LINFO1103 : Introduction à L'Algorithmique

Synthèse

Professeur : Pierre Dupont

Année 2021-2022

Cours 1 : Introduction à l'Algorithmique

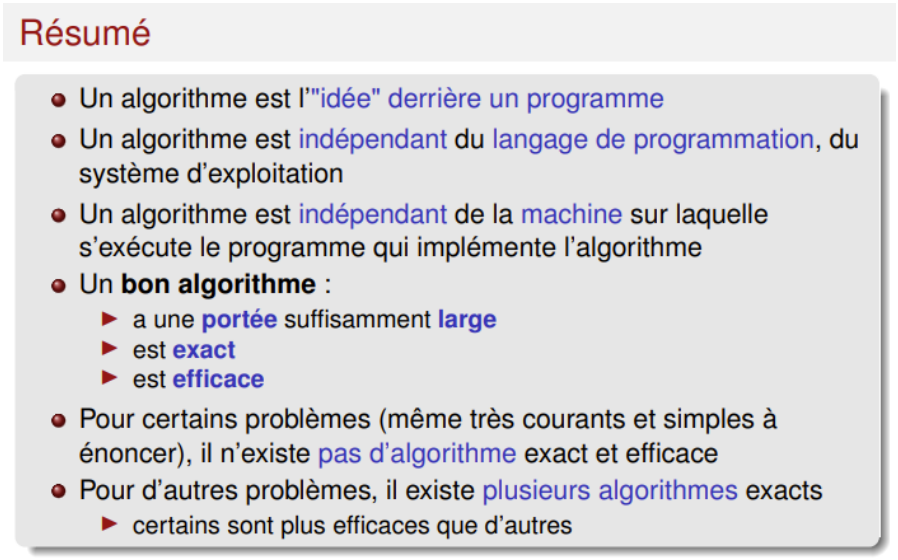
Algorithme : « idée » derrière un programme, permettant de résoudre un problème quelconque, et n'est pas influencé par son écriture (langage utilisé) ou l'environnement dans lequel il est exécuté (bon pc ou non)

Nous cherchons des algorithmes corrects et efficaces

2 qualités d'un algorithme :

1. **Exactitude** : doit fonctionner pour sur toutes les instances du problème (+ cas limites). Si on trouve un contre-exemple, cela veut dire que l'algorithme est inexact mais le contraire n'est pas d'office vrai !
2. **Efficacité** : L'algorithme est efficace s'il est rapide et peut résoudre des instances de grande taille. Il vaut mieux avoir un algorithme efficace sur un pc lent qu'un algorithme non efficace sur un pc rapide

⇨ Essayer de trouver un *équilibre entre exactitude et efficacité*



Cours 2 : Complexité Calculatoire

Comment faire pour caractériser l'efficacité d'un programme ? 2 choses :

1. Le temps calcul que le programme met à produire un résultat ➔ complexité TEMPORELLE
2. L'espace utilisé par le programme ➔ complexité SPATIALE

Données influençant le temps d'exécution :

- Les *données du problème* (la taille n + les valeurs spécifiques)
- *L'algorithme utilisé* pour résoudre le problème
- Mais aussi : *le matériel*, *le logiciel*, *la charge de la machine*, *la charge du réseau*

Taille : nombre de valeurs à spécifier pour définir une instance particulière du problème

Il y a souvent un nombre infini d'instances possibles. Selon l'instance particulière considérée, un algorithme peut prendre plus ou moins de temps. On peut les classer alors en 3 cas :

Nom	Lettrage	Prononciation
Meilleurs Cas	Ω(n)	Omega de n
Cas Moyen	Θ(n)	Theta de n
Pire Cas	O(n)	O de n

Remarques :

a) **Nous nous intéressons le plus souvent au pire cas car nous voulons une garantie sur le temps maximum d'exécution, le meilleur cas donne une estimation optimiste et le cas moyen est difficile à définir.**
b) **Autant être le plus pessimiste possible pour voir si dans le pire des cas, combien de temps l'algorithme mettrait à s'implémenter ➔ on regarde le pire cas O(n)**
c) **Si le pire cas == meilleur cas, on utilise le cas moyen**

Mesure expérimentale : processus qui consiste à écrire un programme, puis l'exécuter pour différentes instances du problème en changeant la taille du problème et mesurer le temps d'exécution ➔ long et chiant à faire

Analyse Asymptotique : analyser le temps ou l'espace en se concentrant sur l'algorithme et l'influence de la taille du problème, généralement dans le pire cas. *L'Analyse Asymptotique s'intéresse à l'évolution de la complexité lorsque la taille du problème augmente*

Donc en fait

- Complexité temporelle = **Analyse Asymptotique** du nombre d'opérations effectuées
- Complexité spatiale = **Analyse Asymptotique** de l'espace utilisé

Constante : tout ce qui ne dépend pas de la taille du problème, même si elle peut varier

Donc si on s'intéresse à l'influence de la taille du problème sur le temps calcul, on peut négliger les constantes

Opération Primitive : instruction en langage de haut niveau (ex : Python) qui représente un nombre constant d'opérations élémentaires. Puisqu'on néglige les constantes, on compte seulement les opérations primitives au lieu des opérations élémentaires

Opération primitive = 1 nanoseconde

Opérations primitives : exemples

- une affectation d'une valeur à une variable : `x = 10`
- une comparaison de deux nombres : `x < y`
- un branchement :
```
if ...:
    ...
else:
    ...
```
- une opération arithmétique élémentaire : `i + 2`
- un accès à un élément d'un tableau (ou d'une `list` en Python) : `A[i]`
- une instruction `return` dans une méthode
- une instruction d'appel à la méthode : `p.MaMethode()`
  ($\neq$ l'exécution de l'ensemble de `MaMethode()` !)

Exemple :

Calcul du nombre d'opérations primitives

Algorithm `arrayMax`
Input: A un tableau de n entiers ($n > 0$)
Output: la valeur maximale dans A

```
currentMax ← A[0]                                  // 2 opérations
for i ← 1 to n − 1 do                              // 1 + 2.(n-1) + 2.n opér.
    if currentMax < A[i] then                      // 3 opér., exécutées (n-1) fois
        currentMax ← A[i]                          // 2 opér., exécutées au plus (n-1) fois
return currentMax                                  // 1 opération
```

Notes : i++ $\equiv i \leftarrow i+1 \Rightarrow$ 2 opérations primitives ; $i \leq n-1 \Rightarrow$2 opérations primitives

Dans le **pire cas** (p.ex. A en ordre croissant), 9n-3 opérations primitives
Dans le **meilleur cas** (p.ex. A en ordre décroissant), 7n-1 opérations primitives

Ce calcul introduit de nouvelles constantes (p.ex. 9 ou 7) que l'on peut négliger pour les mêmes raisons que précédemment !

currentMax = A[0]	2 opérations car 1 affectation + un accès à un élément du tableau ➔ **2**
For i <- 1 to n – 1 do	1) on initialise i == 1 2) i + 1 3) assigner la nouvelle valeur à i 4) calculer n-1 5) vérifier que i < n-1 Les points 2 et 3 on va les faire (n-1) fois car la boucle s'arrête en n-1 ==> 2(n-1). Les points 4 et 5 on va les faire n fois car ce sera n-1 fois VRAI et une dernière fois FAUX = 2n Si tu additionnes tout tu as ➔ **1 + 2(n-1) + 2n**
If currentMax < A[i] then	C'est 3(n-1) opérations (accéder à un élément du tableau + comparaison des 2 valeurs + branchement if) tout ça multiplié par (n-1) parce qu'il a lieu pour chaque passage dans la boucle ➔ **3(n-1)**
currentMax <- A[i]	2(n-1) (accéder au tableau + affectation nouvelle valeur) dans pire cas on va le faire (n-1) fois mais ça dépend de si on rentre dans le branchement if ou pas (donc pire cas on rentre à chaque fois dans le branchement alors 2*(n-1), meilleur cas on rentre qu'une seule fois alors 2) ➔ **Soit 2(n-1) soit 2**
Return currentMax	1 opération élémentaire car 1 *return* ➔ **1**

Pire cas : 2 + 1 + 2(n-1) + 2n + 3(n-1) + 2(n-1) + 1 = 9n-3

Meilleur cas : 2 + 1 + 2(n-1) + 2n + 3(n-1) + 2 + 1 = 7n-1

Comment trouver la complexité d'un algorithme ?

