

Algorithmique 1

Sylvain Daudé

HAI101I / HA8203I

- 12h cours, 18h TD, 15h TP
- Evaluation :
 - Un examen intermédiaire en amphi : 25%
 - Une note de TD-TP : 25%
 - Un examen final en amphi : 50% avec règle du max
 - Seconde session complète à 100%

Objet du cours

Ce cours porte sur les algorithmes récurrents et itératifs ainsi que leur efficacité (*complexité en temps*). Deux sections sont dédiées aux algorithmes de recherche et de tri. Une synthèse du cours est disponible en ligne sur Moodle.

Principale référence

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction à l'algorithmique*. Ed. Dunod.

Cet ouvrage de référence est disponible en ligne gratuitement. Il vous sera utile pendant tout votre cursus.

Plan

- 1 Généralités
- 2 Structures linéaires
- 3 Arrêt d'un algorithme
- 4 Validité d'un algorithme
- 5 Complexité en temps d'un algorithme
- 6 Algorithmes de recherche dans un tableau
- 7 Algorithmes de tri

Plan

- 1 Généralités
- 2 Structures linéaires
- 3 Arrêt d'un algorithme
- 4 Validité d'un algorithme
- 5 Complexité en temps d'un algorithme
- 6 Algorithmes de recherche dans un tableau
- 7 Algorithmes de tri

Définition d'un algorithme

Algorithme : séquence de calcul bien définie, en réponse à un problème, selon un nombre fini d'opérations.

- S'il renvoie un résultat principal, c'est une *fonction*, sinon une *procédure*.
- Les effets de l'algorithme autres que le résultat principal (affichages, modification de l'environnement...) sont les *effets de bord*.

Remarques

- Certains problèmes sont bien posés mais trop complexes pour un algorithme ("indécidables" ou "incalculables"). Exemple : problème de l'arrêt.
- Certains algorithmes nécessiteraient trop de ressources pour être concrètement utilisés. Exemple : le jeu d'échecs : 300 millions d'années pour connaître les conséquences de chaque coup.
- Dans ces deux cas, on adopte des méthodes approchées, les *heuristiques*.

Vocabulaire sur un exemple en pseudo-code

Signature ou prototype

Catégorie	Nom	Paramètres	Type du résultat principal
-----------	-----	------------	----------------------------

Fonction	facto	(E n : entier)	entier
-----------------	-------	----------------	--------

Spécifications : renvoie la factorielle d'un entier naturel n

Variables : produit : entier

spécifications : problème, contraintes sur les paramètres, effets de bord

variables : emplacements nommés de stockage

Début

produit ← 1

pour i de 1 à n faire

 produit ← produit * i

finPour

Renvoyer produit

corps

Fin

Définitions

- Algorithme itératif : contient au moins une boucle (pour, tant que).
- Algorithme récursif : partiellement défini à partir de lui-même et d'un cas de base.

Remarque

Un algorithme peut être ni itératif ni récursif, les deux à la fois ou seulement un des deux : tout est possible.

Exemple d'algorithme itératif en pseudo-code

FONCTION : FactIt (E n : entier) : entier

SPÉCIFICATIONS : $n \in \mathbb{N}$. Renvoie $n! = 1 \times 2 \times \dots \times n$ avec $0! = 1$

Variable : res : entier

DÉBUT

res $\leftarrow$ 1

pour i de 1 à n **faire**

res $\leftarrow$ res * i

fin pour;

Renvoyer res

FIN

Exemple d'algorithme récursif en pseudo-code

FONCTION : FactRec (E n : entier) : entier

SPÉCIFICATIONS : $n \in \mathbb{N}$. Renvoie $n! = 1 \times 2 \times \dots \times n$ avec $0! = 1$

DÉBUT

si $n = 0$ **alors**

Renvoyer 1

sinon

Renvoyer $n \times \text{FactRec}(n - 1)$

fin si;

FIN

- **Type** : catégorie d'une valeur : entier, nombre, booléen, tableau...
- **Constante** : valeur invariable ayant un type : 5, 'c', "Peachy".
- **Variable** : triplet (nom, type, valeur). Nom et type définis à la déclaration, valeur affectée dans un deuxième temps (par exemple dans le corps d'un algorithme).
 - **Environnement** : ensemble de variables (notamment). Un environnement peut contenir des sous-environnements.
 - **Portée d'une variable** : ensemble des environnements où elle existe.
- **Appel d'algorithme** : nom de l'algorithme suivi de valeurs pour les paramètres, les **arguments**, entre parenthèses. A pour valeur le résultat principal de l'algorithme si c'est une fonction. **Appel récursif** : appel d'un algorithme par lui-même.
- **Opération** : application d'un opérateur à des opérandes : $3*8$
- **Expression** : formule contenant constantes, noms de variables, paramètres, appels d'algorithmes et opérations. A pour valeur le résultat de la formule.

A vous de jouer !

Trouver les types, constantes, variables, appels, opérations, expressions

FONCTION : PuisIt (E x : nombre,
E n : entier) : nombre

Variable : res : nombre

DÉBUT

res $\leftarrow$ 1

pour i de 1 à n **faire**
 res $\leftarrow$ res * x

fin pour;

Renvoyer res

FIN

FONCTION : PuisRec (E x : nombre,
E n : entier) : nombre

DÉBUT

si $n = 0$ **alors**
 Renvoyer 1

sinon
 Renvoyer
 $x \times \text{PuisRec}(x, n - 1)$

fin si;

FIN

A vous de jouer !

Correction

- types : nombre, entier
- constantes : 0, 1
- variables : res
- appels : PuisRec(n-1,x) (appel récursif)
- opérations : $\text{res} \leftarrow 1$ (affectation), $\text{res} * x$, $\text{res} \leftarrow \text{res} * x$, $n=0$, $n-1$, $x \times \text{PuisRec}(n-1,x)$
- expressions : tous les précédents sauf affectations, x , n (paramètres), i (indice de boucle)

Attention !

L'indice de boucle et les paramètres ne sont *pas* des variables.

Opérateurs en pseudo-code et python

- **Opérateurs de calcul** : + - * / ^ (puissance) div (quotient) mod (reste)
- **Opérateurs de comparaison** : = $\neq$ < $\leq$ > $\geq$
- **Opérateurs logiques** : non, et, ou. Evaluation paresseuse :
 - dans "a et b", si a est faux, b n'est pas évalué
 - dans "a ou b", si a est vrai, b n'est pas évalué.
- **Opérateur ternaire ou conditionnel** : cond(a,b,c) : vaut b si a est vrai, c si a est faux. b et c doivent être du même type. Utilise l'évaluation paresseuse.
- **Affectation** : variable $\leftarrow$ expression. N'a pas de valeur.
- **En Python** : ^ devient **, div //, mod %, ==, $\neq$!=, $\leq$ <=, $\geq$ >=, non not, et and, ou or, cond(a,b,c) b if a else c, $\leftarrow$ =

A vous de jouer !

Les expressions suivantes sont-elles correctes ?
Si oui, quel est leur type et leur valeur ?

- $1+5*2$ entier, 11
- $1+5/2$ nombre, 3.5
- $8 \text{ div } 3 + 7 \text{ mod } 5$ entier, 4
- $5.0 \text{ mod } 2$ erreur
- $1=5*2$ booléen, false
- $\text{true et (true ou false)}$ booléen, true
- $\text{true ou } 3$ erreur
- true ou (5/0=1) booléen, true
- $\text{cond}(5 \text{ mod } 2 = 1, 8, 15)$ entier, 8
- $\text{cond}(5 \text{ div } 2 = 1, \text{true}, 0)$ erreur
- $\text{cond}(5 \text{ div } 2 = 2, \text{cond}(5 \text{ mod } 2 = 0, 3, 8), 11)$ entier, 8

Structure conditionnelle

Définition

Une *structure conditionnelle* modifie le déroulement de l'algorithme selon qu'une condition est vérifiée ou non.

Syntaxe

```
si a alors  
    instructions I  
sinon si b1 alors  
    instructions SS1  
...  
sinon si bn alors  
    instructions SSn  
sinon  
    instructions S  
fin si;
```

En Python :

```
if a :  
    Instructions I  
elif b1 :  
    Instructions SS1  
(...)  
elif bn :  
    Instructions SSn  
else :  
    Instructions S
```

A vous de jouer !

Qu'affichent les blocs d'instructions suivants ?

Variable : i : entier

DÉBUT

$i \leftarrow 0$

si $i < 3$ **alors**

$i \leftarrow 5$

sinon si $i \geq 4$ **alors**

$i \leftarrow 7$

fin si;

afficher i

FIN

Correction : 5

Variable : i : entier

DÉBUT

$i \leftarrow 0$

si $i < 3$ **alors**

$i \leftarrow 5$

fin si;

si $i \geq 4$ **alors**

$i \leftarrow 7$

fin si;

afficher i

FIN

Correction : 7

Boucle tant que

Définition

Une boucle *tant que* répète un bloc d'instructions tant qu'une certaine condition est vraie. On utilise une boucle *tant que* lorsque le nombre d'itérations n'est pas connu à l'avance.

Syntaxe

```
tant que expression faire  
    instructions  
fin tq;
```

En Python :

```
while expression :  
    instructions
```

Remarque

La boucle *Tant que* est plus générale que la boucle Pour.

A vous de jouer !

Que renvoient les blocs d'instruction suivants ?

