

# Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 2 : algorithmes récurrents, analyse en moyenne

Benjamin Wack



2025 – 2026

## La dernière fois

- ▶ Qu'est-ce qu'un algorithme
- ▶ Notion de coût
- ▶ Notions de complexité (au pire) et ordres de grandeur

## Aujourd'hui

- ▶ Comment écrire un algorithme récursif ?
- ▶ Comment évaluer son coût ?
- ▶ Comment évaluer l'efficacité d'un algorithme plus finement que dans le pire cas ?
- ▶ Tri rapide

# Plan

- Algorithme récursif
  - Schémas récursifs
  - Analyse de coût

- Analyse en moyenne
  - Contexte
  - Méthode

- Tri par segmentation (« Tri rapide »)

# Notion d'algorithme récursif

Aux constructions habituelles on ajoute la possibilité d'appeler l'algorithme *lui-même* sur une autre donnée.

## Calcul de la factorielle

FACT( $n$ )

**Données** : un entier  $n$

**Résultat** : la valeur de  $n! = n \times (n - 1) \times \dots \times 2 \times 1$

si  $n = 0$

└ renvoyer 1

renvoyer  $n \times \text{FACT}(n - 1)$

- ▶ caractéristique de l'**algorithme**, pas du problème
- ▶ non limité aux entiers, et particulièrement adapté aux **structures récursives**
- ▶ ce n'est pas de la **triche**!  
Travail fourni = identifier la structure récursive du problème

# Structure

## Prérequis

- ▶ Savoir directement **résoudre** le problème sur des **cas de base**
- ▶ Savoir **utiliser** une (ou des) instances **plus petites** pour résoudre le problème

« plus petit » = entier inférieur, sous-arbre, liste plus courte... mais pas seulement

## PAIR( $n$ )

**Données** : un entier  $n$

**Résultat** : booléen «  $n$  est-il pair ? »

si  $n = 0$

└ renvoyer *Vrai*

sinon

└ renvoyer PAIR( $n - 2$ )

Tout chemin d'exécution doit mener à un cas de base en temps fini.

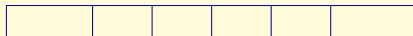
# Écriture d'un algorithme récursif

## Un problème qui casse pas des briques

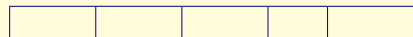
On cherche à construire un mur de longueur  $n$  avec des briques de longueur 2 ou 3.



$$2+2+3+2+3+2$$



$$3+2+2+2+2+3$$



$$3+3+3+2+3$$

Exemples pour  $n = 14$

Combien y a-t-il de possibilités distinctes ?

# Analyse du problème

- ▶ **Données :**  
la longueur  $n$  du mur (les tailles des briques sont fixées)  
(la récursivité ne peut donc porter que sur  $n$ )
- ▶ **Utilisation de sous-problèmes :**  
Une fois la première brique posée il reste un mur de longueur  $< n$ .
- ▶ **Reconstruction de la solution :** Les arrangements de longueur  $n$   
= ceux commençant par une brique de longueur 2  
+ ceux commençant par une brique de longueur 3
- ▶ **Cas de base :** pour  $n = 0$  ou  $n = 2$  un seul choix;  $n = 1$  impossible

```
BRIQUES( $n$ )  
si  $n = 1$   
  └ renvoyer 0  
sinon si  $n = 0$  ou  $n = 2$   
  └ renvoyer 1  
sinon  
  └ renvoyer  $BRIQUES(n - 2) + BRIQUES(n - 3)$ 
```

## Exemple : Conversion en base 2

### Spécification

Étant donné un entier  $n$ , on cherche à produire à l'écran l'écriture binaire de  $n$ .

### Idée

On sait que :

- ▶  $n \% 2$  donne le **dernier** bit
- ▶ les bits les plus à gauche sont aussi ceux de  $n/2$ .

### Remarque

Le problème ne se résout pas naturellement de manière impérative : il faut « commencer par la fin ».

## Exemple : Conversion en base 2 (suite)

### Structure récursive

- ▶ Les cas  $n = 0$  et  $n = 1$  sont connus.
- ▶ On sait résoudre le problème si on a résolu l'instance  $n/2$ .
  
