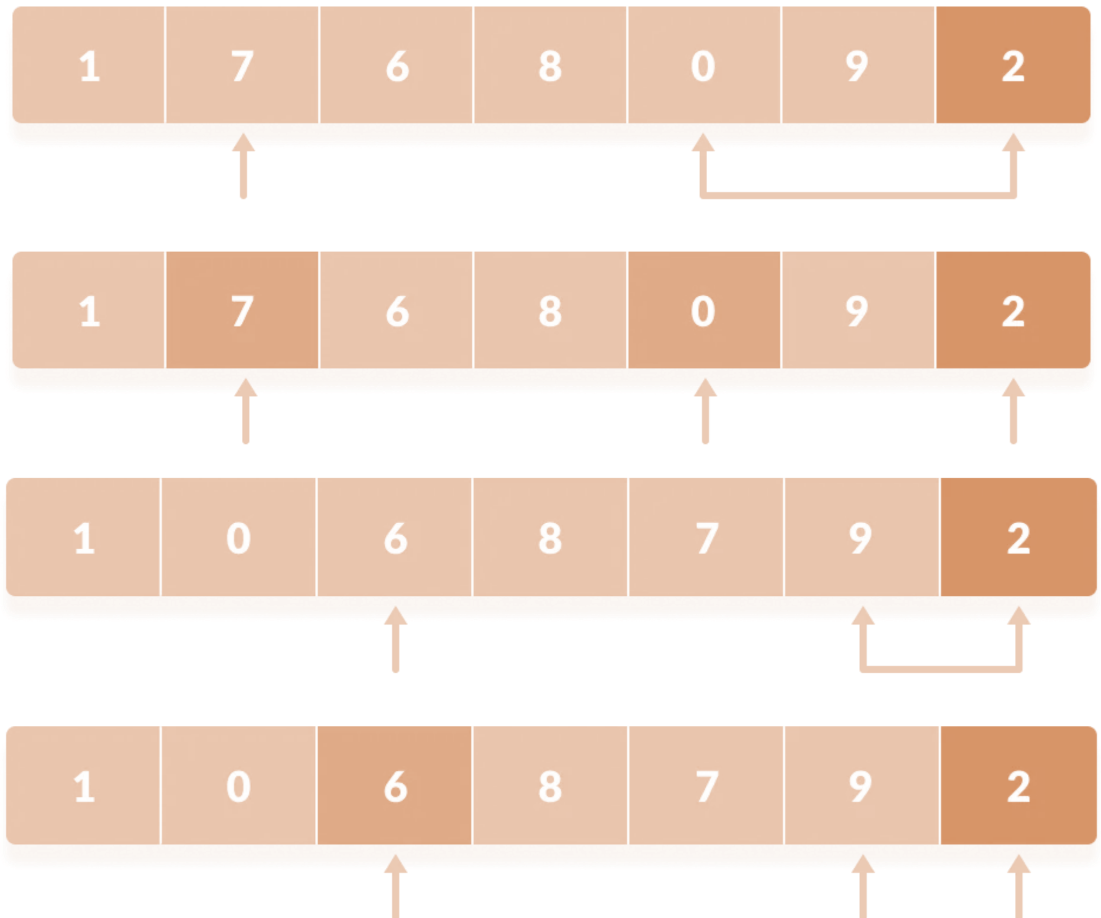
↓

1	0	2	8	7	9	6

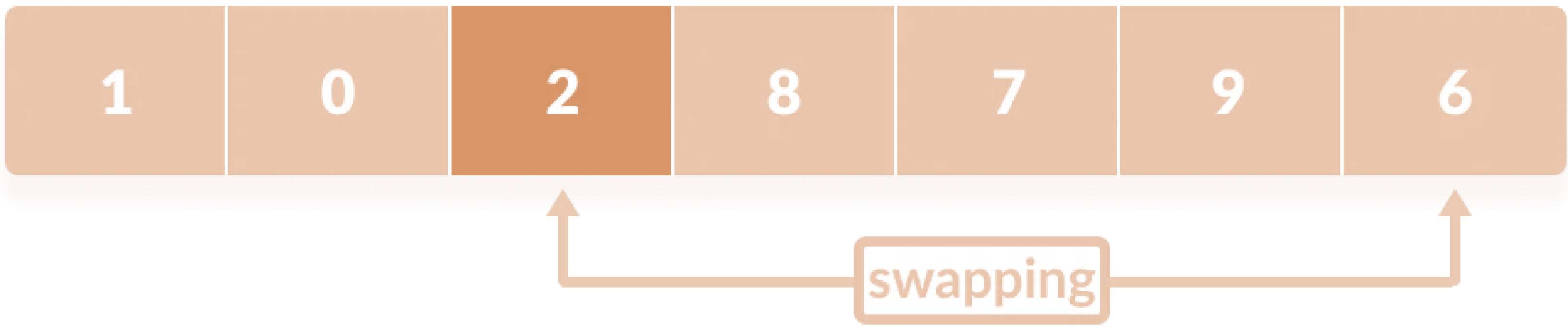
Tri rapide



Tri rapide



Tri rapide



Tri rapide

Complexité temporelle	
Meilleur	O($n \log n$)
Pire	O(n^2)
Moyenne	O($n \log n$)
Complexité spatiale	O($\log n$)
Stabilité	Non

