

---

# IN 303 – Structure de données

Exercices et Résumés de Cours

Ylène Aboulfath – Coline Gianfrotta – Maël Guiraud  
Franck Quessette – Yann Strozecki – Sandrine Vial (resp. Cours)

---



---

# Avant-Propos

---

Dans ce document vous trouverez 3 parties :

- une partie qui contient les exercices que vous ferez en TD. Chaque exercice fait en présentiel sera noté  $\mathcal{P}$ , chaque exercice qui sera fait en distanciel sera noté  $\mathcal{D}$ ;
- une partie qui contient quelques conseils sur la rédaction des preuves et des algorithmes ainsi que quelques rappels mathématiques utiles pour le cours.
- une partie qui contient les résumés des cours ;



---

# Table des matières

---

<b>Table des matières</b>	<b>iii</b>
<b>1 Exercices</b>	<b>1</b>
1.1 Complexité . . . . .	1
1.2 Les tris . . . . .	12
1.3 Les piles et les files . . . . .	12
1.4 Les arbres . . . . .	16
<b>2 Conseils</b>	<b>19</b>
2.1 Pseudo-Code . . . . .	19
2.2 Mathématiques élémentaires . . . . .	22
2.3 Techniques de preuves . . . . .	23
<b>3 Résumés de cours</b>	<b>25</b>
3.1 Introduction à la complexité des algorithmes . . . . .	25
3.2 Etudes de complexité . . . . .	28
3.3 Les tris . . . . .	30
3.4 Les structures de données abstraites . . . . .	33
3.5 Les arbres . . . . .	35



---

# Chapitre 1

## Exercices

---

### 1.1 Complexité

#### Exercice 1 – P Un simple calcul

1. Donner un algorithme qui calcule la somme des multiples de 3 ou 5 inférieur à  $n$ . Par exemple, si  $n$  vaut 17, alors l'algorithme doit retourner 60.
2. Calculer le nombre d'opérations arithmétiques de votre algorithme en fonction de  $n$ . Pouvez-vous le modifier afin de réduire sa complexité?

#### Correction exercice 1

1. Un algorithme qui calcule la somme des entiers multiples de 3 ou 5 inférieur à  $n$ .

---

**SommeMultiple\_3\_ou\_5( $n$  : entier) : entier**

---

```
Debut
   $i \leftarrow 1$ 
  somme  $\leftarrow 0$ 
  tant que ( $i \leq n$ ) faire
    si  $i \bmod 3 = 0$  ou  $i \bmod 5 = 0$ 
      somme  $\leftarrow$  somme +  $i$ 
     $i \leftarrow i + 1$ 
  fin tant que
Fin
```

2. Le nombre d'opérations arithmétiques de cet algorithme en fonction de  $n$  est le suivant :
  - A chaque itération il y a :
    - 2 divisions (pour calculer  $i \bmod 3$  et  $i \bmod 5$ )
    - 1 addition pour calculer  $somme \leftarrow somme + i$  (cette addition n'est faite que lorsque la condition du Si est vrai. On va donc la compter à chaque fois dans le cas d'une évaluation dans le pire cas.
    - 1 addition pour calculer  $i \leftarrow i + 1$
  - Au total on aura donc 4 opérations arithmétiques par itération.
  - L'algorithme fera  $n$  itérations au totalCet algorithme fera au plus  $4n$  opérations arithmétiques pour faire la somme des entiers multiples de 3 ou 5 inférieurs à  $n$ .

Pour améliorer la complexité on peut essayer d'écrire le problème sous forme mathématique à savoir : qu'il faut faire la somme de tous les multiples de 3 inférieur à  $n$  d'y ajouter la somme de tous les multiples de 5 et d'y retrancher tous les multiples de 15. On peut réécrire cette somme de la façon suivante :

$$Somme = \sum_{i=1}^{\lfloor \frac{n}{3} \rfloor} 3i + \sum_{i=1}^{\lfloor \frac{n}{5} \rfloor} 5i - \sum_{i=1}^{\lfloor \frac{n}{15} \rfloor} 15i$$

$$Somme = 3 \sum_{i=1}^{\lfloor \frac{n}{3} \rfloor} i + 5 \sum_{i=1}^{\lfloor \frac{n}{5} \rfloor} i - 15 \sum_{i=1}^{\lfloor \frac{n}{15} \rfloor} i$$

$$Somme = 3 \times \frac{\lfloor \frac{n}{3} \rfloor \times \lfloor \frac{n}{3} \rfloor + 1}{2} + 5 \times \frac{\lfloor \frac{n}{5} \rfloor \times \lfloor \frac{n}{5} \rfloor + 1}{2} - 15 \times \frac{\lfloor \frac{n}{15} \rfloor \times \lfloor \frac{n}{15} \rfloor + 1}{2}$$

L'algorithme issu de la formule mathématique fait un nombre constant d'opérations arithmétiques quelle que soit la valeur de  $n$ .