- ▶ Précondition :  $n$  est un entier  $\geq 0$
- ▶ Postcondition : on a produit l'écriture binaire de  $n$

### Écriture binaire d'un entier

ECRITURE\_BINAIRE( $n$ )

**si**  $n < 2$

└ Écrire  $n$

**sinon**

└ ECRITURE\_BINAIRE( $n/2$ )

└ Écrire  $n \% 2$

Un algorithme récursif ne se termine **pas toujours** par l'appel récursif.

Et si on voulait stocker le résultat dans un tableau ?

# Retour sur la factorielle

## Complément au cours précédent

Le coût de l'appel d'une procédure est :

- ▶ le coût du corps de la procédure pour ses paramètres d'appel
- ▶ plus le coût de l'évaluation de ses paramètres.

## Calcul de la factorielle

$FACT(n)$

si  $n = 0$

└ renvoyer 1

renvoyer  $n \times FACT(n - 1)$

$$\text{D'où } \begin{cases} \text{Coût}_{FACT}(0) & = 0 \\ \text{Coût}_{FACT}(n) & = 3 + \text{Coût}_{FACT}(n - 1) \end{cases}$$

La fonction de coût d'un algorithme récursif obéit généralement elle-même à une équation récursive.

## Quelques équations récursives classiques

Un seul appel + corps de fonction en temps constant  $k$

$$\text{Si } \text{Cost}(n) = k + \text{Cost}(n - 1)$$

$$\text{alors } \text{Cost}(n) = k \times n + \text{Cost}(0) = \mathcal{O}(n) \quad (\text{coût linéaire})$$

Un seul appel + corps de fonction qui coûte  $f(n)$

$$\text{Si } \text{Cost}(n) = \text{Cost}(n - 1) + f(n)$$

alors

$$\text{Cost}(n) = \text{Cost}(0) + f(0) + f(1) + \dots + f(n) = \text{Cost}(0) + \sum_{i=0}^n f(i)$$

Comme  $\text{Cost}(0)$  est constant on peut généralement l'ignorer.

En particulier (coûts polynomiaux)

$$\sum_{i=1}^n k = k \times n = \mathcal{O}(n)$$

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$$

$$\text{et de manière plus générale } \sum_{i=1}^n i^p = \mathcal{O}(n^{p+1}).$$

## Quelques équations récurrentes classiques (2)

Plusieurs ( $a$ ) appels récursifs

Si  $Cout(n) = a \times Cout(n - 1)$  (avec  $a > 1$ )  
alors  $Cout(n) = a^n \times Cout(0) = \mathcal{O}(a^n)$  (coût exponentiel)

La complexité est aussi exponentielle dès qu'on fait plusieurs appels récursifs sur des données de tailles  $n - 1$ ,  $n - 2$ , etc.

## Cas particuliers

### Calcul de coût direct

Il est parfois possible de donner directement le coût.

Ex : l'algorithme d'écriture binaire effectue clairement  $\lfloor \log_2(n) \rfloor$  divisions.

### Chaque appel compte

```
A(n)
si n = 0
  └ renvoyer 1
sinon
  └ renvoyer A(n-1) + A(n-1) + 1
```

- ▶ mathématiquement équivalents
- ▶ algorithmiquement très différents

```
B(n)
si n = 0
  └ renvoyer 1
sinon
  └ renvoyer 2 × B(n-1) + 1
```

## Cas particuliers (2)

### PGCD

**SOUSTRCTIONS** ( $a, b$ )

**Données** : Deux entiers  $a$  et  $b$  non nuls

**Résultat** : Le plus grand diviseur commun à  $a$  et  $b$

**si**  $a = b$

└ renvoyer  $a$

**sinon si**  $a > b$

└ renvoyer *SOUSTRCTIONS* ( $a - b, b$ )

**sinon**

└ renvoyer *SOUSTRCTIONS* ( $b, a$ )

« Décroissance » des données en un sens à préciser

# Motivation

## Recherche séquentielle en table

RECH\_SEQ( $e, T, n$ )

**Données** : l'élément  $e$  à rechercher, un tableau  $T$  de taille  $n$

**Résultat** : le premier indice  $i$  tel que  $T[i] = e$ , ou  $-1$  s'il est absent

$i = 0$

**tant que**  $i < n$  et puis  $T[i] \neq e$

└  $i = i + 1$

**si**  $i < n$

└ renvoyer  $i$

**sinon**

└ renvoyer  $-1$

## Complexité au pire

- ▶ la boucle `while` ne peut pas faire plus de  $n$  comparaisons
- ▶ pire cas facile à exhiber :  $e$  absent de  $T$  ou en dernière position

D'où : la complexité de RECH\_SEQ est en  $\mathcal{O}(n)$ .

