# Algorithmique 1

Sylvain Daudé

HAI101I / HA8203I

# Organisation

- 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%

# Cadre du cours

## Objet du cours

Ce cours porte sur les algorithmes récursifs 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

# Plan

# Algorithme

## Définition d'un algorithme

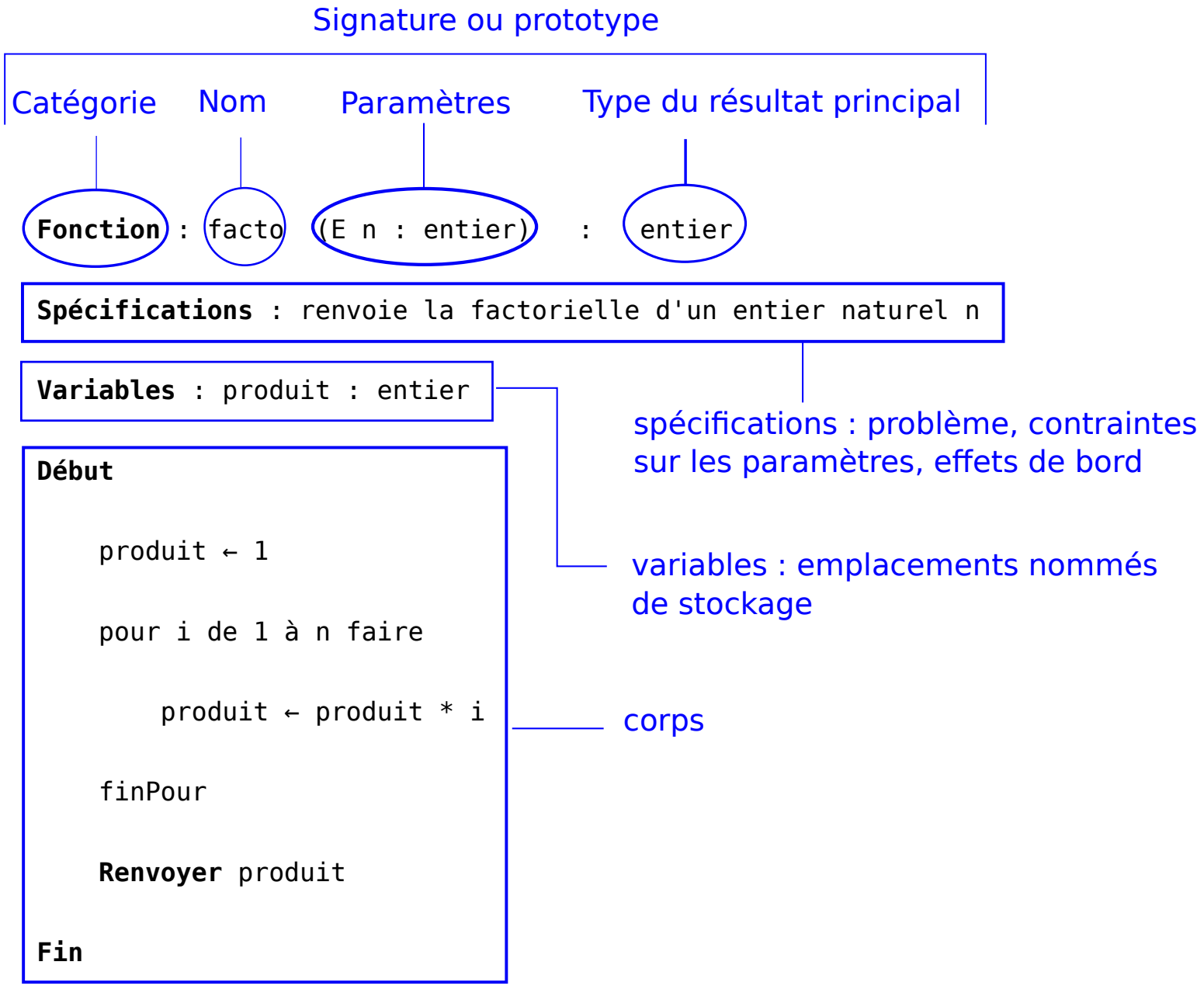
Algorithme : séquence de calcul bien définie, en réponse à un problème, selon un nombre fini d'opérations.

- 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



# Algorithme itératif et récursif

## 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.

## Attention

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 $n! = 1 \times 2 \times ... \times n$ 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 $n! = 1 \times 2 \times ... \times n$ avec $0! = 1$

---

**DÉBUT**
    **si** $n = 0$ **alors**
        **Renvoyer** 1
    **sinon**
        **Renvoyer** $n \times FactRec(n-1)$
    **fin si;**
**FIN**

---

# Définitions

- **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).
  - **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, x$\leftarrow$3.
- **Expression** : tout ce qui a une valeur : formule contenant constantes, noms de variables, paramètres, appels d'algorithmes et opérations ayant une valeur. A pour valeur le résultat de la formule.
- **Instruction** : une ligne du corps de l'algorithme (affectation, appel, renvoi d'une valeur) ou un bloc de lignes (si, pour, tant que).