### Exercice 2 – P Calcul du coût d'un algorithme

1. Déterminer le nombre d'affectations, en fonction de  $n$  et  $m$ , dans les algorithmes suivants.

---

#### Algorithme 1.1 A( $n$ : entier) : entier

---

```
Debut
  i ← 1
  tant que (i ≤ n) faire
    i ← i + 2
  fin tant que
Fin
```

---

#### Algorithme 1.2 B( $n$ : entier) : entier

---

```
Debut
  i ← 1
  j ← n
  tant que (i ≤ j) faire
    i ← i + 1
    j ← j - 1
  fin tant que
Fin
```

---

#### Algorithme 1.3 C( $n$ : entier) : entier

---

```
Debut
  i ← 1
  tant que (i ≤ n) faire
    j ← 1
    tant que (j ≤ n) faire
      j ← j + 1
    fin tant que
    i ← i + 1
  fin tant que
Fin
```

---

#### Algorithme 1.4 D( $n$ : entier) : entier

---

```
Debut
  i ← 1
  tant que (i ≤ n) faire
    j ← i + 1
    tant que (j ≤ n) faire
      j ← j + 1
    fin tant que
  fin tant que
```



---

```

     $i \leftarrow i + 1$ 
  fin tant que
Fin

```

---



---

**Algorithme 1.5**  $E(m, n : \text{entiers}) : \text{entier}$ 


---

```

Debut
   $i \leftarrow 1$ 
  tant que ( $i \leq m$  et  $i \leq n$ ) faire
     $i \leftarrow i + 1$ 
  fin tant que

Fin

```

---



---

**Algorithme 1.6**  $F(m, n : \text{entiers}) : \text{entier}$ 


---

```

Debut
   $i \leftarrow 1$ 
  tant que ( $i \leq m$  ou  $i \leq n$ ) faire
     $i \leftarrow i + 1$ 
  fin tant que

Fin

```

---



---

**Algorithme 1.7**  $G(m, n : \text{entiers}) : \text{entier}$ 


---

```

Debut
   $i \leftarrow 1$ 
   $j \leftarrow 1$ 
  tant que ( $j \leq n$ ) faire
    si ( $i \leq m$ )
       $i \leftarrow i + 1$ 

    sinon
       $j \leftarrow j + 1$ 
    fin si

  fin tant que

Fin

```

---



---

**Algorithme 1.8**  $H(m, n : \text{entiers}) : \text{entier}$ 


---

```

Debut
   $i \leftarrow 1$ 
   $j \leftarrow 1$ 
  tant que ( $j \leq n$ ) faire
    si ( $i \leq m$ )
       $i \leftarrow i + 1$ 

    sinon
       $j \leftarrow j + 1$ 
       $i \leftarrow 1$ 

    fin si
  a)

  fin tant que