Pour 2 fonctions f(x) et g(x), on dit que (quand x tends vers l'infini) :

1) **f(x) appartient à O(g(x)) si g(x) croit au moins aussi vite que f(x) ➔ f(x) <= g(x)**

2) **f(x) appartient à Ω(g(x)) si g(x) croit au maximum aussi vite que f(x) ➔ f(x) >= g(x)**

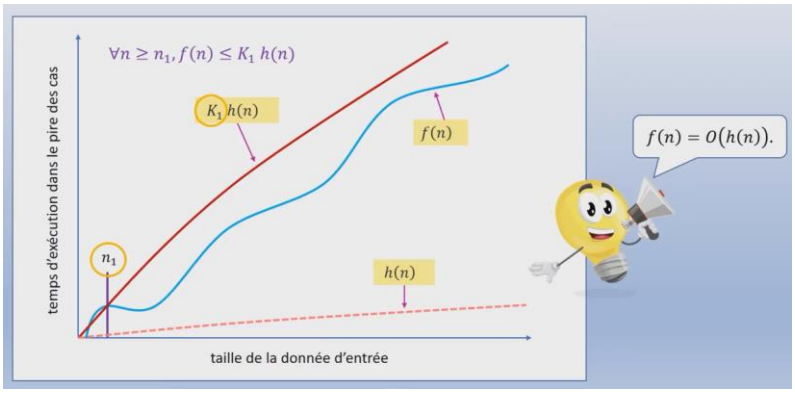
3) **f(x) appartient à Θ(g(x)) si g(x) croit aussi vite que f(x) ➔ f(x) = g(x)**

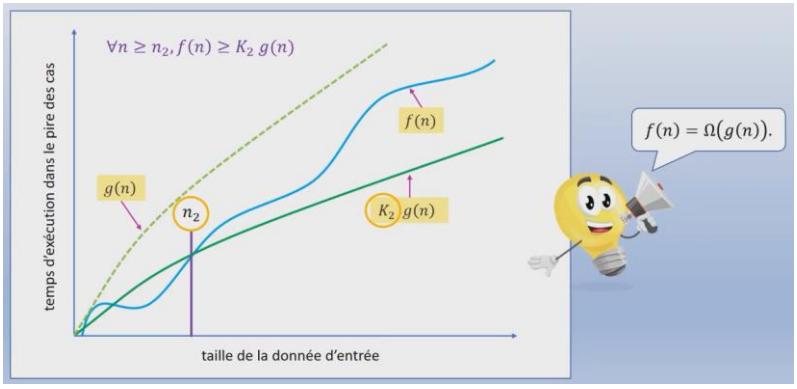
Un exemple simple serait x^2 appartient $O(x^3)$, parce que x^3 croit plus vite que x^2

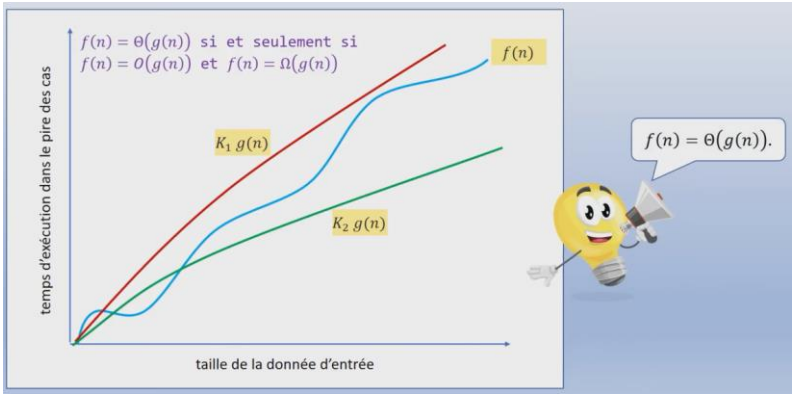
Pour comparer la vitesse de croissance de 2 fonctions, il faut calculer la limite quand x tend vers l'infini de f(x)/g(x), que je vais appeler L.

Si L = infini, alors f(x) croit plus vite, si L = 0, alors g(x) croit plus vite et si L = une constante différente de 0, alors f(x) et g(x) croient aussi vite.

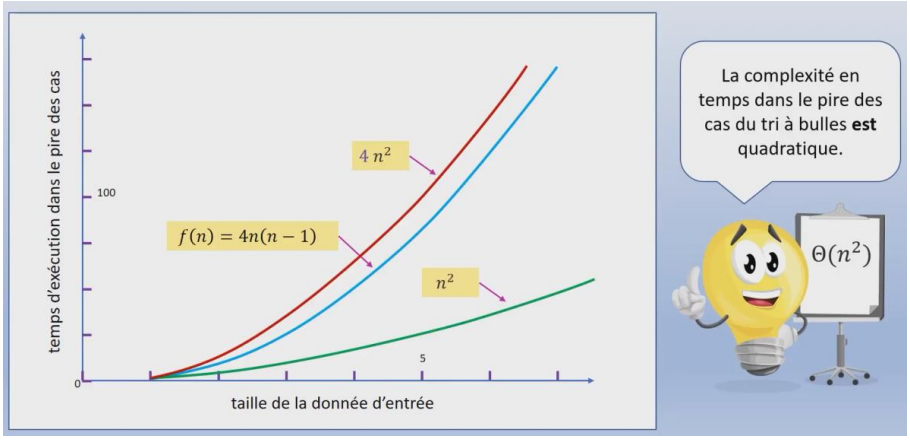
Revenons à notre exemple : x^2/x^3 peut se simplifier et donne 1/x qui donne évidemment 0 quand x tend vers l'infini







Exemple :



Relations de dominance

A retenir

$$n! \gg 3^n \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \sqrt{n} \gg \log n \gg 1$$

- $n!$ croît (finalement) plus vite que toute exponentielle
- a^n croît plus vite qu'une autre exponentielle b^n de plus petite base ($1 \leq b < a$)
- Toute exponentielle (de base > 1) croît plus vite que toute puissance
- n^a croît plus vite qu'une autre puissance n^b de degré moindre ($b < a$)
- Le logarithme croît moins vite que toute puissance positive, même fractionnaire ($\log n \ll \sqrt{n} = n^{\frac{1}{2}}$)
- Le logarithme croît néanmoins et finira donc par dépasser n'importe quelle constante ($\log n \gg 1$)

Résumé

- La **complexité temporelle** d'un algorithme mesure comment évolue le **temps calcul** du programme associé
  - lorsque la taille du problème augmente
  - en négligeant des constantes multiplicatives et des termes non-dominants
- Ces complexités s'exprime en $\mathcal{O}$ (borne supérieure), Ω (borne inférieure) ou Θ (borne exacte)
- La complexité dans le pire cas **peut être différente** de la complexité dans le meilleur cas
  - Exemple : $\Theta(1)$ meilleur cas, $\Theta(n^2)$ pire cas $\Rightarrow$ globalement $\mathcal{O}(n^2)$
- La complexité dans le pire cas est **parfois égale** à la complexité dans le meilleur cas
  - Exemple : $\Theta(n)$ meilleur cas, $\Theta(n)$ pire cas $\Rightarrow$ globalement $\Theta(n)$

Cours 3 : Algorithme de recherche dans les tableaux

Pour trouver la complexité spatiale (espace utilisé par le programme), nous utilisons le même principe que pour la complexité temporelle.

Il est possible que 2 algorithmes résolvant le même problème aient une complexité temporelle différente, et c'est possible également que l'algorithme ayant la plus grande complexité soit plus rapide au final. Cela dépend :

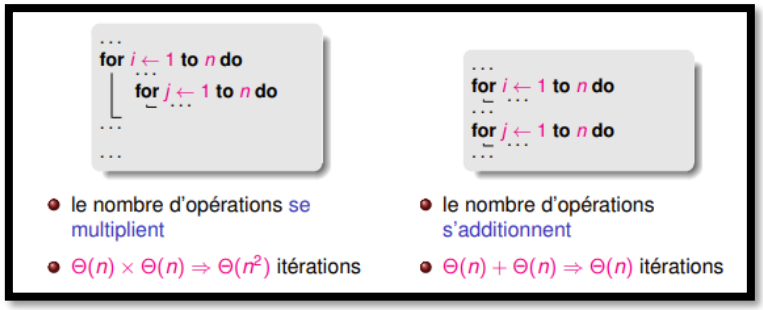
1. Du Programmeur, du code qu'il écrit
2. D'autres facteurs comme la machine, la charge du réseau, le système d'exploitation

On peut se demander alors *quel est l'intérêt d'implémenter un algorithme avec une plus petite complexité ?*

Car l'algorithme ayant la plus petite complexité finira toujours par être meilleur lorsque la taille du problème augmente ! Donc cela peut être possible que l'algorithme ayant une complexité plus grande soit plus rapide que l'autre algorithme, mais pour certaines données

Supposons que l'algorithme *Magnifique* et l'algorithme *Splendide* sont deux manières différentes de résoudre le même problème. Si je prouve que la complexité temporelle de l'algorithme *Magnifique* est, dans **tous** les cas, en $\Theta(n)$ alors que celle de l'algorithme *Splendide* est, dans **tous** les cas, en $\Theta(n^2)$, est-il possible qu'un programme Python implémentant *Splendide* s'exécute plus vite qu'un programme Python implémentant *Magnifique* ?