# A vous de jouer !

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

---

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

---

**Variable :** res : nombre

---

**DÉBUT**
    res ← 1
    **pour** `i de 1 à n` **faire**
        res ← 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 * PuisRec(x,n-1)
    **fin si;**
**FIN**

---

# A vous de jouer !

## Correction

- types : nombre, entier
- constantes : 0, 1
- variables : res
- appels : PuisRec(x,n-1) (appel récursif)
- opérations : res←1 (affectation), res*x, res←res*x, n=0, n-1, x * PuisRec(x,n-1)
- expressions : tous les précédents sauf affectations, x, n (paramètres), i (indice de boucle)
- instructions : affectations, lignes Renvoyer, blocs Pour et Si

## Attention !

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

# Opérateurs en pseudo-code et python

- **Opérateurs de calcul** : + - * / ˆ (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** : ˆ 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 div 3 + 7 mod 5 entier, 4
- 5.0 mod 2 erreur
- 1=5*2 booléen, false
- true et (true ou false) booléen, true
- true ou 3 erreur
- true ou (5/0=1) booléen, true
- cond(5 mod 2 = 1, 8, 15) entier, 8
- cond(5 div 2 = 1, true, 0) erreur
- cond(5 div 2 = 2, cond(5 mod 2 = 0, 3, 8), 11) entier, 8

# Instruction conditionnelle

## Définition

Une *instruction 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 ← 0
    **si** i<3 **alors**
        i ← 5
    **sinon si** i≥4 **alors**
        i ← 7
    **fin si;**
    afficher i
**FIN**

Correction : 5

**Variable :** i : entier

**DÉBUT**
    i ← 0
    **si** i<3 **alors**
        i ← 5
    **fin si;**
    **si** i≥4 **alors**
        i ← 7
    **fin si;**
    afficher i
**FIN**

Correction : 7

# Boucle tant que

## Définition

Une boucle *tant que* est une instruction qui 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 ← 0
    **tant que** `2*i+1<10` **faire**
        i ← i+1
    **fin tq;**
    Renvoyer i
**FIN**

Correction : 5

**Variable :** i : entier

**DÉBUT**
    i ← 5
    **tant que** `i` $\neq$ `0` **faire**
        i ← i-2
    **fin tq;**
    Renvoyer i
**FIN**

Correction : Rien ! (boucle infinie)

# Boucle pour

## Définition

Une boucle *pour* est une instruction qui 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

# Structures linéaires

## 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 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] ← 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*i

**fin pour;**

**Renvoyer** T

**FIN**

---

# A vous de jouer ! Compléter la fonction.

---

**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**

---

# Tableaux en Python

- 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)

# Listes

## 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 ← cons (1, cons (2, cons(3, [] ))) ——> L vaut [1, 2, 3], tête 1, queue [2,3]
- A ← tête(queue(queue(L))) ——> A vaut 3
- Test si L a au moins 2 éléments ——> non(estVide(L)) et non estVide(queue(L))

# A vous de jouer !

## Ecrire la valeur de la liste L après les étapes suivantes

| | |
|---|---|
| Variable : L : liste d'entiers | ? |
| L ← cons(2,[]) | [2] |
| L ← cons(tête(L),queue(L)) | [2] |
| L ← cons(3,L) | [3,2] |
| Pour i de 1 à 5 faire | |
| L ← cons(i,L) | |
| finPour | [5,4,3,2,1,3,2] |
| L ← queue(queue(L)) | [3,2,1,3,2] |
| L ← cons(L[2], []) | erreur : L[2] illégal |
| L ← 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 ← 0, L ;
    **tant que** `non estVide(M)` **faire**
        cpt, M ← 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 :** cptEgauxLIte (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 ← L,0 ;
    tant que non estVide(M) faire
        si tête(M)=e alors
            cpt ← cpt+1
        fin si;
        M ← 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
```

# Listes en Python

## 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

# Piles

## 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) —> P vaut [1, 2] (sommet : 2)
- A $\leftarrow$ depiler(P) —> 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 $\leftarrow$ [] | [] |
| 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 $\leftarrow$ 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 ← [ ] ;
**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 :** invPileIte (ES P : pile d'entier)

---

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

---

**Variable :** Q, R : pile d'entier

---

**DÉBUT**

```
    Q, R ← [ ], [ ] ;
    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**

# Piles en Python

## 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

# Files

## 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) —> F = [1, 2, 3] (tête : 1)
- A $\leftarrow$ defiler(F) —> A vaut 1, F 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 ← [ ] ;
    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**

---

# Files en Python

## 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 F

# Attention

## 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

# Motivation

## 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.

# Arrêt d'un algorithme itératif

## 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.

# Premier exemple de preuve d'arrêt

## Exemple

---

**FONCTION :** F1 (E n : entier) : entier

---

**Variable :** res : entier

---

**DÉBUT**
    res ← 1
    **pour** `i de 1 à n` **faire**
        res ← 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 (E n : entier) : entier

---

**Variable :** p, res : entier

---

**DÉBUT**

p, res $\leftarrow$ n, 0

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

p, res $\leftarrow$ p div 2, res +1

**fin tq;**

**FIN**

---

**Renvoyer** res

- Chaque itération contient 1 comparaison, 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 ?