```

---

**Fin**

2. Quels sont les algorithmes ayant un coût linéaire, et quels sont ceux ayant un coût quadratique ?

**Correction exercice 2**

1. Le nombre d'affectations de l'algorithme **A** est de 1 affectation par itération. L'algorithme fera  $\frac{n}{2}$  itérations. Il y a 1 affectation pour l'initialisation. Au total, il y aura donc  $\frac{n}{2} + 1$  affectations.
2. Le nombre d'affectations de l'algorithme **B** est de 2 affectations par itération. L'algorithme fera  $\frac{n}{2}$  itérations. Il y a 2 affectations pour l'initialisation. Au total, il y aura donc  $2 \times \frac{n}{2} + 2 = n + 2$  affectations.
3. Le nombre d'affectations de l'algorithme **C** est de :
  - 1 affectation pour l'initialisation.
  - 1 affectation par itération de la boucle sur  $j$ . La boucle sur  $j$  est répétée  $n$  fois. Au total cela fait donc  $n$  affectations.
  - 2 affectations + les  $n$  affectations de la boucle sur  $j$  par itération de la boucle sur  $i$ . La boucle sur  $i$  est répétée  $n$  fois. Au total pour la boucle sur  $i$ , cela fait  $n \times (n + 2)$ .Le total fait donc  $n \times (n + 2) + 1$  affectations.
4. Le nombre d'affectations de l'algorithme **D** est de :
  - 1 affectation pour l'initialisation.
  - 2 affectations + les affectations de la boucle sur  $j$  par itération de la boucle sur  $i$ . La boucle sur  $i$  est répétée  $n$  fois.
  - La boucle  $j$  fera 1 affectation par itération. Cette boucle sera exécutée  $n - i$  fois. Ce nombre dépend de la valeur dans la boucle  $i$ . Elle sera donc exécutée  $n - 2$  fois au premier passage puis  $n - 3$  fois, ... 1 fois. Au total cela fera donc  $(n - 2) + (n - 3) + \dots + 1 = \frac{(n-2)(n-1)}{2}$  affectations.On arrive à un total de  $1 + 2n + \frac{(n-2)(n-1)}{2}$  affectations.
5. Le nombre d'affectations de l'algorithme **E** est de :
  - 1 affectation pour l'initialisation.
  - 1 affectation par itération de la boucle. Cette boucle sera exécutée  $\min(m, n)$  fois.On arrive à un total de  $1 + \min(m, n)$  affectations.
6. Le nombre d'affectations de l'algorithme **F** est de :
  - 1 affectation pour l'initialisation.
  - 1 affectation par itération de la boucle. Cette boucle sera exécutée  $\max(m, n)$  fois.On arrive à un total de  $1 + \max(m, n)$  affectations.
7. Le nombre d'affectations de l'algorithme **G** est de :
  - 2 affectations pour l'initialisation.
  - 1 affectation par itération de la boucle. Cette boucle sera exécutée  $(m + n)$  fois.On arrive à un total de  $2 + (m + n)$  affectations.
8. Le nombre d'affectations de l'algorithme **H** est de :
  - 2 affectations pour l'initialisation.
  - 1 affectation par itération de la boucle. Cette boucle sera exécutée  $(m \times n)$  fois.On arrive à un total de  $2 + (m \times n)$  affectations.
9. Les algorithmes **A**, **B**, **E**, **F**, **G** sont des algorithmes linéaires. Les algorithmes **C**, **D**, **H** sont des algorithmes quadratiques.

**Exercice 3 –  $\mathcal{P}$  Algorithme mystère**

1. Que calcule l'algorithme suivant ? Donner une preuve de votre réponse.

**Algorithme 1.9  $f(n : \text{entier}) : \text{entier}$** 


---

```

Debut
   $r \leftarrow 0$ 
  pour  $i$  de 1 à  $n$  faire
     $r \leftarrow 2r + 1$ 
  fin pour
  retourner  $r$ 

```

**Fin**

2. Que calcule l'algorithme suivant et quelle est sa complexité ? Environ quel temps faut-il pour calculer  $g(100)$  en supposant que l'on fait  $10^{10}$  additions par secondes.

**Algorithme 1.10  $g(n : \text{entier}) : \text{entier}$** 


---

```

Debut
  si ( $n = 0$ )
    retourner 0

  sinon
    retourner ( $g(n-1) + g(n-1) + 1$ )
  fin si

```

**Fin****Correction exercice 3**

1. L'algorithme calcule  $2^n - 1$ . Si l'on calcule les premières itérations on obtient les valeurs suivantes pour  $r$  : 1, 3, 7, 15, ....
2. Cet algorithme calcule aussi  $2^n - 1$ .

**Exercice 4 –  $\mathcal{P}$  Minimum d'un tableau**

Pour un tableau  $t$ , on note  $t.n$  sa taille et  $t[i]$  l'élément d'indice  $i$  qui doit être compris entre 0 et  $t.n - 1$ .