Variable : i : entier

DÉBUT

i $\leftarrow$ 0

tant que $2 * i + 1 < 10$ **faire**

i $\leftarrow$ i+1

fin tq;

Renvoyer i

FIN

Correction : 5

Variable : i : entier

DÉBUT

i $\leftarrow$ 5

tant que i $\neq$ 0 **faire**

i $\leftarrow$ i-2

fin tq;

Renvoyer i

FIN

Correction : Rien ! (boucle infinie)

Boucle pour

Définition

Une boucle *pour* répète (ou itère) un bloc d'instructions. Les itérations sont repérées par un *indice de boucle*. Il y a autant d'itérations que de valeurs de l'indice. On l'utilise lorsque le nombre d'itérations est connu à l'avance.

Syntaxe

```
pour i de a à b [par pas de c] faire  
    instructions
```

```
fin pour;
```

```
# Par défaut, c=1 ;
```

```
# Pas d'itération si i dépasse strictement la valeur b.
```

En Python :

```
for i in range (a, B, c) :
```

```
    Instructions
```

```
# Par défaut, a=0 et c=1 ;
```

```
# Pas d'itération si i dépasse ou prend la valeur B
```

A vous de jouer !

Qu'affichent les blocs d'instruction suivants ?

DÉBUT

```
pour i de 1 à 5 faire  
    afficher 10 - 2*i  
fin pour;
```

FIN

Correction : 8 6 4 2 0

DÉBUT

```
pour i de 1 à 5 faire  
    pour j de 6 à 3 par  
        pas de -2 faire  
            afficher i*j  
        fin pour;  
    fin pour;
```

FIN

Correction : 6 4 12 8 18 12 24 16 30
20

Plan

- 1 Généralités
- 2 Structures linéaires**
- 3 Arrêt d'un algorithme
- 4 Validité d'un algorithme
- 5 Complexité en temps d'un algorithme
- 6 Algorithmes de recherche dans un tableau
- 7 Algorithmes de tri

Définition

Dans ce cours, les "structures linéaires" (tableaux 1D, listes, piles, files) contiennent des données de même type "alignées" les unes derrière les autres.

- 1 tableaux 1D
- 2 listes
- 3 piles
- 4 files

Tableaux 1D (statiques)

Définition

Un tableau 1D statique contient un nombre fixe d'éléments appelé *taille* du tableau. Chaque élément du tableau est repéré par un entier naturel situé entre 0 et la taille du tableau -1, appelé l'*indice* de l'élément. Un tableau ne peut pas être vide.

Exemple de représentation

Si le tableau T contient les valeurs 1, 6 et 10 dans cet ordre, on peut le noter

$[1,6,10]$ et le représenter ainsi :

$T[i]$	1	6	10
i	0	1	2

On a $\text{taille}(T)=3$, $T[0]=1$, $T[1]=6$, $T[2]=10$.

A vous de jouer !

Ecrire la valeur du tableau T après les étapes suivantes

Variable : T : tableau de 5 nombres	[?, ?, ?, ?, ?]
T[0] $\leftarrow$ 1	[1, ?, ?, ?, ?]
Pour i de 1 à 4 faire	
T[i] $\leftarrow$ 2*T[i-1]	
finPour	[1, 2, 4, 8, 16]
T[5] $\leftarrow$ 32	Erreur

Utilisation des tableaux : exemple 1

PROCÉDURE :

doubleTab (ES T : tableau d'entier)

SPÉCIFICATIONS :

Double les valeurs de T.

DÉBUT

```
pour i de 0 à Taille(T) - 1 faire  
     $T[i] \leftarrow 2 * T[i]$   
fin pour;
```

FIN

Utilisation des tableaux : exemple 2

FONCTION : creeTabCarres
(E n : entier) : tableau d'entier

SPÉCIFICATIONS : Crée le
tableau des n premiers carrés ($n \in \mathbb{N}$)

Variable : T : tableau de n entiers

DÉBUT

pour i de 0 à Taille(T)-1 **faire**
 $T[i] \leftarrow i^2$

fin pour;

Renvoyer T

FIN

A vous de jouer !

FONCTION : compteEgaux(E e : entier,
E T : tableau d'entiers) : entier

SPÉCIFICATIONS : Calcule et renvoie
le nombre d'éléments de T égaux à e.

Variable : cpt : entier

DÉBUT

```
cpt ← 0 ;  
pour i de 0 à taille(T) - 1 faire  
    si T[i] = e alors  
        cpt ← cpt + 1  
    fin si;  
fin pour;  
Renvoyer cpt
```

FIN

- Création : `T=[4,6,17]` ou `T=[0]*3` ou `T=[i*i for i in range(n)]`
- Accès à un élément : `T[i]`
- Taille du tableau : `len(T)`

Définition

Une liste est soit vide, soit constituée d'un élément, sa tête, suivie d'une liste, sa queue.

- **Constantes** : la liste vide []
- **Fonctions prédéfinies** :
 - Tête d'une liste non vide : **FONCTION** : tête(E liste) : élément
 - Queue d'une liste non vide : **FONCTION** : queue(E liste) : liste
 - Construction avec tête et queue : **FONCTION** : cons (E élément, E liste) : liste
 - Test qu'une liste est vide : **FONCTION** : estVide(E liste) : booléen

Exemple

- $L \leftarrow \text{cons}(1, \text{cons}(2, \text{cons}(3, []))) \longrightarrow L \text{ vaut } [1, 2, 3], \text{ tête } 1, \text{ queue } [2, 3]$
- $A \leftarrow \text{tête}(\text{queue}(\text{queue}(L))) \longrightarrow A \text{ vaut } 3$
- Test si L a au moins 2 éléments : $\text{non}(\text{estVide}(L))$ et $\text{non}(\text{estVide}(\text{queue}(L)))$

A vous de jouer !

Ecrire la valeur de la liste L après les étapes suivantes

Variable : L : liste d'entiers

?

$L \leftarrow \text{cons}(2, [])$

[2]

$L \leftarrow \text{cons}(\text{tête}(L), \text{queue}(L))$

[2]

$L \leftarrow \text{cons}(3, L)$

[3,2]

Pour i de 1 à 5 faire

$L \leftarrow \text{cons}(i, L)$

finPour

[5,4,3,2,1,3,2]

$L \leftarrow \text{queue}(\text{queue}(L))$

[3,2,1,3,2]

$L \leftarrow \text{cons}(L[2], [])$

erreur : L[2] illégal

$L \leftarrow \text{cons}([], 5)$

erreur : usage : cons(élément, liste)

Utilisation des listes : exemple itératif

FONCTION :

longueur (E L : liste d'entier) : entier

SPÉCIFICATIONS :

Renvoie la longueur de la liste L.

Variable : cpt : entier, M : liste d'entiers

DÉBUT

cpt, M $\leftarrow$ 0, L ;

tant que non estVide(M) **faire**

 cpt, M $\leftarrow$ cpt+1, queue(M)

fin tq;

Renvoyer cpt

FIN

Utilisation des listes : exemple récursif

FONCTION :

doubleListe (E L : liste d'entier) : liste d'entier

SPÉCIFICATIONS :

Renvoie une nouvelle liste contenant les valeurs de L doublées.

DÉBUT

```
si estVide(L) alors
    Renvoyer [ ]
sinon
    Renvoyer
    cons(2*tête(L),doubleListe(queue(L)))
fin si;
```

FIN

A vous de jouer ! Compléter les fonctions

FONCTION : cptEgauxLIt (E e : entier, E L : liste d'entiers) : entier

SPÉCIFICATIONS : Compte le nombre d'éléments de L égaux à e (itératif)

Variable : cpt: entier ; M: liste d'entiers

DÉBUT

M, cpt $\leftarrow$ L, 0 ;

tant que non estVide(M) **faire**

si tête(M) = e **alors**

 cpt $\leftarrow$ cpt + 1

fin si ;

 M $\leftarrow$ queue(M)

fin tq ;

Renvoyer cpt

FIN

FONCTION : cptEgauxLRec (E e : entier, E L : liste d'entiers) : entier

SPÉCIFICATIONS : Compte le nombre d'éléments de L égaux à e (récursif)

DÉBUT

si estVide(L) **alors**

Renvoyer 0

sinon si tête(L) = e **alors**

Renvoyer 1 +

 cptEgauxLRec(e, queue(L))

sinon

Renvoyer

 cptEgauxLRec(e, queue(L))

fin si ;

FIN

Remarque

En Python, les listes sont représentées par des tableaux.

- liste vide : []
- tête(liste) : liste[0]
- queue(liste) : liste[1:]
- cons(élément, liste) : [élément]+liste
- estVide(liste) : not liste

Définition

Une pile est soit vide, soit constituée d'un élément, son sommet, suivi d'une pile. On accède à ce qui vient d'être empilé. La lecture du sommet est destructive.

- **Constantes** : la pile vide []
- **Fonctions prédéfinies** :
 - Suppression du sommet d'une pile non vide et retour de sa valeur :
FONCTION : depiler(ES pile) : élément
 - Ajout d'un élément au sommet d'une pile :
PROCÉDURE : empiler(E élément, ES pile)
 - Test qu'une pile est vide :
FONCTION : estVide(E pile) : booléen

Exemple

- $P \leftarrow []$; empiler (1, P) ; empiler(2, P) $\longrightarrow$ P vaut [1, 2] (sommet : 2)
- $A \leftarrow \text{depiler}(P)$ $\longrightarrow$ A vaut 2, P vaut [1]

A vous de jouer !