**FONCTION :**
F (E n : entier) : entier

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

**Variable :** i : entier

**DÉBUT**
    i ← n
    **tant que** `i` $\neq$ `0` **faire**
        i ← i - 10
    **fin tq;**
    **Renvoyer** i
**FIN**

- Chaque itération contient 3 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 $i \geq 10$.

# A vous de jouer ! 2/2

## G se termine-t-il ?

---

**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 6 opérations dont se termine.
- *i* progresse au moins d'une unité à chaque itération, donc finit par dépasser *b*. Comme les itérations s'arrêtent lorsque $i \geq b$, il y a un nombre fini d'itérations.
- L'algorithme se termine.

# Terminaison d'un algorithme récursif

## 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 de preuve d'arrêt

## 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*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.

# A vous de jouer ! 1/2

## 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 élémentaires.
- $n \in \mathbb{N}$ augmente d'une unité à chaque appel donc n'atteint jamais 0.
- Modification : remplacer $F(n+1)$ par $F(n-1)$.

# A vous de jouer ! 2/2

## 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 mod 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, donc finit par atteindre 1 ou un diviseur plus grand. Il y a donc un nombre fini d'appels récursifs.
- L'algorithme se termine.

# Plan

# Motivation

## Un bon algorithme...

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

# Validité d'un algorithme itératif

## Vocabulaire

**Trace d'un algorithme** : suivi des valeurs des variables sur un exemple d'exécution.
**Invariant de boucle** : propriété censée être vraie à chaque itération.

## Méthode

1. chercher un invariant de boucle, souvent à partir d'une trace ;
2. écrire l'invariant à la fin des itérations.

# Premier exemple de justification de validité

**FONCTION :**
F1 (E n : entier) : entier

**SPÉCIFICATIONS :**
Calcule $n!$ avec $n \in \mathbb{N}$

**Variable :** res : entier

**DÉBUT**
    res ← 1
    **pour** `i de 1 à n` **faire**
        res ← res * i
    **fin pour;**
    **Renvoyer** res
**FIN**

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

1. **Invariant proposé** : $Inv_i$ : "$res_i = i!$".
2. A la fin du pour, $i = n$ donc, d'après l'invariant, la valeur renvoyée est $res_n = n!$ : **l'algorithme est valide**.

# Deuxième exemple de justification de validité

**FONCTION :**
Cube (E n : entier) : entier

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

**Variable :** A, B, C, Z : entier

**DÉBUT**
A, B, C, Z ← 1, 0, n, 0 ;
**tant que** C > 0 **faire**
Z ← Z + A + B ;
B ← B + A + A + 1 ;
A, C ← A+3, C-1 ;
**fin tq;**
**Renvoyer** Z
**FIN**

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

**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** :

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

**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 ← T[0]
**pour** `i de 1 à taille(T)-1` **faire**
res ← res+T[i]
**fin pour;**
**Renvoyer** res
**FIN**

---

1. **Invariant** : $Inv_i$ : "$res_i$ est la somme des valeurs $T[0]$ à $T[i]$".
2. A la fin du pour, l'algorithme renvoie $res_{n-1} = T[0] + \ldots + T[n-1]$ : **l'algorithme est valide**.

# Validité d'un algorithme récursif

## Vocabulaire

- **équation de récurrence** : égalité reliant le résultat de l'algorithme avec celui des appels récursifs ;

## Méthode

1. "lire" les équations de récurrence sur l'algorithme ;
2. énoncer le résultat censé être renvoyé ;
3. prouver le résultat par récurrence ou induction.

# 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*F2(n-1)
    **fin si;**
**FIN**

---

1. **Équation de récurrence** :
   $F2(0) = 1$ et $F2(n) = n \times F2(n-1)$ pour $n > 0$.
2. **Résultat de l'algorithme** :
   $P_n$ : "$F2(n) = n!$"
3. **Preuve** :
   - **Initialisation** : Pour $n = 0$, on a $F2(0) = 1 = 0!$ donc $P_0$ est vraie.
   - **Récurrence** : Pour $n>0$, on suppose $P_{n-1}$ vrai, c'est à dire $F2(n-1) = (n-1)!$ Alors $F2(n) = n \times F2(n-1) = n \times (n-1)! = n!$. Donc $P_n$ est vrai.
   - **Conclusion** : pour tout $n \in \mathbb{N}$, $F2(n) = n!$
4. **L'algorithme est valide.**

# A vous de jouer ! Compléter la preuve

**FONCTION :**
F4 (E n : entier) : entier)

**SPÉCIFICATIONS :** Renvoie la somme des entiers de 1 à $n$, avec $n \in \mathbb{N}^*$

**DÉBUT**
    **Renvoyer** cond(n=1, 1, n+F4(n-1))
**FIN**

- **Équations de récurrence** :
  si $n = 1$ alors $F4(n) = 1$
  et si $n > 1$ alors $F4(n) = n + F4(n-1)$
- **Résultat** :
  $P_n$ : "$F4(n) = 1 + \ldots + n$".
  - **Initialisation** : Pour $n = 1$, $F(n) = 1$ qui est bien la somme de 1 à 1.
    Donc $P_1$ est vrai.
  - **Récurrence** : Pour $n > 1$,
    on suppose $P_{n-1}$ **vrai** :
    $F4(n-1) = 1 + \ldots + n - 1$.
    Comme $F4(n) = n + F4(n-1)$,
    $F4(n) = 1 + \ldots + n$.
    Donc $P_n$ est vrai.
  - **Conclusion** : pour tout $n \in \mathbb{N}$, $F4(n)$ renvoie la somme des entiers de 1 à $n$.
- **L'algorithme est valide.**