1. Écrire un algorithme qui retourne la valeur minimale d'un tableau.
2. Combien de comparaisons effectue cet algorithme (on ne compte que les comparaisons avec des valeurs du tableau) en fonction de  $t.n$  ?
3. Comment modifier l'algorithme précédent afin de calculer la valeur maximale du tableau ? On suppose qu'on dispose de l'opération  $-t$  qui change le signe des valeurs du tableau.
4. Écrire alors un algorithme qui retourne les valeurs à la fois du minimum et du maximum dans ce tableau. Combien de comparaisons fait-il ?
5. Trouver un algorithme qui n'effectue que  $\frac{3n}{2}$  comparaisons.
6. Écrire un algorithme qui retourne les 2 plus grandes valeurs du tableau, et donner son coût.
7. Sur un tableau de 8 entiers, trouver une méthode qui permet de calculer les 2 plus grandes valeurs en ne faisant que 9 comparaisons.
8. Écrire l'algorithme correspondant dans le cas général.

**Correction exercice 4**

1. Voici l'algorithme de recherche d'un élément minimum dans un tableau.

---

*Minimum*(*t* : tableau) : entier

---

**Debut**  
*i* ← 0  
 ind\_min ← 0  
 tant que (*i* < *t.n*) faire  
   si (*t*[*i*] < *t*[ind\_min])  
     ind\_min ← *i*  
  
 fin si  
*i* ← *i* + 1  
  
 fin tant que  
 retourner *t*[ind\_min]

**Fin**

2. A chaque itération il y a une comparaison entre les éléments du tableau. L'algorithme fait  $t.n$  itérations. Au total il y a donc  $t.n$  comparaisons.
3. Pour modifier l'algorithme, plusieurs façon de le faire :
- Soit on change juste le < par un >.
  - Soit on utilise l'algorithme Minimum sur le tableau  $-t$  au lieu du tableau  $t$ .
4. Un algorithme qui recherche à la fois le minimum et le maximum du tableau pourrait ressembler à :

---

*Min\_Max*(*t* : tableau) : 2 entiers

---

**Debut**  
 min ← Minimum(*t*)  
 max ← Maximum(-*t*)  
 retourner min, max

**Fin**

5. L'idée d'un algorithme qui effectue  $\frac{3n}{2}$  comparaisons pour trouver à la fois le minimum et le maximum est de regrouper les éléments par paire. Pour une paire donnée, on va donc comparer la plus petite des 2 valeurs à la valeur minimale connue jusqu'à présent et on fera pareil entre la plus grande des 2 valeurs et la valeur maximale connue. Ce qui revient à dire que pour chaque paire, on fait une comparaison entre les éléments de la paire pour connaître la plus grande et la plus petite des 2 valeurs, ensuite on fait les comparaisons avec le minimum et le maximum. Ce qui au total fait 3 comparaisons par paire d'éléments et il y a au total  $\frac{n}{2}$  paires d'éléments d'où le résultat.
6. Voici un algorithme naïf pour répondre à la question :

---

*2\_plus\_grandes\_valeurs*(*t* : tableau) : 2 entiers

---

**Debut**  
*i* ← 0  
 ind\_max1 ← 0  
 tant que (*i* < *t.n*) faire  
   si (*t*[*i*] < *t*[ind\_max1])  
     ind\_max1 ← *i*

```

    fin si
    i ← i + 1

    fin tant que
    si ind_max1 = 0 alors
        ind_max2 = 1
    sinon
        ind_max2 = 0
    fin si
    i ← 0
    tant que (i < t.n et i ≠ ind_max1) faire
        si (t[i] < t[ind_max2])
            ind_max2 ← i

    fin si
    i ← i + 1

    fin tant que
    retourner t[ind_max1] et t[ind_max2]

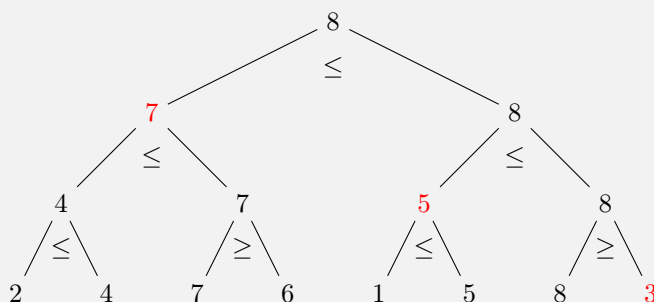
```

**Fin**

Cet algorithme coûte en nombre de comparaisons (des éléments entre eux) :

- première boucle :  $n$  comparaisons
  - deuxième boucle  $n - 1$  comparaisons
- Au total, cela donne  $2n - 1$  comparaisons.