- (◉) Oui, mais pour certaines données seulement
- ( ) Oui, et pour n'importe quelles données
- ( ) Non, ce n'est jamais possible



La Recherche Dichotomique

La recherche dichotomique est une méthode **récursive**. Cet algorithme sert à **retrouver un** élément. On procède de cette manière :

- On regarde si l'élément au milieu est l'élément : Si oui, on arrête, sinon, on continue de chercher mais en bornant notre recherche [borne inférieure ; milieu-1] si dans partie inférieure ou [milieu + 1 ; borne supérieure] si dans partie supérieure.
- On continue récursivement de cette manière.
- On arrête lorsque l'on a trouvé l'élément ou que l'intervalle entre les 2 bornes devient négatif : *A[622] et A[621]*
- La complexité temporelle dépend du nombre de récursions
- Dans le pire cas, la récursion s'arrête lorsque la taille de l'intervalle est réduite à 1
- **Complexité temporelle : min. Ω(1) ou Max. O(log(n))**

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

- Temps constant $\mathcal{O}(1) = \Theta(1)$
  - addition de deux nombres
  - `print("Hello, World!")`
  - $f(n) = \min(n, 100)$
- Fonction logarithmique $\mathcal{O}(\log(n))$
  - recherche dichotomique
  - problème divisé par 2 (ou 3 ou 4 ou ...) à chaque étape
- Fonction linéaire $\mathcal{O}(n)$
  - énumération de chaque élément d'une collection de taille n
  - nombre constant d'opérations par élément
- Fonction super-linéaire $\mathcal{O}(n \log n)$
  - algorithme de tri par fusion
- Fonction quadratique $\mathcal{O}(n^2)$
  - énumération des paires d'éléments d'une collection de taille n
  - 2 boucles imbriquées, chacune $\mathcal{O}(n)$ itérations
- Fonction cubique $\mathcal{O}(n^3)$
  - énumération des triplets d'éléments d'une collection de taille n
  - 3 boucles imbriquées, chacune $\mathcal{O}(n)$ itérations
- Fonction exponentielle $\mathcal{O}(2^n)$
  - énumération des 2^n sous-ensembles d'une collection de taille n

Par exemple, avec $n = 5$ et l'ensemble : $\{10, 20, 30, 40, 50\}$

sous-ensembles :	$\{30\}$	ou	$\{20, 40\}$	ou	$\{10, 20, 50\}$	ou	...
codage :	**00100**	ou	**01010**	ou	**11001**	ou	...

Globalement, $2^5 = 32$ codes possibles sur 5 bits

- Fonction factorielle $\mathcal{O}(n!)$
  - énumération des $n!$ permutations d'une collection de taille n
  - le voyageur de commerce par énumération des tours possibles

Résumé

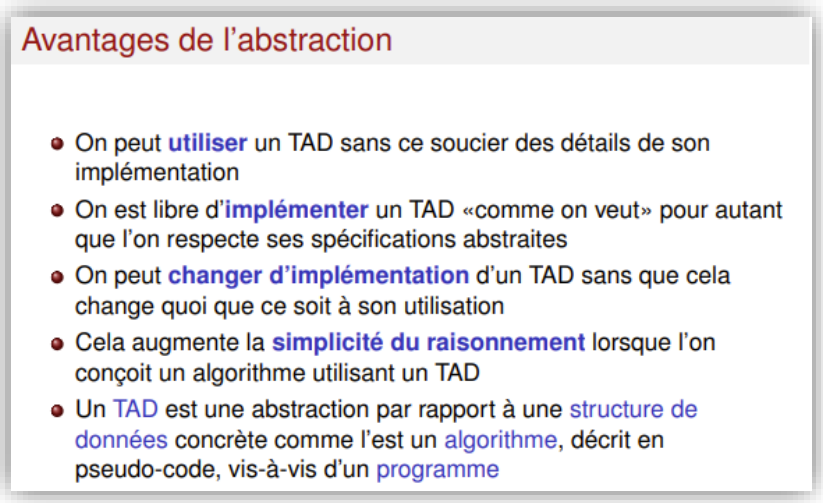
- La **recherche dichotomique** divise le problème en sous-problèmes dont la taille décroît exponentiellement
  - diviser pour régner
  - de façon très efficace : $\mathcal{O}(\log n)$ étapes
- Les **relations de dominance** définissent des classes de complexités d'algorithmes
  - schémas d'algorithmes standards dont la complexité est connue
- L'estimation en pratique de la complexité temporelle requiert un **protocole expérimental rigoureux**
  - lisser l'influence des "constantes"
  - analyser le temps calcul lorsque la taille des instances augmente

Cours 4 : Type Abstrait de Données

Structure de Données : organisation particulière des données d'un programme

Type Abstrait de Données (TAD) : spécification abstraite d'une structure de données. Un TAD décrit ce qui peut être mémorisé, et quelles opérations peuvent être effectuées sur l'information mémorisée

- ⇨ Lorsque l'on conçoit un algorithme, au départ, on reste loin de toute implémentation, la représentation concrète des données n'est pas fixée. On parle alors de Type Abstrait de Données.
- ⇨ Mais *pourquoi alors avoir recours à ces TAD ?* elle permet de définir des types de données non-primitifs, c'est-à-dire non disponibles (non déjà implémentés) dans les langages de programmation courants.



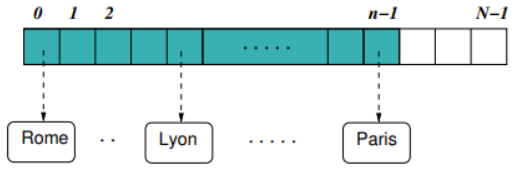
Le travail difficile n'est pas l'implémentation d'un TAD mais plutôt la spécification précise du TAD

Différents TAD :

1) TAD Pile

Pile (*Stack*) : La pile est une collection d'éléments qui peuvent être ajoutés ou retirés selon le principe *LIFO (Last In First Out)* : Le dernier élément arrivé est retiré en premier. On ne peut retirer aucun objet si la pile est vide. *Push* = ajout objet fin, *Pop* = retire objet fin

Implémentation d'une pile par un tableau

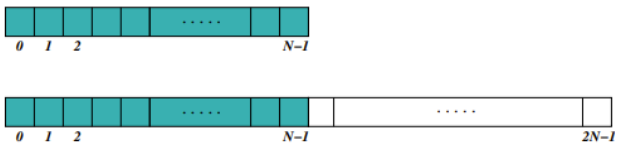


- La pile est mémorisée dans un tableau de capacité N
- Le nombre d'éléments dans la pile vaut n (de 0 à $n-1$)
- Une opération `push` consiste à ajouter un élément à l'indice n et à incrémenter n
- Une opération `pop` consiste à renvoyer l'élément (= une référence) à l'indice $n-1$ et à décrémenter n

Inconvénient

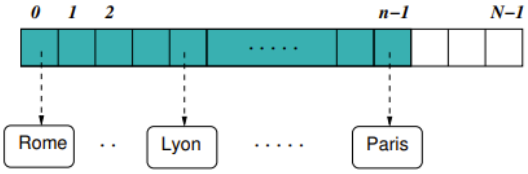
L'opération `push` peut créer un débordement du tableau si n dépasse la **capacité maximale** N

Implémentation par un tableau dynamique



- Lorsqu'une opération `push` crée un débordement, le tableau est automatiquement étendu (p.ex. $N \Rightarrow 2N$)
- On évite le problème d'une pile pleine
- La structure de données `list` en Python est en réalité implémentée en interne comme un tableau dynamique
- En pratique :
  - un nouveau tableau de taille double est créé
  - le contenu du tableau original est recopié dans le nouveau tableau

Complexité calculatoire de l'implémentation

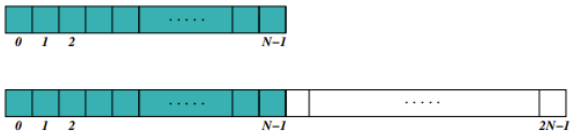


- Toutes les opérations du TAD pile se font en $\Theta(1)$
- Mais la complexité spatiale est en $\Theta(N)$ ($\neq \Theta(n)$)

Remarque

On pourrait aussi effectuer les opérations `push` et `pop` en début de tableau avec l'inconvénient d'une complexité temporelle en $\Theta(n)$

Complexité temporelle avec un tableau dynamique



Complexité temporelle de `push`

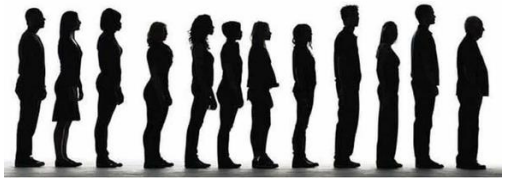
- $\Theta(1)$ si pas de débordement
- $\Theta(N)$ lors d'un débordement (dans ce cas $n = N \Rightarrow \Theta(n)$)
- Sur n `push` consécutifs : $n-1$ fois $\Theta(1)$, 1 fois $\Theta(n)$
- **Complexité amortie** d'un `push` $\frac{1}{n} \times \Theta(n) \Rightarrow \Theta(1)$

Complexité temporelle de `pop`

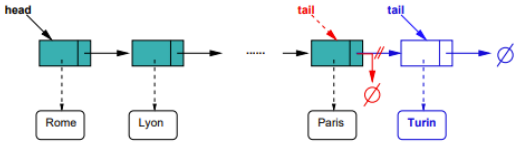
- $\Theta(1)$ dans tous les cas
- On ne réduit pas le tableau après l'avoir agrandi
  - ce serait une très mauvaise idée
  un `push` pourrait recréer un débordement immédiatement après !

2) TAD File

File (Queue) : La file est une collection d'éléments qui peuvent être ajoutés ou retirés selon le principe de *FIFO (First In First Out)* : Le premier élément arrivé est le premier à partir. On ne peut retirer aucun objet si la file est vide. *Enqueue* = insertion en fin de file, *Dequeue* = retrait au début de la file.