Calculer la valeur de la pile P après chaque étape

Variable : P : pile d'entiers	?
P ← []	[]
Pour i de 1 à 3 faire	
empiler(i,P)	
finPour	[1,2,3]
empiler(depiler(P),P)	[1,2,3]
empiler(P[1],P)	erreur : P[1] illégal
empiler(P,4)	erreur : usage : empiler(élt, pile)
P ← cons(3,P)	erreur : cons illégal
depiler(P)	[1,2]
depiler(P)	[1]
depiler(P)	[]
depiler(P)	erreur : P est vide

Utilisation des piles : exemple

PROCÉDURE :

doublePile (ES P : pile d'entier)

SPÉCIFICATIONS :

Double les valeurs de P.

Variable : Q : pile d'entier

DÉBUT

Q $\leftarrow$ [] ;

tant que non estVide(P) **faire**
 empiler(2*depiler(P), Q)

fin tq;

tant que non estVide(Q) **faire**
 empiler(depiler(Q), P)

fin tq;

FIN

A vous de jouer ! Compléter la fonction

PROCÉDURE : invPilelte (ES P : pile d'entier)

SPÉCIFICATIONS : Inverse l'ordre des valeurs de la pile P.

Variable : Q, R : pile d'entier

DÉBUT

Q, R $\leftarrow$ [], [] ;

tant que non estVide(P) **faire**
 empiler(depiler(P), Q)

fin tq;

tant que non estVide(Q) **faire**
 empiler(depiler(Q), R)

fin tq;

tant que non estVide(R) **faire**
 empiler(depiler(R), P)

fin tq;

FIN

Remarque

En Python, les piles sont représentées par des tableaux.

- pile vide : []
- depiler(P) : P.pop()
- empiler(e, P) : P.append(e)
- estVide(P) : not P

Définition

Une file est soit vide, soit constituée d'un élément, sa tête, suivi d'une file, sa queue. On accède au premier élément enfilé. La lecture du sommet est destructive.

- **Constantes** : la file vide []
- **Fonctions prédéfinies** :
 - Suppression de la tête d'une file non vide et retour de sa valeur :
FONCTION : defiler(ES file) : élément
 - Ajout d'un élément en fin de file :
PROCÉDURE : enfiler(E élément, ES file)
 - Test qu'une file est vide :
FONCTION : estVide(E file) : booléen

Exemple

- $F \leftarrow []$; enfiler(1, F) ; enfiler (2, F) ; enfiler (3, F) $\longrightarrow F = [1, 2, 3]$ (tête : 1)
- $A \leftarrow \text{defiler}(F) \longrightarrow A \text{ vaut } 1, F \text{ vaut } [2, 3]$

A vous de jouer !

Calculer la valeur de la file F après chaque étape

Variable : F : file d'entiers	?
F $\leftarrow$ []	[]
Pour i de 1 à 3 faire	
enfiler(i,F)	
finPour	[1,2,3]
enfiler(defiler(F),F)	[2,3,1]
enfiler(F[1],F)	erreur : F[1] illégal
enfiler(F,4)	erreur : usage : enfiler(élt, file)
F $\leftarrow$ cons(3,F)	erreur : cons illégal
defiler(F)	[3,1]
defiler(F)	[1]
defiler(F)	[]
defiler(F)	erreur : F est vide

A vous de jouer ! Compléter la fonction

PROCÉDURE : inverseFile (ES F : file d'entier)

SPÉCIFICATIONS : Inverse l'ordre des éléments de la file F

Variable : P : pile d'entier

DÉBUT

P $\leftarrow$ [];

tant que non estVide(F) **faire**
 empiler(**defiler**(F), P)

fin tq;

tant que non estVide(P) **faire**
 enfiler(**depiler**(P), F)

fin tq;

FIN

Remarque

En Python, les files sont représentées par des tableaux.

- file vide : []
- defiler(F) : F.popleft() [utilise la collection deque]
- enfiler(e, F) : F.insert(0,e)
- estVide(F) : not P

Ne pas confondre pseudo-code et Python

En pseudo-code, même si les tableaux, listes, piles, files partagent la même notation, **les opérateurs d'une structure ne fonctionnent pas sur les autres** ! C'est différent du Python !

Plan

- 1 Généralités
- 2 Structures linéaires
- 3 Arrêt d'un algorithme**
- 4 Validité d'un algorithme
- 5 Complexité en temps d'un algorithme
- 6 Algorithmes de recherche dans un tableau
- 7 Algorithmes de tri

Un bon algorithme...

... est un algorithme qui se termine, c'est à dire qui effectue un nombre fini d'opérations. Cette section présente une méthode pour prouver qu'un algorithme se termine.

Méthode

Pour prouver qu'un algorithme itératif se termine, on prouve :

- qu'il y a un nombre fini d'itérations
- que chaque itération se termine.

Exemple

FONCTION : F1 (E n : entier) : entier

Variable : res : entier

DÉBUT

res $\leftarrow$ 1

pour i de 1 à n **faire**

res $\leftarrow$ res * i

fin pour;

Renvoyer res

FIN

- Chaque itération contient 2 opérations donc se termine.
- i parcourt les indices 1 à n donc il y a un nombre fini d'itérations.
- Donc l'algorithme se termine.

Deuxième exemple de preuve d'arrêt

Exemple

FONCTION : F ($\underline{E}$ n : entier) : entier

Variable : p, res : entier

DÉBUT

$p, res \leftarrow n, 0$

tant que $p > 0$ **faire**

$p, res \leftarrow p \text{ div } 2, res + 1$

fin tq;

Renvoyer res

FIN

- Chaque itération contient 2 affectations et 2 opérations donc se termine.
- $p \in \mathbb{N}$ diminue strictement à chaque itération jusqu'à atteindre 0. Lorsqu'il atteint 0, la boucle se termine donc il y a un nombre fini d'itérations.
- Donc l'algorithme se termine.

A vous de jouer ! 1/2

F se termine-t-il ? Si oui, le prouver, sinon, corriger.

FONCTION :

F (E n : entier) : entier

SPÉCIFICATIONS : Renvoie le chiffre des unités de $n \in \mathbb{N}$

Variable : i : entier

DÉBUT

i $\leftarrow$ n

tant que i $\neq$ 0 **faire**

i $\leftarrow$ i - 10

fin tq;

Renvoyer i

FIN

- Chaque itération contient 2 opérations donc se termine.
- Si n=9 par exemple, i prend les valeurs 9, -1, -11, -21 ... et il y a un nombre infini d'itérations : l'algorithme ne s'arrête pas.
- Modification : remplacer la condition $i \neq 0$ par $i \geq 10$.

G se termine-t-il ? Si oui, le prouver, sinon, corriger.

FONCTION :

G (E a,b : entier) : entier

SPÉCIFICATIONS : a, b $\in \mathbb{N}$ avec
a $\leq$ b. Calcule quelque chose.

Variable : i : entier

DÉBUT

i $\leftarrow$ a

tant que i < b **faire**

i $\leftarrow$ i + cond(i mod 7 = 0, 3, 1)

fin tq;

Renvoyer i

FIN

- Chaque itération contient 5 opérations dont se termine.
- i progresse au moins d'une unité à chaque itération, donc i $\geq$ b en au maximum b-a itérations. Comme les itérations s'arrêtent lorsque i $\geq$ b, il y a un nombre fini d'itérations.
- L'algorithme se termine.

Méthode

Pour prouver qu'un algorithme récursif se termine, on montre :

- qu'il y a un nombre fini d'appels récursifs ;
- qu'en dehors des appels récursifs, il y a un nombre fini d'opérations.

Exemple

FONCTION : F2 (E n : entier) : entier

SPÉCIFICATIONS : calcule la factorielle d'un entier $n \in \mathbb{N}$.

DÉBUT

```
si  $n=0$  alors
  Renvoyer 1
sinon
  Renvoyer  $n \cdot \text{F2}(n-1)$ 
fin si;
```

FIN

- $n \in \mathbb{N}$ décroît d'une unité à chaque appel et il y a un cas de base pour $n=0$, donc il y a un nombre fini d'appels récursifs.
- En dehors des appels récursifs, il y a un nombre borné d'opérations.
- L'algorithme se termine.

F se termine-t-il ? Si oui, le prouver, sinon, corriger.

FONCTION :

F (E n : entier) : entier

SPÉCIFICATIONS : $n \in \mathbb{N}$.

Calcule $2n$ sans multiplication.

DÉBUT

si $n = 0$ **alors**

Renvoyer 0

sinon

Renvoyer $2+F(n+1)$

fin si;

FIN

- En dehors des appels récursifs, il y a un nombre fini d'opérations.
- $n \in \mathbb{N}$ augmente d'une unité à chaque appel donc n'atteint jamais 0.
- Modification : remplacer $F(n+1)$ par $F(n-1)$.

G se termine-t-il ? Si oui, le prouver, sinon, corriger.

FONCTION :

G (E a,b : entier) : entier

SPÉCIFICATIONS : $a, b \in \mathbb{N}^*$ avec
 $a \leq b$. Calcule quelque chose.

DÉBUT

si $b \bmod a = 0$ **alors**

Renvoyer a

sinon

Renvoyer G(a-1,b)

fin si;

FIN

- En dehors des appels récursifs, il y a un nombre fini d'opérations.
- a diminue de 1 à chaque appel, 1 est un diviseur de b donc le cas de base est atteint en au plus $a - 1$ appels récursifs. Il y a donc un nombre fini d'appels récursifs.
- L'algorithme se termine.

Plan

- 1 Généralités
- 2 Structures linéaires
- 3 Arrêt d'un algorithme
- 4 Validité d'un algorithme**
- 5 Complexité en temps d'un algorithme
- 6 Algorithmes de recherche dans un tableau
- 7 Algorithmes de tri

Un bon algorithme...