# Plan

# Motivation et problématique

## 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).

## Problématiques

- Comment compter les opérations ?
  - Sur machine, une multiplication et une addition ont des durées différentes.
- Et si le nombre d'opérations est variable selon les paramètres ?
  - Il faut plus d'opérations pour traiter une grande image qu'une petite.
- Qu'est-ce qu'un nombre **raisonnable** d'opérations ?
  - 10000 opérations pour calculer un coup d'échecs, c'est très raisonnable...
  - mais pour afficher "bonjour" c'est énorme !

# Comptage des opérations

## Simplifications

- Toutes les opérations élémentaires (affectation, addition, appel...) sont comptées à égalité ;
- on suppose que le nombre d'opérations dépend d'un unique entier $n$ appelé **taille de l'entrée** ;
- si, pour un même $n$, le nombre d'opérations dépend aussi de la configuration des données, on étudie uniquement celle qui génère le plus d'opérations, appelée **pire cas** ;
- on note $T_n$ le nombre d'opérations obtenu. C'est une suite.

## Définitions

- Le nombre d'opérations est appelé **complexité en temps** de l'algorithme ou, par commodité, **complexité** ;
- il existe aussi la complexité en espace, qui mesure l'espace occupé par les variables et sort du cadre de ce cours.

# A vous de jouer (1)

## Calculer $T_n$

---

**FONCTION :** F1 (E n : entier) : entier

---

**Variable :** res : entier

---

**DÉBUT**
    res ← 1
    **pour** `i de 1 à n` **faire**
        res ← res * i
    **fin pour;**
    **Renvoyer** res
**FIN**

---

- 1 affectation + n itérations contenant 1 affectation + 1 multiplication
- Donc $T_n = 2n + 1$

# A vous de jouer (2)

## Calculer $T_n$ dans le pire cas (distinguer $n$ pair et $n$ impair)

**FONCTION :**
F4 (E n : entier) : entier

**Variable :** som : nombre

**DÉBUT**
```
    som ← 0
    pour i de 1 à n faire
        si i mod 2 = 1 alors
            som ← som + i*i
        sinon
            som ← 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
- donc 1 + (5+4+5+4+...) (n termes)
- Si n est pair : $1 + 5\frac{n}{2} + 4\frac{n}{2} = 4,5n + 1$
- Si n est impair : $1 + 5\frac{n+1}{2} + 4\frac{n-1}{2} = 4,5n + 1,5$
- Pire cas : $T_n = 4,5n + 1,5$

# Comptage pour un algorithme récursif

## Problématique

- Lorsqu'on appelle un algorithme récursif, il effectue des opérations.
- Puis il s'appelle, et les appels effectuent des opérations.
- Puis ces appels font des appels, qui effectuent des opérations...

## Solution

- $T_n$ = opérations de l'algorithme + opérations faites dans ses appels
- $T_n$ est une suite définie par une **équation de récurrence**, il reste à l'exprimer directement en fonction de $n$.

# Exemple de comptage pour un algorithme récursif

---

**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*F2(n-1)
    **fin si;**
**FIN**

---

- **Cas de base :** si n=0,
  1 comparaison : $T_0 = 1$
- **Équation de récurrence** : si n>0 :
  1 comparaison, 1 multiplication,
  1 appel, 1 soustraction + les
  opérations de F2(n-1) : $T_n = 4 + T_{n-1}$
- **Résolution :** $T_n$ est une suite
  arithmétique : $T_n = 4n + 1$.

# Un exemple où la taille de l'entrée est divisée

**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 mod 2 = 1 alors
        Renvoyer false
    sinon
        Renvoyer F5(n div 2)
    fin si;
FIN
```

- **Cas de base :** $T_1 = 1$
  et si $n$ est impair >1, $T_n = 3$
- **Equation de récurrence :**
  si $n$ est pair $> 1$ : 5 opérations + celles de F5(n div 2) :
  $T_n = 5 + T_{n \ div \ 2}$
- **Pire cas :** n>1 est toujours pair, c'est à dire qu'au départ $n = 2^k$, ou encore $k = log_2(n)$.
- **Résolution :** Depuis le cas de base, on ajoute 5 autant de fois que $n = 2^k$ peut être divisé par 2, c'est à dire $k$ fois :
  $T_n = 1 + 5k = 1 + 5log_2(n)$