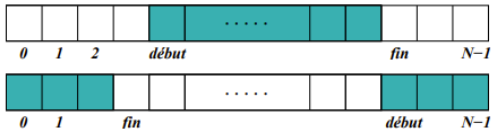
Implémentation par une liste simplement chaînée



- **Possibilité 1**
  - fin de file = fin de liste (**tail**) ⇒ `enqueue` en $\Theta(1)$
  - début de file = début de liste (**head**) ⇒ `dequeue` en $\Theta(1)$
- **Possibilité 2**
  - fin de file = début de liste (**head**) ⇒ `enqueue` en $\Theta(1)$
  - début de file = fin de liste (**tail**) ⇒ `dequeue` en $\Theta(n)$
- Dans tous les cas, complexité spatiale en $\Theta(n)$
- Pas de capacité maximale a priori

Implémentation par un tableau circulaire

2 configurations possibles :



- Gestion du tableau circulaire par de simples additions modulo
  - `fin = (fin + 1) % N`
  - `début = (début + 1) % N`
- Toutes les opérations de la file en $\Theta(1)$
- Complexité spatiale en $\Theta(N)$
- Pas de capacité maximale si le tableau est dynamique

Rappels :

Préconditions (rappels)

- Des conditions **supposées vraies** au moment de l'**appel** à la méthode
- Ces conditions décrivent l'**état initial** avant l'exécution de la méthode
- Ces conditions peuvent *porter sur* les **valeurs des paramètres**, l'**état de l'objet courant** ou toute information pouvant influer sur l'exécution de la méthode
- On s'intéresse aux préconditions les plus faibles, c'est-à-dire aux hypothèses les moins restrictives pour pouvoir exécuter la méthode et qu'elle produise le résultat attendu

Postconditions (rappels)

- Ces conditions **doivent être vérifiées à la fin** de l'exécution de la méthode
  si la méthode a été appelée alors que ses préconditions étaient satisfaites
- Les préconditions + postconditions constituent un **contrat**
- Les postconditions décrivent l'**effet de la méthode** vis-à-vis de l'extérieur de la méthode
- Ces conditions peuvent *porter sur* l'**état de l'objet courant**, une propriété de la **valeur renvoyée** ou toute information modifiée par la méthode
- On s'intéresse aux postconditions les plus fortes, c'est-à-dire les propriétés les plus générales qui sont garanties à la fin de l'exécution de la méthode (si le contrat est respecté !)

Précisions terminologiques

- Une `list` est une structure de données Python qui est implémentée en interne comme un tableau dynamique
- Une liste chaînée est une autre structure de données, basée sur des noeuds qui s'enchaînent par une référence `next` au noeud suivant
  - Une liste chaînée n'est donc **pas** une `list` Python
  - Il existe d'autres structures de données chaînées qui ne sont **pas** des listes chaînées : arbres, graphes, ...
- Une pile (`Stack`) ou file (`Queue`) est un type abstrait de données (TAD) qui peut être implémenté par un tableau, de préférence dynamique, ou par une liste chaînée

En résumé

- Un TAD est une abstraction par rapport à une structure de données
- On peut raisonner sur les propriétés d'un TAD sans ce soucier des détails de son implémentation
  - Une **pile** est une collection d'éléments qui sont ajoutés/retirés selon le principe LIFO
  - Une **file** est une collection d'éléments qui sont ajoutés/retirés selon le principe FIFO
- Un TAD est défini par la spécification de ses méthodes
  - généricité de l'utilisation du TAD indépendamment de son implémentation
  - préconditions les plus faibles possibles
  - postconditions les plus fortes possibles
- Implémentations efficaces d'une pile/file (ajout/retrait en $\Theta(1)$)
  - liste simplement chaînée
  - tableau dynamique (circulaire pour une file)

Cours 5 : Algorithmes de tris

Le problème du tri est l'un des éléments centraux dans la vie de tous les jours, et encore plus dans l'Informatique. Les ordinateurs passent plus de temps à trier que n'importe quelle autre tâche.

3 tris vus dans ce chapitre : Le tri à bulles, le tri par sélection et le tri par insertion

3 Caractéristiques principales :

Tri en place : tri qui trie la liste entrée en input plutôt que de créer une nouvelle liste et effectuer des opérations sur celle-là

Tri Stable : tri qui ne swap pas les clés dont les valeurs sont égales

Tri Adaptatif : tri qui peut moins comparer + swaper en fonction de la liste d'input

A. Le Tri à Bulles

```
Algorithm BubbleSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant
for i ← n − 1 down to 0 do
    for j ← 1 to i do
        if A[j − 1] > A[j] then
            swap(A[j], A[j − 1])
return A
```

Le tri à bulles est une sorte de tri qui fonctionne de cette manière :

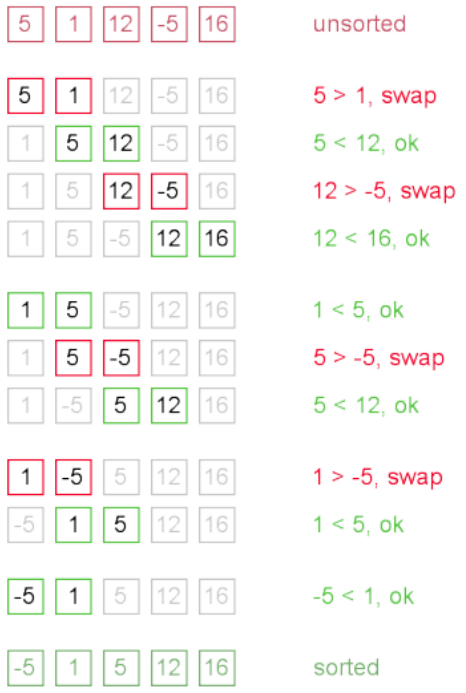
- Inverser les éléments adjacents qui ne sont pas triés
- Un plus grand élément est comme une bulle remontant vers la fin (rapide)
- Un plus petit élément est comme une bulle descendant vers le début (lent)

Complexité :

- Comparaisons : $\Theta(n^2)$ (tous les cas)
- Swaps : $O(n^2)$
- Meilleur cas : 0 swaps
- Pire cas : $\Theta(n^2)$ swaps

Caractéristiques :

Intuitif, non adaptatif, stable, en place mais à éviter car peu efficace



B. Le Tri par Sélection

```
Algorithm SelectionSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant
for i ← 0 to n − 1 do
    min ← i
    // Recherche du min dans le tableau restant
    for j ← i + 1 to n − 1 do
        if A[j] < A[min] then min ← j;
    if min ≠ i then swap(A[i], A[min]);
return A
```

Le tri par sélection est une sorte de tri qui fonctionne simplement comme cela :

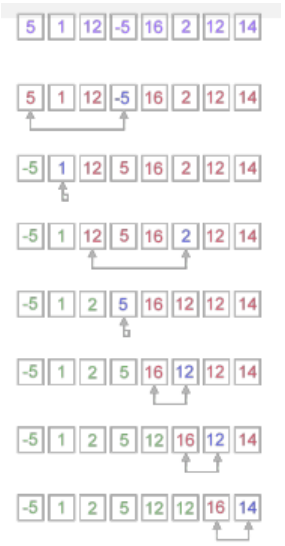
- Mettre le plus petit élément restant dans le devant de la liste jusqu'à que la liste soit triée

Complexité :

- Comparaisons : Θ(n²) (tout les cas)
- Swaps : O(n)
- Meilleur cas : 0 swaps
- Pire cas : Θ(n) swaps

Caractéristiques :

Non adaptatif, non stable, en place, intéressant pour minimiser le nombre d'échanges mais reste peu efficace



C. Le Tri par Insertion

```
Algorithm InsertionSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant
for i ← 1 to n − 1 do
    key ← A[i]
    j ← i − 1
    // Insertion de A[i] dans le tableau trié A[0 . . . i − 1]
    while j ≥ 0 && A[j] > key do
        A[j + 1] ← A[j]          // Déplacement d'une position
        j ← j − 1
    A[j + 1] ← key
return A
```

Le tri par insertion est une sorte de tri assez spéciale : en effet sa technique est peut-être un peu plus complexe ;

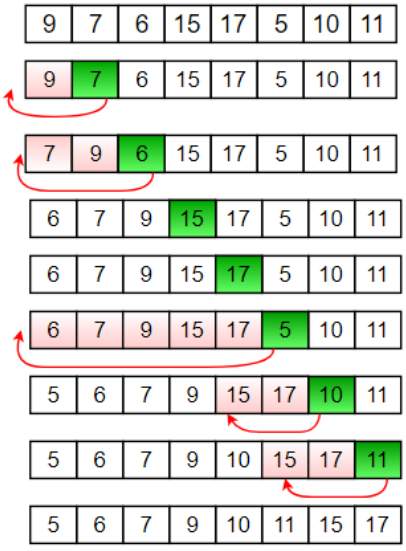
- On prend le second élément de la liste, on le compare avec le premier et on le positionne au bon endroit
- On continue cela avec le second,…

Complexité :

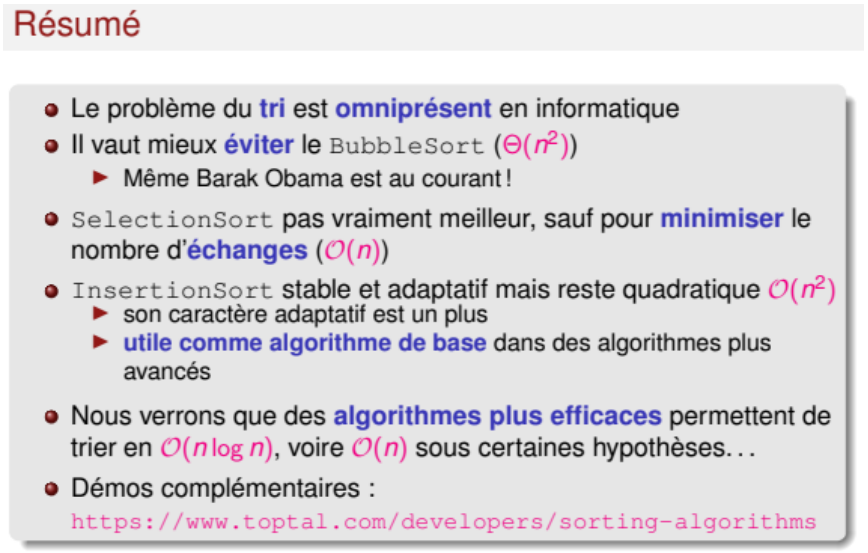
- Comparaisons : O(n²) (tout les cas)
- Swaps : O(n²)
- Meilleur cas : Θ(n) comp + swaps
- Pire cas : Θ(n²) comp + swaps