... est un algorithme valide, c'est à dire conforme aux spécifications. Cette section présente une méthode pour prouver qu'un algorithme est valide.

Vocabulaire

- **équation de récurrence** : égalité reliant les valeurs des variables d'une itération à la suivante ;
- **invariant de boucle** : propriété censée être vraie à chaque itération.

Méthode

- 1 "lire" les équations de récurrence sur l'algorithme ;
- 2 chercher un invariant de boucle ;
- 3 prouver l'invariant par récurrence ou induction ;
- 4 écrire l'invariant à la fin des itérations.

Premier exemple de preuve de validité

FONCTION :

F1 ($\underline{E} \ n : \text{entier}$) : entier

SPÉCIFICATIONS :

Calcule $n!$ avec $n \in \mathbb{N}$

Variable : res : entier

DÉBUT

res $\leftarrow$ 1

pour i de 1 à n **faire**

 res $\leftarrow$ res * i

fin pour;

Renvoyer res

FIN

var_i : valeur de var à la fin de l'itération i .

① Équation de récurrence :

$res_0 = 1$ et si $i > 0$, $res_i = res_{i-1} \times i$.

② Invariant proposé : Inv_i : " $res_i = i!$ ".

③ Preuve de l'invariant :

- **Initialisation** : Pour $i = 0$, on a $res_0 = 1 = 0!$ donc Inv_0 est vrai.
- **Récurrence** : Pour $1 \leq i \leq n$, on suppose Inv_{i-1} vrai, c'est à dire $res_{i-1} = (i-1)!$.
D'après les équations de récurrence, $res_i = res_{i-1} \times i$.
Donc $res_i = (i-1)! \times i = i!$.
Donc Inv_i est vrai.
- **Conclusion** : l'invariant est vrai pour toutes les itérations.

④ A la fin du pour, la valeur renvoyée est $res_n = n!$: l'algorithme est valide.

Deuxième exemple de preuve de validité

FONCTION :

Cube ($\underline{E} \ n : \text{entier}$) : entier

SPÉCIFICATIONS :

Calcule n^3 avec $n \in \mathbb{N}$.

Variable : A, B, C, Z : entier

DÉBUT

A, B, C, Z $\leftarrow$ 1, 0, n, 0 ;

tant que C > 0 **faire**

 Z $\leftarrow$ Z + A + B ;

 B $\leftarrow$ B + A + A + 1 ;

 A, C $\leftarrow$ A+3, C-1 ;

fin tq;

Renvoyer Z

FIN

var_i : valeur de var à la fin de l'itération i .

Équations de récurrence :

$$\begin{pmatrix} A_0 \\ B_0 \\ C_0 \\ Z_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ n \\ 0 \end{pmatrix} \text{ et } \begin{pmatrix} A_{i+1} \\ B_{i+1} \\ C_{i+1} \\ Z_{i+1} \end{pmatrix} = \begin{pmatrix} A_i + 3 \\ B_i + 2A_i + 1 \\ C_i - 1 \\ Z_i + A_i + B_i \end{pmatrix}$$

Invariant difficile : "trace" pour $n=4$:

itération i	A_i	B_i	C_i	Z_i
0	1	0	4	0
1	4	3	3	1
2	7	12	2	8
3	10	27	1	27
4	13	48	0	64

Invariant proposé :

$$Inv_i : \begin{pmatrix} A_i \\ B_i \\ C_i \\ Z_i \end{pmatrix} = \begin{pmatrix} 3i + 1 \\ 3i^2 \\ n - i \\ i^3 \end{pmatrix}$$

Deuxième exemple - suite

1 Équation de récurrence et invariant :

$$ER : \begin{pmatrix} A_0 \\ B_0 \\ C_0 \\ Z_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ n \\ 0 \end{pmatrix} \text{ et } \begin{pmatrix} A_{i+1} \\ B_{i+1} \\ C_{i+1} \\ Z_{i+1} \end{pmatrix} = \begin{pmatrix} A_i + 3 \\ B_i + 2A_i + 1 \\ C_i - 1 \\ Z_i + A_i + B_i \end{pmatrix} \quad Inv_i : \begin{pmatrix} A_i \\ B_i \\ C_i \\ Z_i \end{pmatrix} = \begin{pmatrix} 3i + 1 \\ 3i^2 \\ n - i \\ i^3 \end{pmatrix}$$

2 Preuve de l'invariant :

- **Initialisation** : Pour $i = 0$, $\begin{pmatrix} A_0 \\ B_0 \\ C_0 \\ Z_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ n \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \times 0 + 1 \\ 3 \times 0^2 \\ n - 0 \\ 0^3 \end{pmatrix}$: Inv_0 est vrai.

- **Récurrence** : Pour $0 < i \leq n$, on suppose Inv_i vrai. On utilise ER et Inv_i :

$$\begin{pmatrix} A_{i+1} \\ B_{i+1} \\ C_{i+1} \\ Z_{i+1} \end{pmatrix} = \begin{pmatrix} A_i + 3 \\ B_i + 2A_i + 1 \\ C_i - 1 \\ Z_i + A_i + B_i \end{pmatrix} = \begin{pmatrix} 3i + 1 + 3 \\ 3i^2 + 2(3i + 1) + 1 \\ n - i - 1 \\ i^3 + 3i + 1 + 3i^2 \end{pmatrix} = \begin{pmatrix} 3(i + 1) + 1 \\ 3(i + 1)^2 \\ n - (i + 1) \\ (i + 1)^3 \end{pmatrix}$$

Donc Inv_{i+1} est vrai.

- **Conclusion** : l'invariant est vrai pour toutes les itérations.

- **Conclusion** : à la fin du TantQue, $C_i = 0 = n - i$ donc $i = n$.
La valeur renvoyée est $Z_n = n^3$: l'algorithme est valide.

A vous de jouer ! Compléter la preuve

FONCTION : somTab

(E T : tableau d'entiers) : entier

SPÉCIFICATIONS : Renvoie
la somme des valeurs de T.

Variable : res : entier

DÉBUT

res $\leftarrow$ T[0]

pour i de 1 à

taille(T)-1 **faire**

res $\leftarrow$ res+T[i]

fin pour;

Renvoyer res

FIN

① **Eq. récurrence** : $res_0 = T[0]$

$res_{i+1} = res_i + T[i+1]$

② **Invariant** : Inv_i : " res_i est la somme des
valeurs $T[0]$ à $T[i]$ ".

③ **Preuve de l'invariant** ($n=taille(T)$) :

- **Initialisation** : pour $i = 0$,
 $res_0 = T[0]$ somme des valeurs de
 $T[0]$ à $T[0]$. Inv_0 est vrai.
- **Récurrence** : on suppose Inv_i vrai
pour $0 \leq i \leq n-2$.
Comme $res_{i+1} = res_i + T[i+1]$
et $res_i = T[0] + \dots + T[i]$
on a $res_{i+1} = T[0] + \dots + T[i+1]$.
Donc Inv_{i+1} est vrai.
- **Conclusion** : l'invariant est vrai
pour toutes les itérations.

④ A la fin du pour, l'algorithme renvoie
 $res_{n-1} = T[0] + \dots + T[n-1]$:
l'algorithme est valide.

Vocabulaire

- **équation de récurrence** : égalité reliant le résultat de l'algorithme avec celui des appels récursifs ;
- **invariant de récursion** : propriété censée être vraie à chaque appel récursif. Le plus souvent, c'est simplement le résultat principal de l'algorithme.

Méthode

- 1 "lire" les équations de récurrence sur l'algorithme ;
- 2 poser l'invariant de récursion ;
- 3 prouver l'invariant par récurrence ou induction ;
- 4 la conclusion de la récurrence prouve l'algorithme.

Exemple de preuve de validité

FONCTION :

F2 (E n : entier) : entier

SPÉCIFICATIONS : calcule
la factorielle d'un entier $n \in \mathbb{N}$.

DÉBUT

```
si  $n=0$  alors
    Renvoyer 1
sinon
    Renvoyer  $n \times F2(n-1)$ 
fin si;
```

FIN

- ❶ **Équation de récurrence :**
 $F2(0) = 1$ et $F2(n) = n \times F2(n-1)$
pour $n > 0$.
- ❷ **Invariant proposé :**
 $Inv_n : "F2(n) = n!"$
- ❸ **Preuve de l'invariant :**
 - **Initialisation :** Pour $n = 0$, on a $F2(0) = 1 = 0!$ donc Inv_0 est vrai.
 - **Récurrence :** Pour $n > 0$, on suppose Inv_{n-1} vrai, c'est à dire $F2(n-1) = (n-1)!$ Alors $F2(n) = n \times F2(n-1) = n \times (n-1)! = n!$. Donc Inv_n est vrai.
 - **Conclusion :** pour tout $n \in \mathbb{N}$, $F2(n) = n!$
- ❹ **L'algorithme est valide.**

A vous de jouer ! Compléter la preuve

FONCTION :

F4 ($\underline{E}$ a, n : entier) : entier

SPÉCIFICATIONS : Renvoie la somme des entiers de $[a, n]$

DÉBUT

Renvoyer cond($n < a$, 0,
 $n + F4(a, n-1)$)

FIN

- **Équations de récurrence :**

si $n < a$ alors $F4(a, n) = 0$

et si $n \geq a$ alors $F4(n) = n + F4(a, n-1)$

- **Invariant :**

Inv_n : " $F4(a, n) = \text{somme de } [a, n]$ ".