# A vous de jouer !

## Complexité en soustractions ? Compléter le raisonnement.

---

**FONCTION :** estPair (E n : entier) : booléen

---

**DÉBUT**
    **Renvoyer** cond(n=0, true, cond(n=1, false, estPair(n-2)))
**FIN**

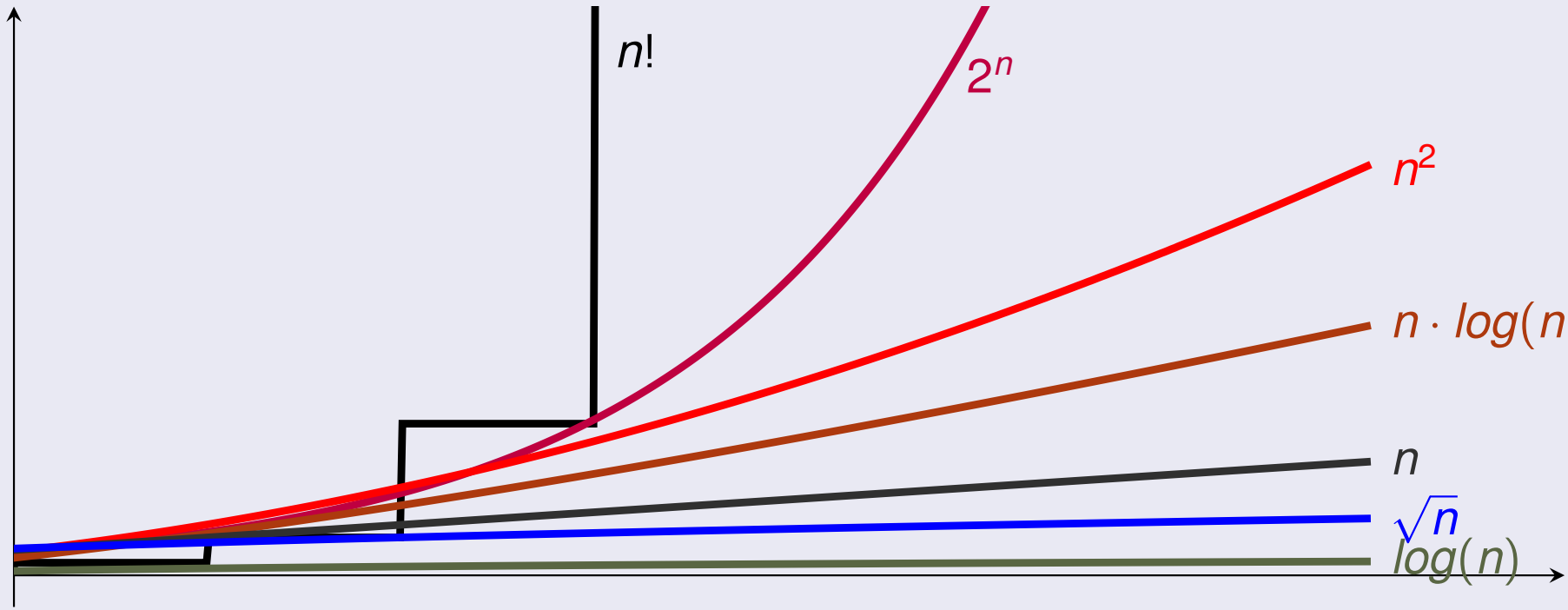
---

- **Cas de base :** $T_0 = T_1 = 0$
- **Equation de récurrence :** si n>1 alors $T_n = 1 + T_{n-2}$
- **Résolution :** Valeurs de $T_n$ : 0, 0, 1, 1, 2, 2, ....
  Généralisation : $T_n = n \; div \; 2$

## Suites, comportement asymptotique

- $T_n$ est une **suite numérique**, notée $T_n$, **positive** et souvent **croissante**.
- On la caractérise en comparant sa croissance à celles de suites de référence positives croissantes ("comparaison asymptotique").

## Croissances comparées des suites de référence



# Domination de suites numériques

## Idée générale

On divise $T_n$ par une suite de référence $R_n$ et on fait tendre $n$ vers $+\infty$.

- si $T_n/R_n$ tend vers 0 alors $T_n$ est **strictement dominée** par $R_n$.
  On note $T_n = o(R_n)$.
- si $T_n/R_n$ tend vers $+\infty$ alors $T_n$ **domine strictement** $R_n$.
  On note $T_n = \omega(R_n)$.
- si $T_n/R_n$ tend vers un nombre strictement positif,
  ou est borné entre deux nombres strictement positifs,
  alors $T_n$ est **équivalente** à $R_n$ ou **de l'ordre de complexité** $R_n$.
  On note $T_n = \theta(R_n)$.
- sinon, $T_n$ n'est **pas comparable** à $R_n$. C'est rare.