Caractéristiques :

Adaptatif, stable, en place et intéressant car adaptatif



Nom	Comparaisons	Swaps	Meilleurs / Pire Cas	En Place ?	Stable ?	Adaptatif ?	Conclusion
Tri à Bulles	$\Theta(n^2)$	$O(n^2)$	Meilleur : 0 swaps et Pire : $\Theta(n^2)$ swaps	Oui	Oui	Non	Intuitif mais à éviter car peu efficace
Tri par Sélection	$\Theta(n^2)$	$O(n)$	Meilleur : 0 swaps et Pire : $\Theta(n)$ swaps	Oui	Non	Non	Intéressant pour minimiser le nombre de swaps mais reste peu efficace
Tri par Insertion	$O(n^2)$ (tous les cas)	$O(n^2)$	Meilleur : $\Theta(n)$ Pire : $\Theta(n^2)$ (comp + swaps)	Oui	Oui	Oui	Intéressant car adaptatif
Tri par Fusion	$\Theta(n*\log(n))$ (tous les cas)	/	$\Theta(n*\log(n))$ dans tous les cas	Non	Oui	Non	Très efficace car complexité faible malgré espace additionnel



Cours 6 : Récursion

Fonction Récursive : fonction qui s'appelle elle-même

Avantages :

- Une manière claire de représenter un algorithme
- Un mode de pensée naturel une fois acquis
- En lien direct avec la preuve par induction pour prouver l'exactitude d'un algorithme

Exemples :

- Recherche dichotomique, tri par fusion
- Factorielle, suite de Fibonacci

Premier Exemple : la fonction factorielle

def fact(n) : *if n == 1 :*	**1. Définir un ou plusieurs cas de base**

return 1 *return n*fact(n-1)*	2. **Construire un sous appel récursif sur un ou des sous-problèmes** 3. **Combiner correctement le résultat des appels récursifs**

L'analyse rigoureuse de la complexité temporelle d'une méthode récursive nécessite la résolution d'un système d'équations de récurrence

1) On détermine un cas de base
2) Tant que le cas de base n'est pas respecté, on appelle la fonction

Erreurs souvent rencontrées :

1) *Oublier le ou les cas de base*
2) *Mal positionner le ou les cas de bases*
3) *Appel récursif mais sans return pour récupérer le résultat*
4) *Appel récursif sans réduction à un sous-problèmes*

Résumé

- La récursion offre une manière de programmer compacte et claire
- La récursion nécessite :
  - un ou plusieurs **cas de base**
  - un ou plusieurs appels récursifs sur des **sous-problèmes**
  - de **récupérer** correctement la **valeur renvoyée** par les appels récursifs
- La récursion terminale offre un gain en complexité spatiale, lorsque le compilateur/interpréteur en tire parti

Cours 7 : Récursion & Induction

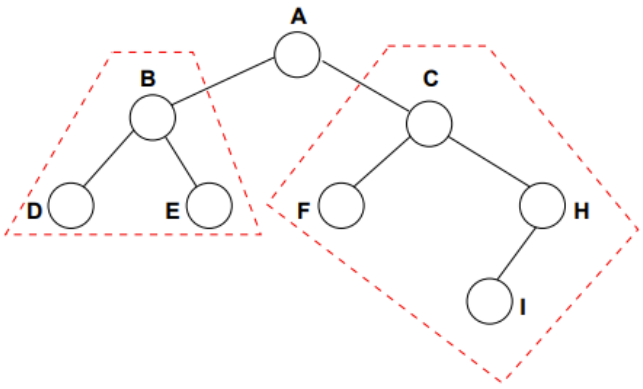
Types abstrait de données définis récursivement :

Liste : soit vide [], soit la concaténation d'une tête de liste (*head*), constituée d'un élément, et d'une fin de liste (*tail*), qui est une liste

⇨ Collection d'éléments auxquels on accède via les méthodes *head()* et *tail()*. Attention, ici on parle de la classe *List* vue en TP, à ne pas confondre avec Liste Python !

Ensemble : soit vide {}, soit l'union d'un singleton et d'un ensemble, tous les éléments étant distincts (pas plus d'infos dans les slides)

Arbre Binaire : soit vide, soit constitué d'un nœud racine, et d'un fils gauche et d'un fils droit qui sont les arbres binaires



A étant le nœud racine, B et C les 2 fils et donc également des arbres binaires

Précisions terminologiques

- Une Liste est un TAD qui décrit une collection d'éléments auxquels on accède via les méthodes `head()` (le premier élément) et `tail()` (une sous-liste)
- Nous avons implémenté ce TAD en Python via une `class List`
- Une liste chaînée est une structure de données, basée sur des noeuds qui s'enchaînent par une référence `next` au noeud suivant
- Une `list` est une structure de données prédéfinie en Python qui est implémentée en interne comme un tableau dynamique

Preuve par Induction :

Prouver l'exactitude d'un algorithme, en particulier récursif, plus généralement, prouver l'exactitude d'un énoncé (comme Analyse Q1)

Le raisonnement est simple : On définit un cas de base qui est vrai (n=0 ou n=1), puis ensuite on calcule le cas où n-1 (ou n+1), et si ce cas est bon aussi, alors l'énoncé est vrai pour n

➔ On a tendance à dire n-1 pour le second cas car on associe ce raisonnement à la récursion

Récursion versus Induction

Récursion

```
def fact(n):
    if n == 0:
        return 1
    return n*fact(n-1)
```

Preuve de l'exactitude par induction

1. Cas de base : `fact(0) = 1 = 0!`
2. Induction :
   si `fact(n-1)` est correct
   alors `fact(n)` est correct
   `fact(n) = n * fact(n-1)`
   $\overset{Hyp.}{=} n * (n-1)! = n!$
3. Conclusion :
   `fact(n)` calcule correctement $n!$ pour tout $n \geq 0$

- L'appel récursif est une **réduction** à un sous-problème
- Le raisonnement inductif est une **généralisation** à un problème plus large
  - Cas de base pour une (ou plusieurs) valeur(s) n_0 (souvent 0 ou 1)
  - Généralisation pour tout $n \geq n_0$

Résumé

- La récursion fournit un moyen de découvrir une solution à un problème combinatoire que l'on ne sait pas résoudre complètement *a priori*
  - il "suffit" d'énoncer une décomposition en sous-problèmes
- La récursion s'applique aussi aux types abstraits de données comme des listes, ensembles, arbres, . . .
- La programmation récursive fait souvent appel à une ou plusieurs méthodes auxiliaires ou à un résultat auxiliaire
  - pour lancer la récursion avec un paramètre supplémentaire servant de résultat temporaire
  - pour récupérer la valeur de ce résultat à la fin de la récursion
- Le raisonnement par induction est très puissant pour démontrer l'exactitude d'un algorithme ou d'une affirmation
  - pour autant qu'on l'utilise correctement. . .

Cours 8 : Tris Efficaces & Récurrences

D. Le Tri par Fusion

Algorithm `MergeSort`
Input: Une liste L de n éléments
Output: La liste L triée en ordre croissant
if $n \leq 1$ **then return** L; // Cas de base
$L_1 \leftarrow$`MergeSort`$(L_{1\ldots\frac{n}{2}})$ // Tri première moitié
$L_2 \leftarrow$`MergeSort`$(L_{(\frac{n}{2}+1)\ldots n})$ // Tri seconde moitié
$L \leftarrow$`Merge`(L_1, L_2) // Fusion
return L

Le Tri par fusion est le quatrième type de tri que nous voyons :

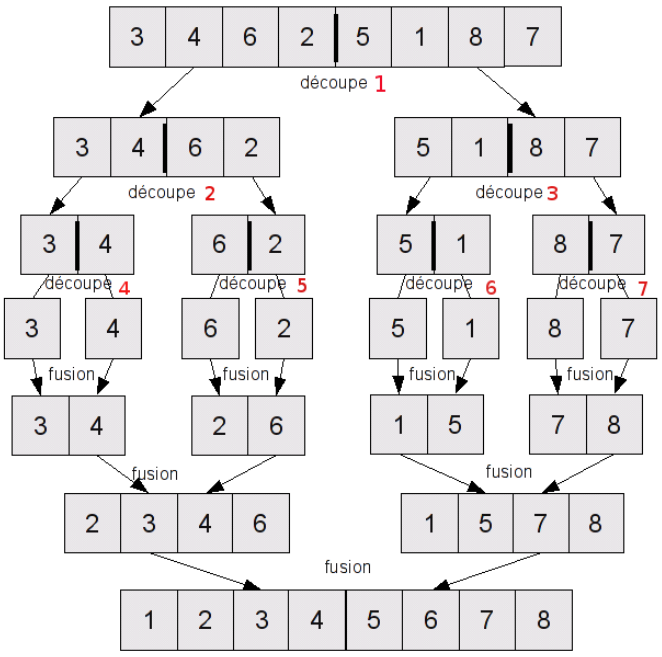
- Séparer en 2 le tableau jusqu'à ne plus qu'avoir plusieurs tableau de 2
- Trier ces tableaux à chaque fois puis les refusionner entre eux pour avoir une liste triée → récursion