# Plus finement

- ▶ Pour tous les autres cas on fait **strictement** moins de  $n$  comparaisons !
- ▶ Ce genre d'algorithme est utilisé de façon répétée :  
le coût d'une exécution « au pire » n'est pas très informatif.

Idée : moyenne de coût de plusieurs exécutions

On « lisse » les écarts de coûts en prenant en compte les différentes instances possibles :

$$C^{moy}(n) = \text{moyenne}(\text{coût}(d)) \quad \text{pour toutes les données } d \text{ de taille } n$$

Mais **cette moyenne est-elle représentative ?**

Et **comment faire une moyenne sur tous les tableaux possibles ?**

# Calcul de coût moyen

## Présumé indispensable

Modèle probabiliste de répartition des données

i.e. pour une taille  $n$  fixée, probabilité pour chacune des données  $d$  possibles qu'elle soit donnée en entrée à l'algorithme

Il doit correspondre à une réalité d'utilisation :

- ▶ recherche dans un dictionnaire : la plupart du temps le mot existe
- ▶ accès mémoire : l'élément recherché est souvent au début (cache)
- ▶ les algorithmes de tri reçoivent souvent des données presque triées
- ▶ ...

Formule générale : moyenne **pondérée**

$$C^{moy}(n) = \sum_{d \text{ de taille } n} (\text{Proba que l'entrée soit } d) \times (\text{coût de l'algo sur } d)$$

## Fin de l'exemple : recherche en table

Deux paramètres sont nécessaires :

- ▶ probabilité  $P(e \in T)$  notée  $p$
- ▶ dans le cas où  $e \in T$ , on supposera équiprobable de le trouver à chacun des indices  $0..n - 1$ .

Listez :

Différents cas	Probabilité	Coût
...	...	...
...	...	...
⋮	⋮	⋮

## Fin de l'exemple : recherche en table

- ▶ probabilité  $P(e \in T)$  notée  $p$
- ▶ si  $e \in T$ , positions équiprobables dans  $T$

$$\begin{aligned}C_{RECH\_SEQ}^{moy}(n) &= (1 - p) \times n + \sum_{i=1}^n \left( p \times \frac{1}{n} \times i \right) \\&= (1 - p)n + \frac{p}{n} \times \sum_{i=1}^n i \\&= (1 - p)n + \frac{p}{n} \frac{n(n+1)}{2} \\&= (1 - p)n + p \frac{n+1}{2}\end{aligned}$$

En particulier si  $p = 1$  on a  $C_{RECH\_SEQ}^{moy}(n) = \frac{n+1}{2}$

et si  $p = 0$  on a  $C_{RECH\_SEQ}^{moy}(n) = n$

## En pratique

- ▶ L'analyse au pire et au mieux donne déjà des informations sur la complexité moyenne :

$$C^{\min}(n) \leq C^{\text{moy}}(n) \leq C^{\max}(n)$$

Mais **attention** en général  $C^{\text{moy}}(n) \neq (C^{\min}(n) + C^{\max}(n))/2$

- ▶ Regrouper les données de même coût :

$$C^{\text{moy}}(n) = \sum_{\text{coûts } c} c \times (\text{probabilité que le coût soit } c)$$

- ▶ Parfois un modèle simplifié s'impose pour que les calculs soient faisables :
  - ▶ minimiser les paramètres
  - ▶ privilégier l'équiprobabilité

# À propos de l'apnée 1 : mesure expérimentale de complexité moyenne

## Contenus

- ▶ Instrumenter les tris pour compter les comparaisons
- ▶ Implémenter le tri par segmentation
- ▶ **Recueillir les données, les présenter et les analyser**

## Modalités

- ▶ S'inscrire en **binôme** avant de commencer
- ▶ Déposer le code et le **compte-rendu**  
(barème consultable dans le module de rendu)
- ▶ **Redéposer** les fichiers dans le module d'évaluation par les pairs
- ▶ Évaluer et commenter les travaux des autres étudiants

# Le tri rapide

- ▶ aussi appelé QuickSort, Tri par segmentation...
- ▶ inventé par Tony Hoare en 1961
- ▶ utilisé dans de nombreuses bibliothèques comme tri « de référence »



C.A.R. Hoare  
(1934-)