- **Initialisation** : Pour $n < a$, $F(a, n) = 0$.
Comme $[a, n]$ est vide, sa somme vaut 0.
Donc Inv_0 est vrai.
- **Récurrence** : Pour $n \geq a$,
on suppose Inv_{n-1} vrai :
 $F4(a, n-1) = \text{somme de } [a, n-1]$.
Comme $F4(a, n) = n + F4(a, n-1)$,
 $F4(a, n)$ est la somme de $[a, n]$.
Donc Inv_n est vrai.
- **Conclusion** : pour tout $n \in \mathbb{N}$,
 $F4(a, n)$ renvoie la somme de $[a, n]$.

- **L'algorithme est valide.**

Plan

- 1 Généralités
- 2 Structures linéaires
- 3 Arrêt d'un algorithme
- 4 Validité d'un algorithme
- 5 Complexité en temps d'un algorithme**
- 6 Algorithmes de recherche dans un tableau
- 7 Algorithmes de tri

Un bon algorithme...

... est un algorithme efficace, c'est à dire qu'il effectue un nombre raisonnable d'opérations et utilise un espace de calcul raisonnable.

Définitions

- **Complexité en temps** : nombre d'opérations élémentaires en fonction des paramètres d'entrée.
- **Complexité en espace** : nombre de données à mémoriser en fonction des paramètres d'entrée.
- **Algorithme efficace** : algorithme de faible complexité.
- **Algorithme optimal** : algorithme de complexité minimale (impossible de faire mieux).

Ce cours se limite :

- à la complexité en temps, appelée abusivement "complexité" ;
- au cas où elle s'exprime en fonction d'un entier n "*taille de l'entrée*" : taille d'un tableau, longueur d'un entier...

Notation : T_n : complexité pour une taille d'entrée n .

Il effectue les simplifications suivantes :

- opérations élémentaires comptées à raison d'une unité par opération (modèle Word-Ram) ;
- majoration du nombre d'opérations ("pire des cas") ;
- comparaison de la croissance asymptotique de T_n (= pour de grandes valeurs de n) avec celle de puissances, exp, log.

Premier exemple

FONCTION :

F4 (E n : entier) : entier

Variable : som : nombre

DÉBUT

som $\leftarrow$ 0

pour i de 1 à n **faire**

si i mod 2 = 1 **alors**

 som $\leftarrow$ som + i*i

sinon

 som $\leftarrow$ som + i

fin si;

fin pour;

Renvoyer som

FIN

- **Nb opérations :** 1 affectation, n itérations composées d'1 modulo, 1 comparaison, 1 affectation, 1 somme et éventuellement 1 produit.
- **Simplification :** chaque itération génère entre 4 et 5 opérations.
- **Complexité :** entre $4n + 1$ et $5n + 1$
- **Ordre de complexité :** n .

Deuxième exemple

FONCTION : F5 (E n : entier) :
booléen

SPÉCIFICATIONS : détermine si
 $n \in \mathbb{N}^*$ est une puissance de 2.

DÉBUT

```
si  $n=1$  alors
    Renvoyer true
sinon si  $n \bmod 2 = 1$  alors
    Renvoyer false
sinon
    Renvoyer F5( $n/2$ )
fin si;
```

FIN

- **Nombre d'appels récursifs :**
nombre de fois que n peut être divisé par 2.
- **Pire des cas :** n divisé par 2
plusieurs fois jusqu'à tomber sur
1, soit $\log_2(n)$ divisions.
- **Nombre d'appels :** $\log_2(n)$
appels récursifs, $\log_2(n) + 1$ avec
le premier.
- **Nb d'opérations par appel :**
5, sauf le dernier : 1.
- **Complexité pire des cas :**
 $5\log_2(n) + 1$
- **Ordre de complexité :** $\log(n)$.

Domination de suites numériques

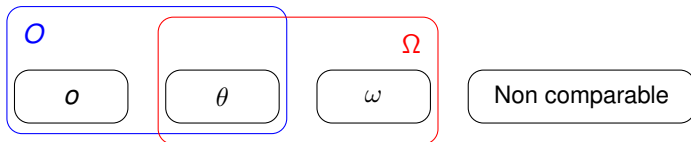
Définitions

Soient T_n et U_n deux suites numériques.

- T_n **domine** U_n s'il existe $C \in \mathbb{R}$ et $n_0 \in \mathbb{N}$ tels que $\forall n \geq n_0, U_n \leq CT_n$.
- T_n et U_n sont **du même ordre** si T_n domine U_n et U_n domine T_n .
- T_n **domine strictement** U_n si elle domine U_n sans être du même ordre.

Notations de Landau

- T_n domine U_n : $T_n = \Omega(U_n)$ ou $U_n = O(T_n)$
- T_n et U_n sont du même ordre : $T_n = \theta(U_n)$ ou $U_n = \theta(T_n)$
- T_n domine strictement U_n : $T_n = \omega(U_n)$ ou $U_n = o(T_n)$



Propriétés

Soient T_n et U_n deux suites **strictement positives** à partir d'un certain rang.

- si T_n/U_n a pour limite 0, alors U_n domine strictement T_n .
- si T_n/U_n a une limite finie >0 , alors T_n et U_n sont du même ordre.
- si T_n/U_n a pour limite $+\infty$, alors T_n domine strictement U_n .

Remarques

- Réciproque de la deuxième propriété fausse ;
- Deux suites qui ne se dominent pas l'une l'autre sont "non comparables asymptotiquement".

A vous de jouer

Dire si chaque suite est o , θ , ω des suites de référence.

T_n	1	$\log(n)$	n	$n\log(n)$	n^2	2^n	$n!$
$3n^2 - 6n + 500$	ω	ω	ω	ω	θ	o	o
$\log(50n^2)$	ω	θ	o	o	o	o	o
$2^{(n^2)}$	ω	ω	ω	ω	ω	ω	o
$2^n - n^2$	ω	ω	ω	ω	ω	θ	o
$(n+1)!$	ω	ω	ω	ω	ω	ω	ω

Catégories de complexité

Un algorithme est dit :

- **à coût constant** si $T_n = O(1)$ (abus de langage) ;
- **logarithmique** si $T_n = O(\log(n))$;
- **linéaire** si $T_n = \theta(n)$;
- **semi-linéaire** si $T_n = \theta(n \log(n))$;
- **quadratique** si $T_n = \theta(n^2)$;
- **polynomial** si $T_n = O(n^a)$ avec a entier naturel ;
- **exponentiel** si $T_n = \theta(a^n)$ avec $a > 1$.

Propriétés

- Croissances comparées : chaque suite est strictement dominée par la suivante :
 $1, \log(n), \sqrt{n}, n, n\log(n), n^2, n^3, 2^n, 3^n, n!$
- tous les logarithmes sont du même ordre de grandeur :
 $\log_{10}(n) = \theta(\log_2(n)) = \theta(\ln(n))$ etc.
- $o, O, \omega, \Omega, \theta$ sont multiplicatives sur les suites strictement positives :
si $a = o(a')$ et $b = o(b')$ alors $ab = o(a'b')$ et de même pour $O, \omega, \Omega, \theta$.

Propriété

Si des boucles *pour* imbriquées contiennent un nombre borné d'opérations élémentaires, alors leur complexité est du même ordre que le produit du nombre de valeurs prises par les indices de boucle.

Exemple

DÉBUT

```
pour  $i$  de 1 à  $n$  faire  
    pour  $j$  de 1 à  $i$  faire  
        op élém  
    fin pour;  
fin pour;
```

FIN

i prend n valeurs = $O(n)$, j prend i valeurs = $O(n)$, donc l'ensemble des deux boucles a une complexité $O(n^2)$.

Propriétés

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} ; \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} ; \sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$$

Exemple

DÉBUT

```
pour i de 1 à n faire  
  pour j de 1 à i faire  
    op élém  
  fin pour;  
fin pour;
```

FIN

Nombre d'opérations : $\sum_{i=1}^n \sum_{j=1}^i 1 = \sum_{i=1}^n i = \frac{n(n+1)}{2} = \theta(n^2)$

A vous de jouer !

Complexités en fonction de n ?

- ❶ Pour i de 1 à n faire
 Pour j de 1 à n faire
 som ← som+1
- ❷ Pour i de 1 à n faire
 Pour j de 1 à n faire
 Pour k de 1 à n faire
 som ← som+1
- ❸ Pour i de 1 à n faire
 Pour j de i à n faire
 som ← som + 1
- ❹ i ← 0 ; TantQue $i^2 < n$ faire
 i ← i+1

- ❶ Itérations à coût borné,
n valeurs pour les deux pour :
 $T_n = \theta(n^2)$
- ❷ Itérations à coût borné,
n valeurs pour les trois pour :
 $T_n = \theta(n^3)$
- ❸ $T_n = \sum_{i=1}^n \sum_{j=i}^n 2 =$
 $2 \sum_{i=1}^n n - i + 1 =$
 $2 \sum_{i'=1}^n i' = n(n+1)$
- ❹ $i^2 < n$ équivaut à $i < \sqrt{n}$
donc $T_n = 2 \lceil \sqrt{n} \rceil$

Méthode pour les algorithmes récursifs

- Chercher une équation de récurrence entre la complexité de l'algorithme et celle de ses appels récursifs ;
- Résoudre cette équation.