Complexité :

- Complexité : Θ(n*log(n)) (tout les cas)

Caractéristiques :

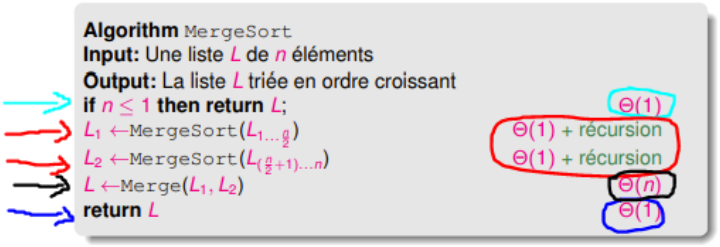
Stable (rare pour tri efficace), récursif, non adaptatif.
Très efficace car complexité faible malgré espace additionnel



Equations de récurrence :

Un système d'équations de récurrence caractérise le temps pris (opérations primitives) pour réduire un problème de taille n

On utilise les équations de récurrence pour analyser la complexité d'algorithmes récursif



$$T(1) = b$$
$$T(n) = c_1 + c_2.n + 2T(\frac{n}{2})$$

b, c_1, c_2 sont des constantes (par rapport à n)

Le tri par fusion a comme équation de récurrence les 2 lignes en bas du cadre.

T(1) = b* provient de *if n<= 1 return L

C1* provient de *return L

C2.n* provient de *L <- Merge(L1,L2)

***2.T(n/2)* provient des 2 *MergeSort* récursifs**

Ensuite, on transforme « récursivement » l'équation 2x fois (une fois en remplaçant *T(n/2)* et l'autre *T(n/4)*) puis on essaie de trouver une formule générale à l'étape *i* en essayant de trouver une forme générale de l'équation

Imax = C'est i tel que tu passes du cas T(n) au cas de base, par exemple T(0)

- Pour T(n) = c1 + ... + T(n/2) et T(1) = b => imax = log2(n)

- Pour T(n) = c1 + ... + T(n - 1) et T(0) = b => imax = n

- Pour T(n) = c1 + ... + T(n - 1) et T(1) = b => imax = n - 1

$T(1) = b$
$T(n) = c_1 + c_2.n + 2T(\frac{n}{2})$

Remplacer $T(\frac{n}{2})$ en utilisant la même équation :
$T(n) = c_1 + c_2.n + 2\left[c_1 + c_2.\frac{n}{2} + 2T(\frac{n}{4})\right]$

Réarranger les termes :
$T(n) = 3c_1 + 2c_2.n + 4T(\frac{n}{4})$

Remplacer $T(\frac{n}{4})$ en utilisant la même équation :
$T(n) = 3c_1 + 2c_2.n + 4\left[c_1 + c_2.\frac{n}{4} + 2T(\frac{n}{8})\right]$

Réarranger les termes :
$T(n) = 7c_1 + 3c_2.n + 8T(\frac{n}{8})$

Trouver une forme générale à l'étape i :
$T(n) = (2^i - 1)c_1 + i.c_2.n + 2^i T(\frac{n}{2^i})$

i représente la profondeur de la récursion, que vaut i_{max} ? $i_{max} = \log_2 n$
$T(n) = (2^{\log_2 n} - 1)c_1 + \log_2 n.c_2.n + 2^{\log_2 n} T(\frac{n}{2^{\log_2 n}})$ et $2^{\log_2 n} = n$

$$T(n) = c_1.(n-1) + c_2.n.\log_2 n + n.T(1) \Rightarrow T(n) \in \Theta(n \log n)$$

➔ **C2*n** c'est que tu as une boucle qui fait n tour dans l'algo

➔ **<=** c'est si tu as une échappatoire, en gros qu'il y a moyen que ton algo avec certains cas se finisse plus rapidement (Du genre un if qui return un truc direct ➔ if autre que cas de base)

➔ La complexité du fait d'accéder à un élément/une partie du tableau est linéaire avec la taille de la partie extraite, donc on ajoute C2*n dans l'équation de récurrence

Taille d'un problème : variable qui influence la complexité, nombre de données entrées en paramètre ou à return (la plupart du temps la taille du problème est de près ou de loin liée à la longueur de quelque chose donné en input)

Pour trouver la profondeur de récursion maximale (*imax*), on doit trouver combien doit valoir *imax* pour arriver au cas de base.

Exemple : cas de base : T(0) = b et équation de récurrence = C1 + T(n-1) ➔ iC1 + T(n-i), on continue de développer pour arriver au cas de base. Ici *imax* vaut n car T(n-n) = T(0), notre cas de base donc *imax* = n (voir ci-dessus en vert des *imax* assez communs)

➔ Tout logarithme croît de la même manière à une constante multiplicative près, quel que soit sa base

Équations de récurrence

Résolution par substitution

1. Énoncer les équations de récurrence en posant $T(n)$ le temps mis pour résoudre un problème de taille n en fonction de constantes (Ex. c_1, c_2), $f(n)$ (Ex. : $n, n^2, \log n, \ldots$), du temps mis pour résoudre un ou plusieurs sous-problèmes (Ex. : $T(\frac{n}{2}), T(n-1), \ldots$)
2. Spécifier la taille et la complexité du (ou des) cas de base (souvent une constante b)
3. Substituer le temps pour les sous-problèmes en utilisant la même équation (mais en adaptant la valeur de n)
4. Répéter la substitution, jusqu'à trouver une formule générale en fonction de i (la profondeur de la récursion)
5. Remplacer i par i_{max} (la profondeur maximale de la récursion) dans la formule générale
6. Laisser tomber les constantes et termes non-dominants pour formuler la complexité en $\mathcal{O}$ ou Θ

Équation de récurrence : exemple

$$T(0) = b$$
$$T(n) = c + 2T(n-1)$$

$$T(n) = c + 2[c + 2T(n-2)]$$

```
def moves(n, left=True):
    if n == 0:
        return
    moves(n-1, not left)
    if left:
        print(n, "left")
    else:
        print(n, "right")
    moves(n-1, not left)
```

Réarranger les termes :
$T(n) = 3c + 4T(n-2)$

Remplacer $T(n-2)$ en utilisant la même équation :
$T(n) = 3c + 4[c + 2T(n-3)]$

Réarranger les termes :
$T(n) = 7c + 8T(n-3)$

Trouver une forme générale à l'étape i :
$T(n) = (2^i - 1).c + 2^i T(n-i)$

i représente la profondeur de la récursion, que vaut i_{max} ? $i_{max} = n$

$$T(n) = (2^n - 1)c + 2^n.T(0) = 2^n.(c+b) - c \Rightarrow$$

$T(n) \in \Theta(2^n)$

Résumé

- Le tri par fusion divise récursivement le problème du tri en sous-problèmes et fusionne les résultats
- Le tri par fusion a une complexité temporelle en $\Theta(n \log n)$ pour trier n éléments
- Les équations de récurrence offrent un outil important pour analyser la complexité d'algorithme récursif
- Plusieurs variantes de la recherche dichotomique ont toutes une complexité temporelle en $\mathcal{O}(\log n)$, où n est la taille du tableau trié
  - la base du logarithme n'importe pas dans l'analyse asymptotique

Cours 9 : Arbres Binaires & Dictionnaires

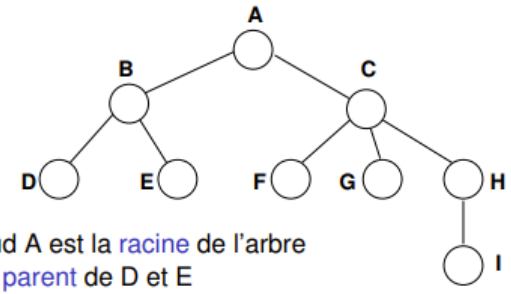
Le TAD Arbre

Le TAD `Arbre` est utile pour représenter des hiérarchies ou structures arborescentes comme, par exemple :

- la table des matières d'un ouvrage
- la structure hiérarchique d'une organisation
- un système de fichiers organisés en répertoires et sous-répertoires
- la structure des classes en Python liées par une relation d'héritage

Le TAD `Arbre` constitue un cas d'école pour l'étude de la récursion

Notions de base sur les arbres



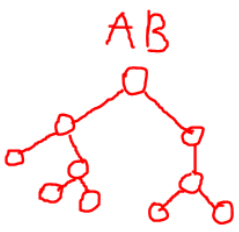
- Le noeud A est la racine de l'arbre
- B est le parent de D et E
- D et E sont les enfants de B
- D et E sont frères
- Les ancêtres de I sont H, C, A
- D, E, F, G, I sont des noeuds externes ou feuilles
- A, B, C, H sont des noeuds internes
- La profondeur de E est 2, la hauteur de l'arbre est 3
- Le degré du noeud C est 3

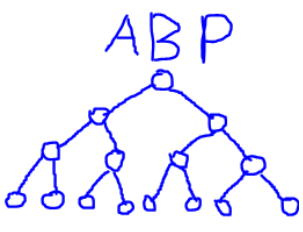
Racine : origine de l'arbre, premier nœud
Frères : nœuds ayant le même parent
Ancêtres : énumération des nœuds parents sur les générations de X
Nœuds externes : les nœuds qui n'ont pas des descendants *(nœuds internes + 1)*
Nœuds internes : tous les nœuds n'étant pas des nœuds externes, donc ayant au moins 1 fils *(n-1/2 où n = nœuds)*
Profondeur : « étage » sur lequel X est situé par rapport au nœud racine qui vaut 0, rajouter +1 à chaque nœud *(nbre de nœuds à profondeur i <= 2^i)*
Hauteur : Profondeur max de l'arbre *(h <= nbre NI)*
Degré : nombres de nœuds connectés à X vers le bas !
N.B : *X : nœud que nous regardons*

Arbre Ordonné : Un arbre est ordonné si les enfants de chaque nœud sont ordonnés. C'est lorsque les enfants sont inférieurs ou égaux aux parents.