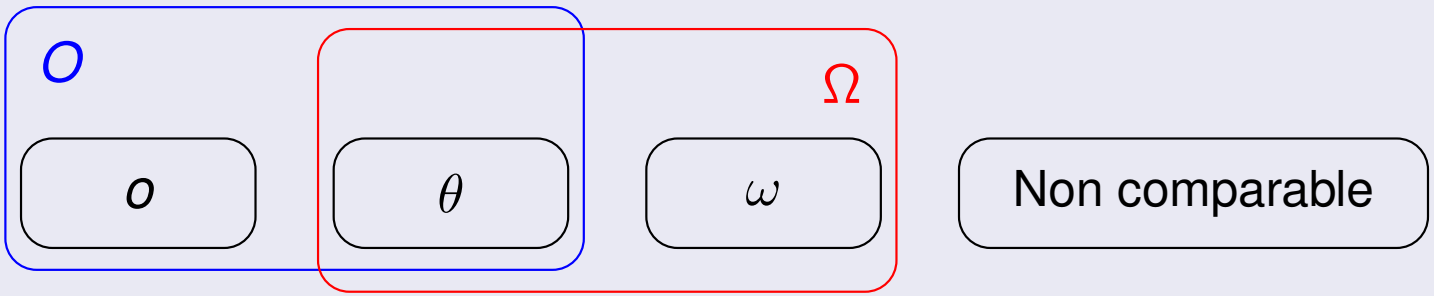
## Attention

"Equivalentes" ne veut pas dire "identiques" : si $T_n = 1000000R_n$ ou $T_n = R_n + 1000000$ alors $T_n$ et $R_n$ sont équivalentes !

# Domination de suites numériques - suite

## Vocabulaire

Domination stricte ou équivalence = **domination au sens large**.
On utilise la majuscule de la domination stricte : $T_n = O(R_n)$ ou $T_n = \Omega(R_n)$.



# Complexités comparées

## Catégories de complexité

Un algorithme est dit :

- **à coût constant** si $T_n = \theta(1)$ ;
  exemple : formule
- **logarithmique** si $T_n = \theta(log(n))$ ;
  exemple : recherche dans un tableau trié
- **linéaire** si $T_n = \theta(n)$ ;
  exemple : recherche du maximum d'un tableau
- **semi-linéaire** si $T_n = \theta(nlog(n))$ ;
  exemple : tri fusion d'un tableau
- **quadratique** si $T_n = \theta(n^2)$ ;
  exemple : tri bulle d'un tableau
- **polynomial** si $T_n = O(n^a)$ avec $a$ entier naturel ;
  exemple : recherche d'un chemin entre deux sommets d'un graphe
- **exponentiel** si $T_n = \theta(a^n)$ avec $a > 1$.
  exemple : parcours de tous les chemins d'un graphe

# A vous de jouer !

## Trouver une suite de référence équivalente et la catégorie de complexité

- $T_n = 3n^3 - 5n + 2$
  On a $T_n/n^3 \to 3$ donc $T_n = \theta(n^3)$ : polynomial
- $T_n = 2^{n+1} - 8n^3$
  On a $T_n/2^n \to 2$ donc $T_n = \theta(2^n)$ : exponentiel
- $T_n = 3nlog(n^2) + 5$
  On a $T_n = 6nlog(n) + 2$ donc $T_n = \theta(nlogn)$ : semi-linéaire
- $T_n = 2T_{n-1}$
  $T_n$ est une suite géométrique : $T_n = 2^n T_0 = \theta(2^n)$ : exponentiel
- $T_n = T_{n-1} + n$
  $T_n = n + ... + 1 + T_0 = \frac{n(n+1)}{2} + T_0 = \theta(n^2)$ : quadratique
- $T_n = 3n + 1$ si $n$ est pair et $T_n = 2n$ si $n$ est impair
  $T_n/n$ tend vers 3 pour les $n$ pairs et vers 2 pours les $n$ impairs, donc $T_n = \theta(n)$ : linéaire.

# Complexités comparées - suite

## Propriétés

- ordre de domination stricte des suites de référence :
  $1, log(n), \sqrt{n}, n, nlog(n), n^2, n^3, 2^n, 3^n, n!$
- 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 **transitives** :
  si $a = o(b)$ et $b = o(c)$ alors $a = o(c)$ et de même pour $O, \omega, \Omega, \theta$.
- $o, O, \omega, \Omega, \theta$ sont **multiplicatives** et **additives** sur les suites >0 :
  si $a = o(a')$ et $b = o(b')$ alors $ab = o(a'b')$ et $a + b = o(a' + b')$
  et de même pour $O, \omega, \Omega, \theta$.

# Algorithmes itératifs + boucles pour

## Propriété

- Première analyse : si des boucles *pour* imbriquées contiennent un nombre borné d'opérations, alors leur complexité est du même ordre que le nombre d'itérations.
- Deuxième analyse : pour calculer les complexités exactes, on a :
$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} ; \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} ; \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

## Exemple

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

- Première analyse : $i$ prend $n$ valeurs, $j$ prend $i = O(n)$ valeurs, d'où $nO(n) = O(n^2)$ opérations.
- Deuxième analyse : nb d'op :
$$\sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2)$$

# A vous de jouer !

## Complexités en fonction de n ?

1. ```
   Pour i de 1 à n faire
       Pour j de 1 à n faire
           som ← som+1
   ```
2. ```
   Pour i de 1 à n faire
       Pour j de 1 à n faire
           Pour k de 1 à n faire
               som ← som+1
   ```
3. ```
   Pour i de 1 à n faire
       Pour j de i à n faire
           som ← som + 1
   ```
4. ```
   i ← 0 ; TantQue i²<n faire
       i ← i+1
   ```