Exemple

FONCTION : F2 (E n : entier) : entier

DÉBUT

si $n=0$ **alors**

Renvoyer 1

sinon

Renvoyer $n \times F2(n - 1)$

fin si;

FIN

- Equation de récurrence :
$$\begin{cases} T_0 &= 1 \\ T_n &= 4 + T_{n-1} \text{ si } n > 0 \end{cases}$$
- Résolution : $T_n = 4n + 1 = \theta(n)$.

Complexité en soustractions ? Compléter le raisonnement

FONCTION : estPair ($\underline{E}$ n : entier) : booléen

DÉBUT

Renvoyer cond($n=0$, true, cond($n=1$, false, estPair($n-2$)))

FIN

- Équation de récurrence : $T_0 = 0$, $T_1 = 0$, si $n \geq 2$ alors $T_n = 1 + T_{n-2}$
- Valeurs de T_n : 0, 0, 1, 1, 2, 2, ...
- Valeur de T_n en fonction de n : $\lfloor n/2 \rfloor$.
- Ordre : T_n est entre $n/2 - 1$ et $n/2$ donc $\theta(n)$.

Master Theorem

Master Theorem

Master Theorem : s'il existe quatre entiers $a \geq 0$, $b > 1$, $d \geq 0$ et $n_0 > 0$ tels que, pour tout $n \geq n_0$, $T(n) \leq aT(\lceil \frac{n}{b} \rceil) + O(n^d)$, alors

$$T(n) = \begin{cases} O(n^d) & \text{si } b^d > a \\ O(n^d \log n) & \text{si } b^d = a \\ O(n^{\frac{\log a}{\log b}}) & \text{si } b^d < a \end{cases}$$

Remarques

- Le M.T. donne un ordre de complexité sans équation de récurrence lorsque les appels se font avec une taille d'entrée divisée.
- a représente le nombre d'appels récurrents dans le corps de l'algorithme, b l'entier par lequel la taille de l'entrée est divisée à chaque appel, $O(n^d)$ les opérations de l'algorithme en plus des appels récurrents.

Exemple

FONCTION : F3 (E n, m : entier) :
entier

DÉBUT

si $n=0$ **alors**

Renvoyer 0

sinon si $n \bmod 2 = 0$ **alors**

Renvoyer $2 \cdot \text{F3}(n/2, m)$

sinon

Renvoyer

$n + 2 \cdot \text{F3}((n-1)/2, m)$

fin si;

FIN

A partir de $n=1$ ($n_0 = 1$),

F3 fait 1 appel récursif ($a = 1$)

au rang $\lfloor n/2 \rfloor$ ($b = 2$).

En dehors des appels, il effectue

$O(1) = O(n^0)$ opérations ($d = 0$).

On est dans le cas $b^d = a$ donc

$T_n = O(n^0 \log(n)) = O(\log(n))$:

l'algorithme est logarithmique.

A vous de jouer !

Trouver a, b, d et un ordre de complexité

PROCÉDURE : F (E n : entier)

SPÉCIFICATIONS : $n \in \mathbb{N}$

DÉBUT

si $n \neq 1$ **alors**

pour i de 0 à n-1 **faire**

pour j de 0 à n-1 **faire**
 op élém

fin pour;

fin pour;

 F(n div 2)

 F(n div 2)

fin si;

FIN

- 2 appels récursifs : **a=2**
- au rang $\lfloor \frac{n}{2} \rfloor$: **b=2**
- autres opérations :
 $\theta(n^2)$: **d=2**
- cas $b^d > a$:
algorithme en $O(n^2)$.

Plan

- 1 Généralités
- 2 Structures linéaires
- 3 Arrêt d'un algorithme
- 4 Validité d'un algorithme
- 5 Complexité en temps d'un algorithme
- 6 Algorithmes de recherche dans un tableau**
- 7 Algorithmes de tri

Principe

Pour chercher un élément e dans un tableau T non trié, on le parcourt case par case jusqu'à trouver e ou atteindre la fin du tableau.

Algorithme de recherche linéaire

FONCTION : RechLin(E e : nombre, E T : tableau de nombres) : entier

SPÉCIFICATIONS : Si T contient e, renvoie le premier indice de T qui contient e. Sinon, renvoie -1.

Variable : i : entier

DÉBUT

 i ← 0

tant que i < taille(T) **et** T[i] ≠ e **faire**
 i ← i + 1

fin tq;

si i = taille(T) **alors**
 Renvoyer -1

sinon
 Renvoyer i

fin si;

FIN

DÉBUT

$i \leftarrow 0$

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

$i \leftarrow i+1$

fin tq;

si $i = \text{taille}(T)$ **alors**

Renvoyer -1

sinon

Renvoyer i

fin si;

FIN

Complexité : pire des cas pour $e \notin T$ en notant $n = \text{taille}(T)$:
 n itérations à coût constant + opérations à coût constant = $\theta(n)$.

Algorithme de recherche dichotomique

Principe

Pour chercher un élément e dans un tableau T trié, on compare e à la valeur au milieu du tableau puis on continue à le chercher dans la bonne moitié du tableau.

Algorithme de recherche dichotomique

FONCTION : RechDic (E e : nombre, E T : tableau de nombres) : entier

SPÉCIFICATIONS : T **trié**. Renvoie le plus petit i tq $T[i]=e$ ou -1.

Variable : deb, mil, fin : entier

DÉBUT

deb, fin $\leftarrow$ 0, taille(T)-1

tant que $deb \leq fin$ **faire**

mil $\leftarrow$ (deb+fin) div 2

si $T[mil] < e$ **alors**

deb $\leftarrow$ mil+1

sinon

fin $\leftarrow$ mil-1

fin si;

fin tq;

Renvoyer cond(deb < taille(T) et $T[deb]=e$, deb, -1)

FIN

A vous de jouer !

Trace de l'algorithme $T=[1,2,3,3,3,4,6,9,10]$, $e=3$?

DÉBUT

$deb, fin \leftarrow 0, \text{taille}(T)-1$

tant que $deb \leq fin$ **faire**

$mil \leftarrow (deb+fin) \text{ div } 2$

si $T[mil] < e$ **alors**

$deb \leftarrow mil+1$

sinon

$fin \leftarrow mil-1$

fin si;

fin tq;

Renvoyer

$\text{cond}(deb < \text{taille}(T) \text{ et}$

$T[deb]=e, deb, -1)$

FIN

i	deb_i	mil_i	fin_i	$T[deb_i..fin_i]$
0	0	?	8	[1,2,3,3, 3 ,4,6,9,10]
1	0	4	3	[1, 2 ,3,3]
2	2	1	3	[3 ,3]
3	2	2	1	

L'algorithme renvoie $deb=2$.

A vous de jouer !

Trace de l'algorithme pour $T=[1,2,3,3,3,4,6,9,10]$ et $e=5$?

DÉBUT

$deb, fin \leftarrow 0, \text{taille}(T)-1$

tant que $deb \leq fin$ **faire**

$mil \leftarrow (deb+fin) \text{ div } 2$

si $T[mil] < e$ **alors**

$deb \leftarrow mil+1$

sinon

$fin \leftarrow mil-1$

fin si;

fin tq;

Renvoyer

$\text{cond}(deb < \text{taille}(T) \text{ et }$

$T[deb]=e, deb, -1)$

FIN

i	deb_i	mil_i	fin_i	$T[deb_i..fin_i]$
0	0	?	8	[1,2,3,3, 3 ,4,6,9,10]
1	5	4	8	[4, 6 ,9,10]
2	5	6	5	[4]
3	6	5	5	

$deb = 6$ et $T[deb] \neq 5$:
l'algorithme renvoie -1.

DÉBUT

deb, fin $\leftarrow$ 0, taille(T)-1

tant que $\text{deb} \leq \text{fin}$ **faire**

 mil $\leftarrow$ (deb+fin) div 2

si $T[\text{mil}] < e$ **alors**

 deb $\leftarrow$ mil+1

sinon

 fin $\leftarrow$ mil-1

fin si;

fin tq;

Renvoyer

 cond($\text{deb} < \text{taille}(T)$ et

$T[\text{deb}] = e$, deb, -1)

FIN

- fin-deb+1 est un entier naturel ;
- il décroît strictement à chaque itération ;
- le tant que s'arrête lorsqu'il vaut 0 ou moins.

Il y a donc un nombre fini d'itérations.

Chaque itération se termine.