Récapitulatif

Nom	Meilleur cas	Pire cas	Cas moyen	Mémoire
Tri à bulles	n	n^2	n^2	1
Tri par insertion	n	n^2	n^2	1
Tri rapide	$n \log n$	n^2	$n \log n$	1
Tri par fusion	$n \log n$	$n \log n$	$n \log n$	n

Applications

- Tri à bulles :
  - Code simple et facile
  - Complexité peu / pas importante

- Tri par insertion :
  - Peu d'éléments dans la liste
  - Peu d'éléments restants à trier

Applications

- Tri par fusion :
  - Compter le nombre d'inversion restante
  - Tri externe
- Tri rapide :
  - Complexité temporelle et spatiale sont importantes
- Tri par tas :
  - Systèmes embarqués et sécurisés (Kernel Linux)

Chapitre 6

Exercices

Problème du drapeau hollandais

Problème du drapeau hollandais

- Problème :

  Étant donné un nombre quelconque de balles rouges, blanches et bleues alignées dans n'importe quel ordre, le problème est à les réarranger dans le bon ordre : les rouges d'abord, puis les blanches, puis les bleues.

  L’algorithme de tri doit être stable.

Correction

- Donner des valeurs numériques
  Rouge = 0, Blanc = 1 et Bleu = 2

- La liste va alors être composée de 4 sections :

  1. L[0… Low-1] 0 (rouges)
  2. L[Low… Mid-1] 1 (blancs)
  3. L[Mid… High] valeurs inconnues
  4. L[High+1… N-1] 2 (bleus)

Correction

- Algorithme

  1. 3 indices low = 1, mid = 1 et high = N-1 avec les 4 sections.

  2. Parcourir la liste du début à la fin et tant que mid est plus petit que high

  3. Si l'élément vaut 0 alors on l'intervertit avec l'élément se trouvant à low et on met a jour : low += 1 et mid += 1

  4. Si l'élément vaut 1 alors on met à jour mid += 1

  5. Si l'élément vaut 2 alors on l'intervertit avec l'élément se trouvant à high et on met à jour : high -= 1

Correction

2	0	0	1	2	1
low					high
mid					

1	0	0	1	2	2
low				high	
mid					

1	0	0	1	2	2
low				high	
	mid				

0	1	0	1	2	2
	low			high	
		mid			

0	0	1	1	2	2
		low		high	
			mid		

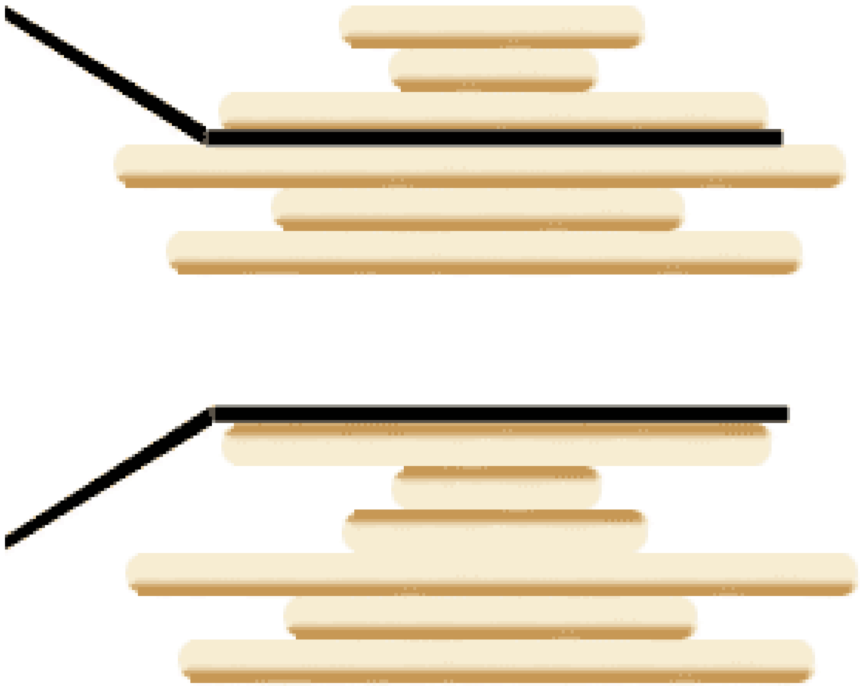
0	0	1	1	2	2
		low		high	
				mid	

0	0	1	1	2	2
		low		high	
					mid

Correction

```python
def sort(a, size):
    lo = 0
    hi = size - 1
    mid = 0
    while mid <= hi:
        if a[mid] == 0:
            a[lo], a[mid] = a[mid], a[lo]
            lo = lo + 1
            mid = mid + 1
        elif a[mid] == 1:
            mid = mid + 1
        else:
            a[mid], a[hi] = a[hi], a[mid]
            hi = hi - 1
    return a
```

Tri de crêpes



Tri de crêpes

- Problème :

  Trier une pile de crêpes de sorte qu’elles soient empilées de la plus petite à la plus grande. La seule opération disponible est le fait d’insérer une spatule à un endroit dans la pile et de retourner toutes les crêpes par-dessus d’un coup.

  Donner le nombre de mouvements minimums pour une pile d’une taille n.