Arbre Binaire : Un arbre est binaire s'il est ordonné + si tout nœuds internes est de degré <= 2

Arbre Binaire Propre : Un arbre binaire est propre si tout nœud interne est de degré 2

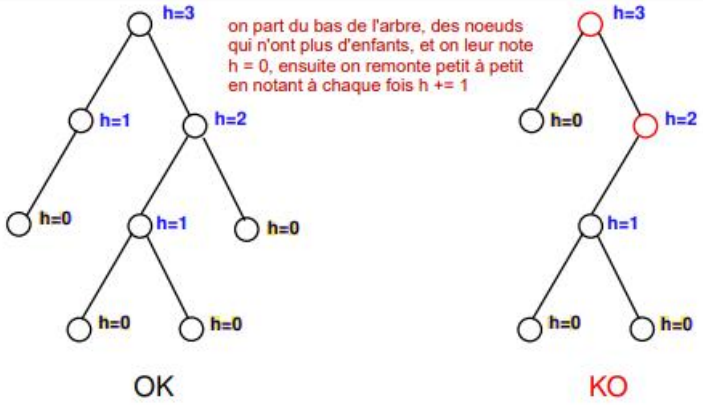




Arbres équilibrés

Définition

Un arbre est équilibré si la différence des hauteurs des sous-arbres gauche et droit de chaque noeud est au plus de 1



Le TAD Dictionnaire

Définition

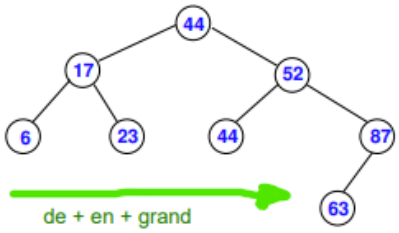
- Un **dictionnaire** est une collection d'entrées pour laquelle on accède à chaque entrée par une clé
- Un dictionnaire contient donc des couples (clé, valeur)
- Le dictionnaire est à clés multiples si plusieurs entrées présentes sont référencées par la même clé
- Le dictionnaire est ordonné si un ordre est défini sur les clés

Exemple : une collection d'étudiants

- identifié par leur NOMA (clé unique)
- la valeur associée est un ensemble d'informations : nom, prénom, année d'études, liste de cours suivis, . . .

Arbre binaire de recherche

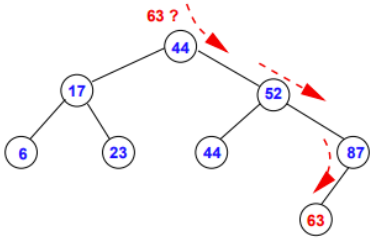
Un **arbre binaire de recherche** (ABR) constitue une implémentation possible d'un dictionnaire ordonné avec des clés éventuellement multiples



Ordre sur les clés : invariant dans un ABR

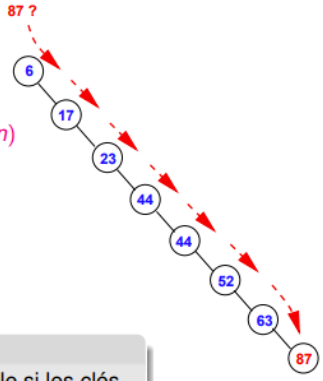
- Les clés mémorisées dans le sous-arbre de gauche d'un noeud v sont $<$ à la clé mémorisée en v
- Les clés mémorisées dans le sous-arbre de droit d'un noeud v sont $\geq$ à la clé mémorisée en v

Recherche d'une clé dans un ABR équilibré



- Complexité temporelle en $\mathcal{O}(h)$, où h est la hauteur de l'arbre
- Si l'arbre est équilibré alors $h \in \Theta(\log_2 n)$, où n est le nombre de noeuds de l'arbre
- Complexité temporelle de la recherche
  - Meilleur cas : $\Theta(1)$ (on trouve la clé à la racine)
  - Pire cas : $\Theta(\log n)$
  - En général : $\mathcal{O}(\log n)$

Recherche d'une clé dans un ABR non équilibré



- Complexité temporelle en $\mathcal{O}(h)$
- L'ABR n'est pas équilibré $\Rightarrow h \in \Theta(n)$
- La complexité temporelle de la recherche
  - Meilleur cas : $\Theta(1)$ (on trouve la clé à la racine)
  - Pire cas : $\Theta(n)$
  - En général : $\mathcal{O}(n)$

Note

Cette configuration apparaît par exemple si les clés sont insérées en ordre croissant

3 Types de Parcours dans un ABR :

1. Parcours Préfixe

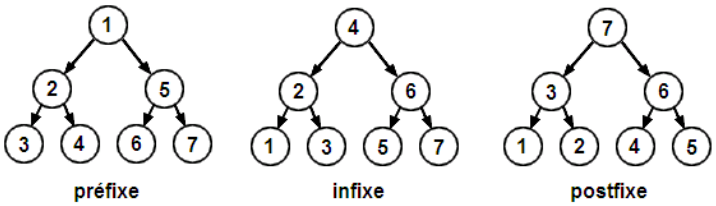
Racine – sous-arbre de gauche – sous-arbre de droite On passe d'abord par la Racine, puis en priorité par les nœuds internes puis les externes

2. Parcours Infixe

sous-arbre de gauche – Racine – sous-arbre de droite On fait simplement un parcours des nœuds de la gauche vers la droite

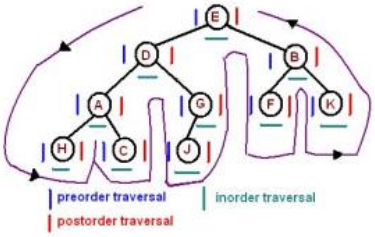
3. Parcours Postfixe

sous-arbre de gauche – sous-arbre de droite – Racine On passe d'abord par le sous-arbre de gauche, en privilégiant les nœuds externes, puis les nœuds internes et enfin la Racine



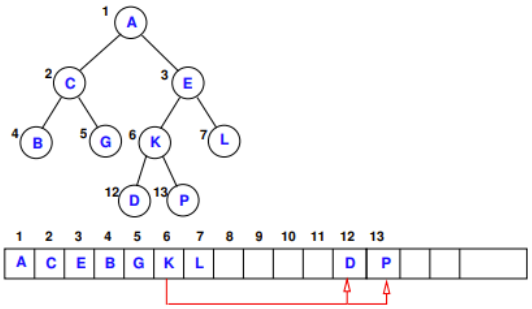
Points communs aux 3 parcours

Pour chacun des parcours, on commence en réalité toujours par la racine et on passe trois fois par chaque noeud **mais** on n'effectue une opération que lors d'un seul passage



- au premier passage (`Left`), pour le parcours préfixe : **EDAHCGJBFK**
- au second passage (`Below`), pour le parcours infixe : **HACDJGEFBK**
- au 3ème passage (`Right`), pour le parcours postfixe : **HCAJGDFKBE**

Implémentation basée sur un tableau (dynamique)



L'information relative à un noeud est mémorisée dans un tableau à un certain rang selon une numérotation par niveaux :

- rang(v) = 1 $\Leftrightarrow$ v est la racine
- rang(v) = 2*rang(u) $\Leftrightarrow$ v est le fils gauche de u
- rang(v) = 2*rang(u)+1 $\Leftrightarrow$ v est le fils droit de u

Complexité calculatoire

Structure chaînée

- Accès à n'importe quel noeud à partir de la racine en $\mathcal{O}(n)$
- Complexité spatiale en $\Theta(n)$

Tableau (dynamique)

- Accès à n'importe quel noeud en $\Theta(1)$ par un calcul d'indice dans le tableau
- La complexité spatiale peut être exponentielle dans le pire cas : $\mathcal{O}(2^n)$, où n est le nombre de noeuds présents dans l'arbre
  - Si l'arbre est strictement complet, c'est-à-dire si tous les niveaux sont «remplis», alors la complexité spatiale est en $\Theta(n)$

Résumé

- Le TAD `Arbre` est très utile pour organiser des données sous la forme de hiérarchies
- Le TAD `Arbre` est récursif
- Un Arbre Binaire de Recherche est une implémentation d'un dictionnaire ordonné
- Les parcours préfixe, infixe, postfixe sont des cas particulier d'un parcours eulérien
- Les arbres (binaires) peuvent être implémentés à l'aide d'une structure chaînée ou d'un tableau (dynamique)

Cours 10 : Invariants et tri

Définition

- Un invariant est une propriété qui ne varie pas (p.ex. toujours vraie)
  - Exemple : chaque domino couvre 2 cases adjacentes qui sont forcément de couleur opposée, **quelque soit** la manière dont sont disposés les dominos sur l'échiquier
- Les invariants sont utiles pour démontrer
  - qu'il y a ou non une solution à un problème
  - qu'un algorithme est exact