Donc **l'algorithme se termine.**

Preuve de validité (principe)

DÉBUT

$deb, fin \leftarrow 0, \text{taille}(T)-1$

tant que $deb \leq fin$ **faire**

$mil \leftarrow (deb+fin) \text{ div } 2$

si $T[mil] < e$ **alors**

$deb \leftarrow mil+1$

sinon

$fin \leftarrow mil-1$

fin si;

fin tq;

Renvoyer

$\text{cond}(deb < \text{taille}(T) \text{ et } T[deb]=e, deb, -1)$

FIN

- **Notation** : $expr_i$: valeur de $expr$ à la fin de l'itération i ; n : $\text{taille}(T)$

- **Equations de récurrence** :

- $mil_{i+1} = (deb_i + fin_i) \text{ div } 2$

- si $T[mil_{i+1}] < e$ alors

$$deb_{i+1} = mil_{i+1} + 1 \text{ et } fin_{i+1} = fin_i$$

- sinon : $deb_{i+1} = deb_i$ et $fin_{i+1} = mil_{i+1} - 1$

- **Invariant** (admis) :

$$0 \dots deb_i - 1 \dots fin_i + 1 \dots n - 1$$

$< e$		$\geq e$
-------	--	----------

- En sortant du TantQue, $deb = fin + 1$:

$$0 \dots deb - 1 \quad deb \dots n - 1$$

$< e$	$\geq e$
-------	----------

- soit $deb < n$ et $T[deb] = e$: alors deb est le premier indice où se trouve e .
- sinon, c'est que $e \notin T$.
- Dans les deux cas, l'algorithme renvoie le bon résultat : **l'algorithme est valide.**

Complexité de l'algorithme dichotomique

DÉBUT

```
deb, fin ← 0, taille(T)-1
tant que deb ≤ fin faire
    mil ← (deb+fin) div 2
    si T[mil] < e alors
        deb ← mil+1
    sinon
        fin ← mil-1
    fin si;
fin tq;
Renvoyer
cond(deb < taille(T) et
    T[deb] = e, deb, -1)
```

FIN

- Si n est une puissance de 2, dans le pire des cas où tout le tableau est $< e$:
 - $fin - deb + 1$ est divisé par 2 à chaque itération jusqu'à 0.
 - Le nombre d'itérations k vérifie $n/2^{k-1} = 1$ donc $n = 2^{k-1}$ donc $k = \log_2(n) + 1$.
 - chaque itération fait 7 ou 8 opérations donc complexité en $\theta(\log(n))$.
- Sinon, $2^k < n < 2^{k+1}$:
 - T est "compris" entre un tableau de taille 2^k et un de taille 2^{k+1} .
 - Complexité entre $\theta(\log_2(2^k)) = \theta(k)$ et $\theta(\log_2(2^{k+1})) = \theta(k+1) = \theta(k)$.
 - $k < \log_2(n) < k+1$ donc complexité $\theta(\log(n))$.
- L'algorithme est **logarithmique**.

A vous de jouer !

Modifier pour renvoyer le plus grand i tel que $T[i] = e$
(ou -1 si $e \notin T$).

DÉBUT

deb, fin $\leftarrow$ 0, taille(T)-1

tant que deb $\leq$ fin **faire**

mil $\leftarrow$ (deb+fin) div 2

si T[mil] $< e$ **alors**

deb $\leftarrow$ mil+1

sinon

fin $\leftarrow$ mil-1

fin si;

fin tq;

Renvoyer

cond(deb < taille(T) et T[deb]=e,
deb, -1)

FIN

0 ... deb _i - 1		fin _i + 1 ... n - 1	
$< e$	...	$\geq e$	

DÉBUT

deb, fin $\leftarrow$ 0, taille(T)-1

tant que deb $\leq$ fin **faire**

mil $\leftarrow$ (deb+fin) div 2

si T[mil] $\leq e$ **alors**

deb $\leftarrow$ mil+1

sinon

fin $\leftarrow$ mil-1

fin si;

fin tq;

Renvoyer

cond(fin $\geq$ 0 et T[fin]=e, fin, -1)

FIN

0 ... deb _i - 1		fin _i + 1 ... taille(T) - 1	
$\leq e$	...	$> e$	

Plan

- 1 Généralités
- 2 Structures linéaires
- 3 Arrêt d'un algorithme
- 4 Validité d'un algorithme
- 5 Complexité en temps d'un algorithme
- 6 Algorithmes de recherche dans un tableau
- 7 Algorithmes de tri**

Motivation

Trier permet de traiter plus efficacement ensuite.

On étudie 2 sortes de tris de tableaux 1D :

- par valeurs : les données à trier ne peuvent prendre que certaines valeurs : tri en $\Omega(n)$
- par comparaison : les données sont quelconques : tri en $\Omega(n \log n)$

Notation

n : taille du tableau à trier

Borne inférieure d'un tri de tableau

- Pour trier un tableau, il faut parcourir toutes ses cases.
- Donc sa complexité est au moins n , c'est à dire $\Omega(n)$.
- Il est possible de faire $\theta(n)$, par exemple lorsque les données ne peuvent prendre que certaines valeurs.
- L'ordre de complexité n est donc une **borne inférieure** pour un tri de tableau.

Tri par valeurs

PROCÉDURE : TriValeurs (ES

T : tableau de nombre de taille n,

E p : entier)

SPÉCIFICATIONS : Les valeurs de T sont entre 0 et p-1. Trie T.

Variable :

Cpt : tableau de nombre de taille p

j, k : entier

DÉBUT

Initialiser Cpt avec des 0

pour i de 0 à n-1 **faire**

 Cpt[T[i]] $\leftarrow$ Cpt[T[i]] + 1

fin pour;

j, k $\leftarrow$ 0, 0

tant que j < n **faire**

si Cpt[k] = 0 **alors**

 k $\leftarrow$ k+1

sinon

 T[j] $\leftarrow$ k ; j $\leftarrow$ j+1 ;

 Cpt[k] $\leftarrow$ Cpt[k]-1

fin si;

fin tq;

FIN

Complexité : $\theta(p + n)$ ou $\theta(n)$ si $p = O(n)$.

A vous de jouer ! Trace pour $p=4$ et $T=[1, 0, 3, 2, 2, 2]$

DÉBUT

Initialiser Cpt avec des 0

pour i de 0 à $n-1$ **faire**

$Cpt[T[i]] \leftarrow Cpt[T[i]] + 1$

fin pour;

$j, k \leftarrow 0, 0$

tant que $j < n$ **faire**

si $Cpt[k] = 0$ **alors**

$k \leftarrow k+1$

sinon

$T[j] \leftarrow k ; j \leftarrow j+1 ;$

$Cpt[k] \leftarrow Cpt[k]-1$

fin si;

fin tq;

FIN

i	j	k	Cpt	T
Initialisation :				
?	?	?	[0,0,0,0]	[1,0,3,2,2,2]
Boucle pour :				
0	?	?	[0,1,0,0]	[1,0,3,2,2,2]
1	?	?	[1,1,0,0]	[1,0,3,2,2,2]
2	?	?	[1,1,0,1]	[1,0,3,2,2,2]
Après la boucle pour :				
?	0	0	[1,1,3,1]	[1,0,3,2,2,2]
Boucle tant que :				
?	1	0	[0,1,3,1]	[0,0,3,2,2,2]
?	1	1	[0,1,3,1]	[0,0,3,2,2,2]
?	2	1	[0,0,3,1]	[0,1,3,2,2,2]
A la fin de l'algorithme :				
?	6	3	[0,0,0,0]	[0,1,2,2,2,3]

Borne inférieure d'un tri par comparaison

- Dans le cas général, on doit comparer les valeurs entre elles pour les ordonner.
- Nombre de comparaisons nécessaires : au moins $n \log n / 4$ (preuve dans le cours) donc complexité $\Omega(n \log n)$.
- Il est possible de faire $\theta(n \log n)$, par exemple avec le tri fusion.
- L'ordre de complexité $n \log n$ est donc une **borne inférieure** d'un tri de tableau **par comparaison**.

PROCÉDURE : TriBulles (ES T : tableau de nombres de taille n)

SPÉCIFICATIONS : Trie le tableau T par ordre croissant.

DÉBUT

```
pour i de 0 à n-2 faire  
  pour j de n-1 à i+1 par pas de -1 faire  
    si T[j]<T[j-1] alors  
      T[j],T[j-1] ← T[j-1],T[j]  
    fin si;  
  fin pour;  
fin pour;
```

FIN

Complexité : $\theta(n^2)$: ce tri n'est pas optimal.

A vous de jouer ! Trace pour $T=[5,6,2,4,3,1]$

PROCÉDURE : TriBulles (ES)

T : tableau de nombres de taille n)

SPÉCIFICATIONS : Trie le tableau
T par ordre croissant.

DÉBUT

```
pour i de 0 à n-2 faire  
  pour j de n-1 à i+1  
    par pas de -1 faire  
      si  $T[j] < T[j-1]$  alors  
         $T[j], T[j-1] \leftarrow$   
           $T[j-1], T[j]$   
      fin si;  
    fin pour;  
  fin pour;
```

FIN

<i>i</i>	<i>j</i>	<i>T</i>
?	?	[5,6,2,4,3,1]
0	5	[5,6,2,4, 1,3]
0	4	[5,6,2, 1,4 ,3]
0	3	[5,6, 1,2 ,4,3]
0	2	[5, 1,6 ,2,4,3]
0	1	[1,5 ,6,2,4,3]
Le plus petit est bien placé.		
1	5	[1,5,6,2, 3,4]
1	4	[1,5,6, 2,3 ,4]
1	3	[1,5, 2,6 ,3,4]
1	2	[1, 2,5 ,6,3,4]
Les 2 plus petits sont bien placés. A la fin :		
4	5	[1,2,3,4, 5,6]

Preuve du tri à bulles 1/2

PROCÉDURE : TriBulles (ES)

T : tableau de nombres de taille n)

SPÉCIFICATIONS : Trie le tableau
T par ordre croissant.

DÉBUT

```
pour i de 0 à n-2 faire
  pour j de n-1 à i+1
    par pas de -1 faire
      si T[j] < T[j-1] alors
        T[j], T[j-1] ←
          T[j-1], T[j]
      fin si;
    fin pour;
  fin pour;
```

FIN

- **Arrêt** : il y a un nombre fini d'itérations (chaque *pour* parcourt un nombre fini de valeurs) qui se terminent, donc **l'algorithme se termine**.

- **Complexité** : chaque itération fait entre 4 et 12 opérations et le nombre d'itérations vaut

$$\begin{aligned} \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 &= \sum_{i=0}^{n-2} n-1-i \\ &= \sum_{i'=1}^{n-1} i' \\ &= \frac{n(n-1)}{2} \\ &\in \theta(n^2) \end{aligned}$$

donc **la complexité est** $\theta(n^2)$.