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

# Plan

# Algorithme de recherche linéaire

## 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**

# Complexité de la recherche linéaire

```
DÉBUT
    i ← 0
    tant que i<n et T[i]≠e faire
        i ← i+1
    fin tq;
    si i=taille(T) alors
        Renvoyer -1
    sinon
        Renvoyer i
    fin si;
FIN
```

**Complexité :** pire des cas pour $e \notin T$ en notant $n$=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 ← 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**

# A vous de jouer !

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

```
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
```

| $i$ | $deb_i$ | $mil_i$ | $fin_i$ | $T[deb_i..fin_i]$ |
|---|---|---|---|---|
| 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 ← 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
```

| $i$ | $deb_i$ | $mil_i$ | $fin_i$ | $T[deb_i..fin_i]$ |
|---|---|---|---|---|
| 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.

# Preuve d'arrêt

```
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
```

- 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 ← 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
```

- **Notation** : $n$ : taille(T) ;
  $expr_i$ : valeur de expr à la fin de l'itération $i$.
- **Invariant** (admis) :

| $0 \ldots deb_i - 1$ | $\ldots$ | $fin_i + 1 \ldots n - 1$ |
|---|---|---|
| $< e$ | | $\geq e$ |

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

| $0 \ldots deb - 1$ | $deb \ldots n - 1$ |
|---|---|
| $< e$ | $\geq e$ |

  - 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 $n = 2^k$, 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 est le nombre de fois où $2^k$ peut être divisé par 2 jusqu'à arriver à 0, c'est à dire $k + 1 = log_2(n) + 1$.
  - chaque itération est à coût constant 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 $\theta(k)$ et $\theta(k+1)$ donc $\theta(k)$.
  - $k < log_2(n) < k + 1$ donc $\theta(k) = \theta(log_2(n))$.
- 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 ← 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**

| $0 \ldots deb_i - 1$ | | $fin_i + 1 \ldots n - 1$ |
|---|---|---|
| $< e$ | $\ldots$ | $\geq e$ |

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

| $0 \ldots deb_i - 1$ | | $fin_i + 1 \ldots taille(T) - 1$ |
|---|---|---|
| $\leq e$ | $\ldots$ | $> e$ |

# Plan

# Motivation du 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]] ← Cpt[T[i]] + 1
**fin pour;**
j,k ← 0,0
**tant que** `j<n` **faire**
**si** `Cpt[k]=0` **alors**
k ← k+1
**sinon**
T[j] ← k ; j ← j+1 ;
Cpt[k] ← 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]] ← Cpt[T[i]] + 1
    fin pour;
    j,k ← 0,0
    tant que j<n faire
        si Cpt[k]=0 alors
            k ← k+1
        sinon
            T[j] ← k ; j ← j+1 ;
             Cpt[k] ← 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**.

# Tri à bulles

**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] ←
                 T[j-1],T[j]
            fin si;
        fin pour;
    fin pour;
FIN
```

| *i* | *j* | *T* |
|---|---|---|
| ? | ? | [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**] |

# Tri fusion

## 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 ← 0,0

**pour** `k de deb1 à fin2` **faire**

**si** `j>=taille(T2) ou (i<taille(T1) et T1[i]<=T2[j])`

**alors**

T[k] ← T1[i] ; i ← i+1

**sinon**

T[k] ← T2[j] ; j ← 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 ← 0,0
**pour** `k de deb1 à fin2` **faire**
**si** `j>=taille(T2) ou (i<taille(T1) et T1[i]<=T2[j])` **alors**
T[k] ← T1[i] ; i ← i+1
**sinon**
T[k] ← T2[j] ; j ← j+1
**fin si;**
**fin pour;**
**FIN**

| *k* | *i* | *j* | *T*1 | *T*2 | *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)

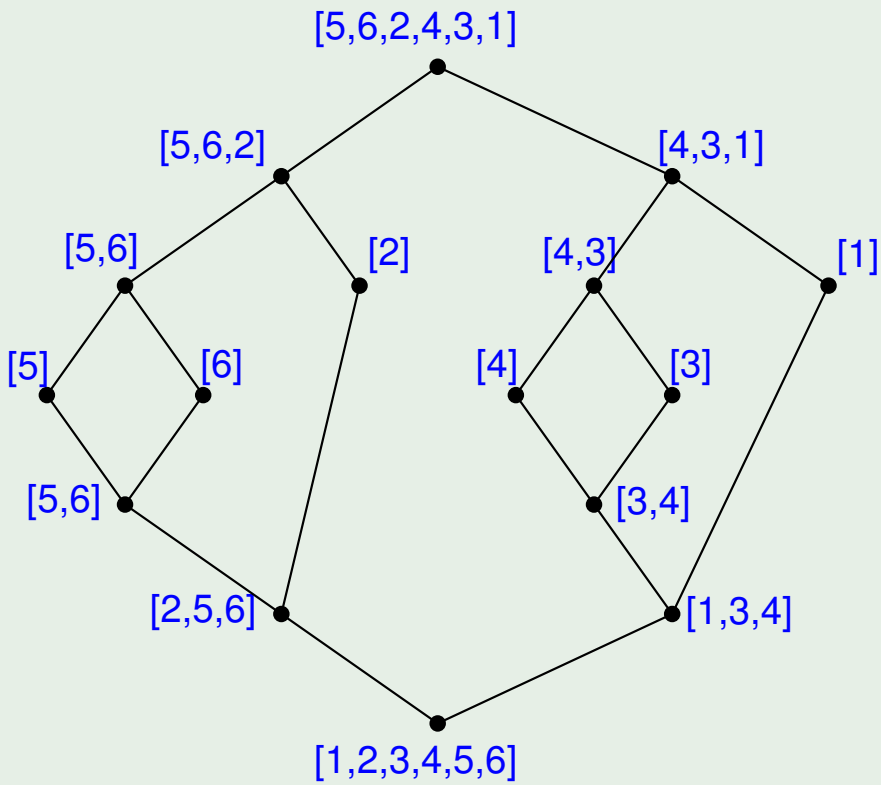
**PROCÉDURE :** triFusionAux
(ES T : tableau de nombre; E deb, fin : entier) : tableau de nombre

**Variable :** mil : entier

**DÉBUT**