Invariant de Boucle : un Invariant de boucle est une propriété qui reste vraie :

- Lors de l'initialisation de la boucle
- Après chaque itération de la boucle
- Après terminaison de la boucle

Exemple 1 :

Tri par insertion

Algorithm `InsertionSort`
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant

```
i ← 1                                              // Initialisation
while i ≤ n − 1 do
    key ← A[i]
    j ← i − 1
    // Insertion de A[i] dans le tableau A[0 . . . i − 1]
    while j ≥ 0 && A[j] > key do
        A[j + 1] ← A[j]        // Déplacement d'une position
        j ← j − 1
    A[j + 1] ← key
    i ← i + 1                                      // Itération
return A                                           // Terminaison
```

Invariant du tri par insertion

Le tableau $A[0 \dots i-1]$ est constitué des éléments initialement aux positions $[0 \dots i-1]$ et est trié par ordre croissant

Preuve d'exactitude du tri par insertion

Invariant du tri par insertion

Le tableau $A[0 \dots i-1]$ est constitué des éléments initialement aux positions $[0 \dots i-1]$ et est trié par ordre croissant

Preuve d'exactitude

1. **Initialisation**
   L'invariant est vrai puisque lorsque $i == 1$ le tableau $A[0 \dots i-1]$ est réduit à $A[0]$ qui est trié
2. **Itération**
   L'invariant reste vrai puisque la boucle **while** interne déplace vers la droite les éléments qui sont plus grands que $A[i]$ et insère la clé, initialement en $A[i]$, à la position correcte
3. **Terminaison**
   La boucle **while** externe s'arrête lorsque $i == n$
   A ce stade, le tableau $A[0 \dots i-1] = A[0 \dots n-1]$ est trié
   L'ensemble du tableau est donc trié et l'algorithme est correct

Exemple 2 :

Exactitude de la factorielle par invariant

Version itérative

```
def factorial(n):
    fact = 1
    for i in range(2, n+1):
        fact *= i
    return fact
```

Preuve de l'exactitude

Invariant : `fact = (i-1)!`

1. Initialisation :
   Lorsque `i==2`, `fact = 1 = (2-1)! = 1!`
2. Itération :
   - A chaque exécution

   `fact` ← `fact * i = (i-1)! * i = i!`
   `i ← i+1` ⇒ `fact = (i-1)!`
3. Terminaison :
   - La boucle s'arrête lorsque `i==(n+1)`
   - A ce stade :

     `fact = (i-1)! = (n + 1 -1)! = n!`
   - L'algorithme est correct

En résumé

- Les invariantes de boucle sont utiles pour démontrer l'exactitude d'algorithmes itératifs
- Les preuves par induction sont utiles pour démontrer l'exactitude d'algorithmes récursifs
- Une preuve par induction généralise pour toute taille n du problème alors que la preuve par invariant considère explicitement la terminaison

2 Tris encore vus à la fin du cours :

1. Tri Spaghetti

Le tri spaghetti

Algorithm `NoodleSort`
Input: Un tableau A de n entiers
Output: Le tableau A trié en ordre croissant

for $i \leftarrow 0$ **to** $n-1$ **do**	// $\Theta(n)$ itérations
Couper un spaghetti (cru !) de longueur $A[i]$ mm	$\Theta(1)$
Disposer les n spaghettis verticalement sur une table	$\Theta(n)$
$i \leftarrow n-1$	$\Theta(1)$
while $i \geq 0$ **do**	// $\Theta(n)$ itérations
Tirer le spaghetti le plus long	$\Theta(1)$
Mesurer sa longueur x	$\Theta(1)$
$A[i] \leftarrow x$	$\Theta(1)$
$i \leftarrow i-1$	$\Theta(1)$
return A	$\Theta(1)$

La complexité temporelle de `NoodleSort` est $\Theta(n)$ dans **tous les cas**
... mais il utilise un **comparateur parallèle** de $\mathcal{O}(n)$ éléments

Le Tri Spaghetti est plutôt simple à comprendre : *pour chaque entier x de notre tableau, prendre un spaghetti de longueur x*. Une fois que l'on a tous nos entiers du tableau représenté par un spaghetti, *on prend tous les spaghetti dans notre poing et on les abaisse à la table*, de sorte qu'elles tiennent toutes debout, reposant sur la surface de la table. Ensuite, *prenez un à un la tige la plus longue restante à chaque fois* et insérez sa longueur x à l'avant du tableau. Répétez l'opération jusqu'à ce qu'il ne reste aucune tiges.

Tri spaghetti — Wikipédia (wikipedia.org)

2. Tri Linéaire

Tri en temps linéaire

Algorithm `LinearSort`
Input: Un tableau A de n entiers (distincts)
Output: Le tableau A trié en ordre croissant

$max \leftarrow$ `FindMax`(A) ; $min \leftarrow$ `FindMin`(A)	$\Theta(n)$
Allouer un tableau B de taille $max - min + 1$	$\Theta(1)$
for $j \leftarrow min$ **to** max **do** $B[j - min] \leftarrow min - 1$;	$\Theta(max - min)$
for $i \leftarrow 0$ **to** $n-1$ **do**	// $\Theta(n)$ itérations
$key \leftarrow A[i]$	$\Theta(1)$
$B[key - min] \leftarrow key$	$\Theta(1)$
$i \leftarrow 0$	$\Theta(1)$
for $j \leftarrow min$ **to** max **do**	// $\Theta(max - min)$ itérations
if $B[j - min] \geq min$ **then** $A[i++] \leftarrow B[j - min]$;	$\Theta(1)$
return A	$\Theta(1)$

La complexité temporelle de `LinearSort` est $\Theta(n + max - min) = \Theta(n)$ dans **tous les cas**

Tri en temps linéaire

Variante pour gérer les clés égales dans A :

- Initialiser $B[.]$ à 0
- Mémoriser dans B le *nombre d'occurrences* de chaque clé de A
- Adapter la recopie dans A

`LinearSort` a une complexité en $\Theta(n)$ **dans tous les cas** mais

- il a une complexité spatiale en $\Theta(max - min)$ qui peut dépasser la mémoire disponible si $max - min$ est très grand
- dans ce cas, l'accès à un indice du tableau B n'a plus lieu en temps constant

Nom	Comparaisons	En Place ?	Stable ?	Adaptatif ?	Conclusion
Tri Spaghetti	Θ(n) (tous les cas)	Oui	Oui	Non	Assez simple à implémenter, utilise aussi un comparateur // en O(n)
Tri Linéaire	Θ(n) (tous les cas)	Oui	Oui	Non	Tri assez complexe niveau implémentation sinon il est efficace

```python
def LinearSort(A):
    """
    Tri Linéaire
    pre: un tableau A de n entiers éléments
    post: le tableau A trié en ordre croissant
    """
    print("tableau d'entrée = ",A)
    max1 = max(A) #On obtient le max de A
    print("max1 = ",max1)
    min1 = min(A) #On obtient le min de A
    print("min1 = ",min1)
    B = [] #On créée un nouveau tableau
    new_len = (max1-min1) + 1
    print("new_len = ",new_len)
    for i in range(new_len): #Permet d'ajouter n éléments dans B
        B.append(i)
    print(B)
    for j in range(min1,max1+1): #Permet de remplir B de 0
        B[j-min1] = min1 - 1
    print(B)
    for i in range(0,len(A)): #Fonction permettant de trier le tableau B
        key = A[i] #key vaut le chiffre positionné à la i-ème place de A
        B[key-min1] = key #C'est ici que tout se fait
        print(B)
    #Si notre tableau comporte que des nombres consécutifs, on peut s'arrêter ici
    i = 0
    for j in range(min1,max1+1):
#Si l'élément itéré est plus grand ou égal à min1, donc != des 0 qu'on a ajouté au début
        if B[j-min1] >= min1:
            A[i] = B[j-min1] #L'élément est repositionné dans A
            i += 1
    return A

A1 = [9,4,2,5,8,3,1]
print(LinearSort(A1))
```

```
tableau d'entrée =  [9, 4, 2, 5, 8, 3, 1]
max1 =  9
min1 =  1
new_len =  9
[0, 1, 2, 3, 4, 5, 6, 7, 8]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 9]
[0, 0, 0, 4, 0, 0, 0, 0, 9]
[0, 2, 0, 4, 0, 0, 0, 0, 9]
[0, 2, 0, 4, 5, 0, 0, 0, 9]
[0, 2, 0, 4, 5, 0, 0, 8, 9]
[0, 2, 3, 4, 5, 0, 0, 8, 9]
[1, 2, 3, 4, 5, 0, 0, 8, 9]
[1, 2, 3, 4, 5, 8, 9]
PS C:\Users\arthu>
```

Le Tri Linéaire est donc un peu compliqué mais dès qu'on le comprend, cela va tout de suite mieux

⇨ **Θ(n) dans tous les cas**

Résumé

- Un invariant est une propriété qui ne varie pas
- Les invariants de boucle sont utiles pour démontrer l'exactitude d'algorithmes itératifs
- Tout algorithme de tri utilisant des comparaisons 2 à 2 a une complexité temporelle qui est en $\Omega(n \log n)$ dans le pire cas
- Un tri en temps linéaire $\Theta(n)$ est possible si
  - on utilise un comparateur parallèle
  - les éléments à trier appartiennent à un domaine limité

Bonne étude bg, tqt tu va réussir ;)