## Quelques caractéristiques

- ▶ intrinsèquement récursif
- ▶ en place (pas besoin de tableau auxiliaire)
- ▶ non stable (les éléments « égaux » ne conservent pas leur ordre initial)

# Principe général

D	A	E	C	B	F
---	---	---	---	---	---

1. Choisir un élément : le **pivot**

<b>D</b>	A	E	C	B	F
----------	---	---	---	---	---

2. **Partitionner** le tableau selon ce pivot :

- ▶ éléments **inférieurs** au pivot à gauche
- ▶ éléments **supérieurs** au pivot à droite

A	C	B	<b>D</b>	F	E
---	---	---	----------	---	---

3. **Trier séparément** les deux sous-tableaux ainsi formés

A	B	C	<b>D</b>	E	F
---	---	---	----------	---	---

## Conception récursive

- ▶ Données : le tableau à trier  
et les indices entre lesquels on travaille
- ▶ Sous-problèmes : des *segments* du tableau initial
- ▶ Pas besoin de reconstruction une fois les segments triés : leurs éléments sont déjà bien placés par rapport au pivot
- ▶ Cas de base : tableau de taille 0 ou 1 (déjà trié)

**TriSegmentation** ( $T, g, d$ )

**Donnée-résultat** *Un tableau  $T$  d'éléments comparables*

**Effet de bord**  *$T$  est trié par ordre croissant entre les indices  $g$  et  $d$*

si  $g < d$

$\left[ \begin{array}{l} \text{pivot} := T[g] \\ k := \text{Partition}(T, g, d, \text{pivot}) \\ \text{TriSegmentation}(T, g, k-1) \\ \text{TriSegmentation}(T, k+1, d) \end{array} \right.$

## La procédure **Partition** (version naïve)

**Partition** ( $T, g, d$ )

**Donnée-résultat** *Un tableau  $T$  d'éléments comparables*

**Résultat** : L'indice  $k$  du pivot correctement placé dans le tableau

**Effet de bord**  $T[g..k-1] < pivot \leq T[k+1..d]$

$T_{aux}$  : tableau temporaire indexé de  $g$  à  $d$

$pivot := T[g]$  ;  $j := g$  ;  $k := d$

**pour**  $i := g + 1$  à  $d$

**si**  $T[i] < pivot$

$T_{aux}[j] := T[i]$

$j := j + 1$

**sinon**

$T_{aux}[k] := T[i]$

$k := k - 1$

$T_{aux}[k] := pivot$

Recopier  $T_{aux}[g..d]$  dans  $T[g..d]$

**renvoyer**  $k$

## Éléments de complexité

- ▶ La taille des données est ici  $d - g$  (taille du *segment* manipulé)
- ▶ **Partition** a un coût linéaire en  $d - g$
- ▶ pas de calcul coûteux dans les paramètres

$$\begin{aligned}
 \text{▶ D'où : } C_{\text{TriSegmentation}}(d - g) &= C_{\text{Partition}}(d - g) \\
 &\quad + C_{\text{TriSegmentation}}(k - 1 - g) \\
 &\quad + C_{\text{TriSegmentation}}(d - k - 1) \\
 &= \mathcal{O}(d - g) \\
 &\quad + C_{\text{TriSegmentation}}(k - 1 - g) \\
 &\quad + C_{\text{TriSegmentation}}(d - k - 1)
 \end{aligned}$$

... mais  $k$  est donné par **Partition**, donc dépend de la donnée.

- ▶ Intuitivement : l'algorithme est efficace (resp. inefficace) quand le pivot est un élément médian (resp. extrême).
- ▶ La suite plus tard...

# En résumé

## Aujourd'hui

- ▶ Un algorithme **récurif** profite de la présence de **sous-problèmes** pour résoudre un problème
- ▶ La **complexité en moyenne** permet d'évaluer le comportement d'un algorithme dans une situation réaliste décrite par une **distribution aléatoire**
- ▶ L'algorithme de **tri rapide** est récursif, peu performant au pire mais efficace en général

## La prochaine fois

- ▶ Type abstrait
- ▶ Pile, file, file à priorité
- ▶ Expression bien parenthésée

# Bonus

## Fonction 91 de McCarthy

MC91 ( $n$ )

**si**  $n \leq 100$

└ **renvoyer**  $MC91(MC91(n + 11))$

**sinon**

└ **renvoyer**  $n - 10$