Preuve du tri à bulles 2/2

PROCÉDURE : TriBulles (ES)

T : tableau de nombres de taille n)

SPÉCIFICATIONS : Trie le tableau
T par ordre croissant.

DÉBUT

```
pour i de 0 à n-2 faire
  pour j de n-1 à i+1
    par pas de -1 faire
      si T[j] < T[j-1] alors
        T[j], T[j-1] ←
          T[j-1], T[j]
      fin si;
    fin pour;
  fin pour;
```

FIN

Validité (preuve complète cf cours) :

- **Invariant** : Inv_i : "à la fin de l'itération i du 1^{er} pour, les $(i+1)$ plus petites valeurs sont bien placées."
- **Initialisation** : avant la 1^{ère} itération, les 0 premières cases sont triées donc **Inv_0 est vrai.**
- **Hérédité** : on suppose Inv_i vraie et qu'il existe une itération $i+1$. La plus petite valeur parmi $T[i+1..n-1]$ est échangée jusqu'à $T[i+1]$ donc **Inv_{i+1} est vrai.**
- **Conclusion** : A la fin, $i=n-2$: les $n-1$ plus petites valeurs sont à leur place donc la dernière aussi : **l'algorithme est valide.**

Principe

- Algorithme de type "Diviser pour mieux régner" :
 - On trie chaque moitié du tableau
 - On fusionne les deux moitiés triées.
- Il faut 2 algorithmes : l'algorithme de tri et l'algorithme de fusion

Algorithme de fusion

PROCÉDURE : fusion (ES T : tableau de nombre, E deb1, fin1, fin2 : entier)

SPÉCIFICATIONS : T[deb1..fin1] et T[fin1+1..fin2] triés. Trie T[deb1..fin2].

Variable : T1, T2 : tab. de fin1-deb1+1 et fin2-fin1 nombres ; i,j : entiers

DÉBUT

recopier T[deb1..fin1] dans T1 et T[fin1+1..fin2] dans T2

i,j $\leftarrow$ 0,0

pour k de deb1 à fin2 **faire**

si j $\geq$ taille(T2) ou (i < taille(T1) et T1[i] $\leq$ T2[j])

alors

 T[k] $\leftarrow$ T1[i] ; i $\leftarrow$ i+1

sinon

 T[k] $\leftarrow$ T2[j] ; j $\leftarrow$ j+1

fin si ;

fin pour ;

FIN

A vous de jouer ! Trace de fusion([2,6,8,4,9], 0, 2, 4)

PROCÉDURE : fusion (ES T:
t. de nb, E deb1,fin1,fin2: entier)

Variable : i,j: entiers ; T1,T2 :
t. de fin1-deb1+1 et fin2-fin1 nb

DÉBUT

recopier T[deb1..fin1] dans T1

recopier T[fin1+1..fin2] dans T2

i,j $\leftarrow$ 0,0

pour k de deb1 à fin2 **faire**

si j $\geq$ taille(T2) ou

 (i < taille(T1) et

 T1[i] $\leq$ T2[j]) **alors**

 T[k] $\leftarrow$ T1[i] ; i $\leftarrow$ i+1

sinon

 T[k] $\leftarrow$ T2[j] ; j $\leftarrow$ j+1

fin si;

fin pour;

FIN

k	i	j	T1	T2	T
?	0	0	[2,6,8]	[4,9]	[?,?,?,?]
0	1	0	[2,6,8]	[4,9]	[2,?,?,?]
1	1	1	[2,6,8]	[4,9]	[2,4,?,?]
2	2	1	[2,6,8]	[4,9]	[2,4,6,?,?]
3	3	1	[2,6,8]	[4,9]	[2,4,6,8,?]
4	3	2	[2,6,8]	[4,9]	[2,4,6,8,9]

Algorithme de tri fusion

PROCÉDURE : triFusionAux (ES T : tableau de nombre; E deb, fin : entier)
: tableau de nombre

SPÉCIFICATIONS : Trie T entre les indices deb et fin.

Variable : mil : entier

DÉBUT

```

si deb<fin alors
  mil ← (deb+fin) div 2
  triFusionAux(T,deb,mil)
  triFusionAux(T,mil+1,fin)
fusion(T,deb,mil,fin)

```

```
fin si;
```

FIN

A vous de jouer ! Trace de triFusionAux([5,6,2,4,3,1],0,5)

PROCÉDURE : triFusionAux
(ES T : tableau de nombre; E deb, fin :
entier) : tableau de nombre

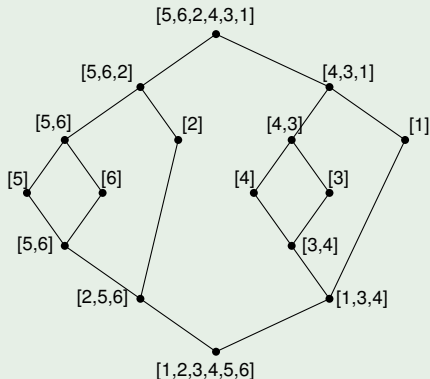
Variable : mil : entier

DÉBUT

si $deb < fin$ **alors**
 $mil \leftarrow (deb + fin) \div 2$
 triFusionAux(T, deb, mil)
 triFusionAux(T, mil + 1, fin)
 fusion(T, deb, mil, fin)

fin si;

FIN



Tri fusion : dernière procédure

Quelles valeurs de deb et fin faut-il choisir pour trier tout le tableau ?

deb=0, fin=taille(T)-1

D'où la dernière procédure du tri fusion :

PROCÉDURE : TriFusion(ES T : tableau de nombres)

SPÉCIFICATIONS : Trie T

DÉBUT

 TriFusionAux(T,0,taille(T)-1)

FIN

Quelques tris célèbres

- **Tri par insertion** : pour k allant de 1 à $\text{taille}(T)-1$, $T[k]$ est inséré parmi $T[0]$ à $T[k-1]$. Adapté lorsque les données sont presque triées.
Complexité : $\theta(n^2)$.
- **Tri par sélection** : pour k allant de 0 à $\text{taille}(T)-2$, on cherche le plus petit élément parmi $T[k]$ à $T[\text{taille}(T)-1]$ et on l'échange avec $T[k]$. Adapté pour de petits ensembles de données.
Complexité : $\theta(n^2)$.
- **Tri par tas** : On structure les données dans un arbre binaire dont les nœuds sont partiellement ordonnés ("tas"). Le sommet de l'arbre est le plus grand élément, on l'extrait et on refait un tas à partir des nœuds restants.
Complexité : $\theta(n \log n)$.
- **Tri rapide ou tri par pivot** : on sépare un tableau entre les éléments plus petits et plus grand qu'un certain élément du tableau ("pivot") et on trie chaque moitié récursivement.
Complexité : $\theta(n^2)$ dans le pire des cas, $\theta(n \log n)$ en moyenne.

A vous de jouer !

Donner les étapes de tri du tableau [1, 3, 5, 2, 9, 4, 0] par sélection et par insertion.

- **Sélection :**

"pour k allant de 0 à $\text{taille}(T)-2$,
on cherche le plus petit élément
parmi $T[k]$ à $T[\text{taille}(T)-1]$
et on l'échange avec $T[k]$."

[1, 3, 5, 2, 9, 4, 0]
[0, 3, 5, 2, 9, 4, 1]
[0, 1, 5, 2, 9, 4, 3]
[0, 1, 2, 5, 9, 4, 3]
[0, 1, 2, 3, 9, 4, 5]
[0, 1, 2, 3, 4, 9, 5]
[0, 1, 2, 3, 4, 5, 9]

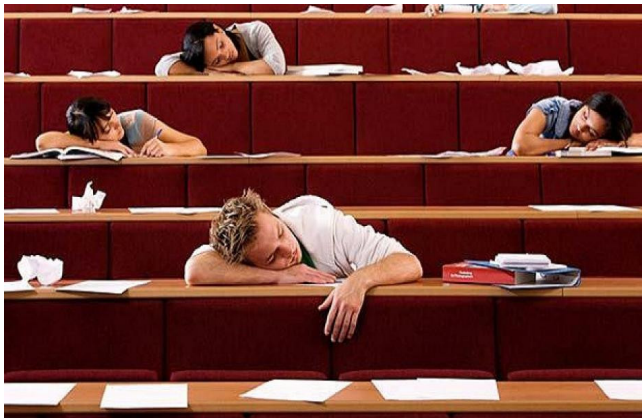
- **Insertion :**

"pour k allant de 1 à $\text{taille}(T)-1$,
 $T[k]$ est inséré parmi $T[0]$ à $T[k-1]$."

[1, 3, 5, 2, 9, 4, 0]
[1, 3, 5, 2, 9, 4, 0]
[1, 3, 5, 2, 9, 4, 0]
[1, 2, 3, 5, 9, 4, 0]
[1, 2, 3, 5, 9, 4, 0]
[1, 2, 3, 4, 5, 9, 0]
[0, 1, 2, 3, 4, 5, 9]

Deux propriétés des tris

- Un tri est **stable** s'il laisse les éléments égaux dans le même ordre.
 - utile pour trier successivement selon plusieurs critères ;
 - exemples : bulles, insertion, fusion ;
 - sinon, possibilité de mémoriser l'emplacement initial des éléments.
- Un tri est **en place** s'il ne nécessite pas d'espace supplémentaire important et modifie directement la structure à trier.
 - important si on dispose de peu de mémoire ;
 - exemples : bulles, sélection, insertion, tas.



Merci pour votre attention !