```
    si deb<fin alors
        mil ← (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 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 taille(T)-2, on cherche le plus petit élément parmi T[k] à T[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 taille(T)-2,
  on cherche le plus petit élément
  parmi T[k] à T[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** :

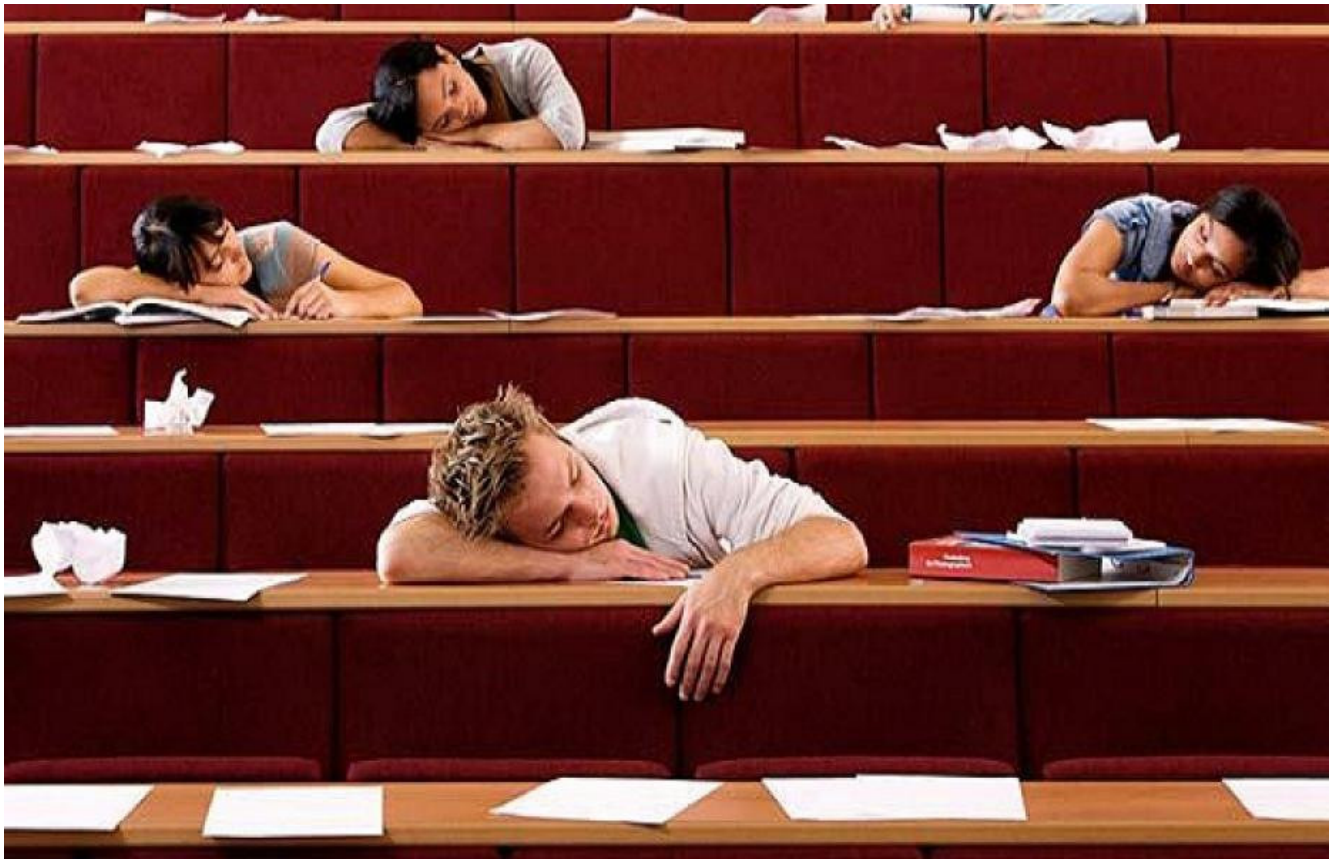
  "pour k allant de 1 à taille(T)-1,
  T[k] est inséré parmi T[0] à T[k-1]."

  [**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.

# Fin du cours



Merci pour votre attention !