7. Il s'agit ici de modéliser la recherche de la plus grande valeur comme lors d'un tournoi de tennis par exemple. Voici un exemple sur 8 entiers, pour trouver l'élément maximum. Pour trouver le deuxième plus grand il est nécessairement parmi les éléments marqués en rouge sur la figure ci-dessous, c'est-à-dire tous les éléments qui ont été comparés à la plus grande valeur. Dans notre cas, il va donc falloir en plus des 7 comparaisons représentées sur la figure, comparer 3 et 5 puis 5 et 7. Ce qui fera bien 9 comparaisons en tout.



### Exercice 5 – D Sous-sommes d'un tableau

On considère un tableau  $t$  de  $n$  entiers signés (positifs et négatifs). On veut trouver un sous-tableau de  $t$  (c'est-à-dire un ensemble d'éléments consécutifs de  $t$ ) ayant la somme maximale.

1. Quel est le sous-tableau ayant la somme maximale dans le tableau suivant :

1, 4, -5, 5, -1, 1, 4, -1

2. Écrire un algorithme qui prend en entrée deux indices  $g$  et  $d$  avec  $0 \leq g \leq d \leq n - 1$  et qui retourne la somme des éléments du sous tableau compris entre les indices  $g$  et  $d$ .
3. Écrire un algorithme qui retourne la somme maximale parmi tous les couples  $(g, d)$  possibles. Vous pouvez utiliser l'algorithme précédent.

**Correction exercice 5**

1. Il y a deux sous-tableau de somme maximale :  $1, 4, -5, 5 - 1, 14$  et  $5, -1, 1, 4$  dont la somme vaut 9.
2. Un algorithme qui calcule la somme des éléments d'un tableau entre les indices  $g$  et  $d$  est :

---

*sous\_somme*( $t$  : tableau,  $g$  : entier,  $d$  : entier) : entier

---

**Debut**

$i \leftarrow g$

**somme**  $\leftarrow 0$

**tant que** ( $i \leq d$ ) **faire**

**somme**  $\leftarrow$  **somme** +  $t[i]$

$i \leftarrow i + 1$

**fin tant que**

**retourner** **somme**

**Fin**

3. Un algorithme qui retourne la somme maximale parmi tous les couples possibles est le suivant :

---

*somme\_max*( $t$  : tableau) : entier

---

**Debut**

**max**  $\leftarrow 0$

**pour**  $l$  **de** 1 **à**  $n$  **faire**

**pour**  $g$  **de** 1 **à**  $n - l + 1$  **faire**

**somme**  $\leftarrow$  *sous\_somme*( $t, g, g + l - 1$ )

**si** (**max** < **somme**)

**max**  $\leftarrow$  **somme**

**fin si**

**fin pour**

**fin pour** **retourner** **max**

**Fin**

**Exercice 6 –  $\mathcal{D}$  Polynôme**

Un polynôme s'écrit  $P(X) = a_0 + a_1X + a_2X^2 + \dots + a_nX^n$ . On stocke le polynôme dans un tableau qui contient ses coefficients  $(a_0, \dots, a_n)$  à l'indice correspondant.

1. Écrire un algorithme qui retourne la somme de deux polynômes.
2. Écrire un algorithme qui retourne la dérivée d'un polynôme.
3. Écrire un algorithme qui retourne le produit de deux polynômes.

**Correction exercice 6**

On supposera que dans la suite tous les tableaux représentant des polynômes sont de taille  $n + 1$ .

1.

---

*somme\_polynome*(*p* : tableau, *q* : tableau ) : tableau
 

---

**Debut**▷ *Variables Locales**i* : entier*R* : tableaupour *i* de 0 à *n* faire    *R*[*i*] ← *P*[*i*] + *Q*[*i*]

fin pour

retourner *R***Fin**

2.

---

*derive\_polynome*(*p* : tableau ) : tableau
 

---

**Debut**▷ *Variables Locales**i* : entier*R* : tableaupour *i* de 1 à *n* faire    *R*[*i*-1] ← *P*[*i*] \* *i*

fin pour

*R*[*n*] ← 0retourner *R***Fin**

3. Soit  $p = \sum_{k=0}^n a_k X^k$  et  $q = \sum_{k=0}^n b_k X^k$  alors leur produit vaut  $r = \sum_{k=0}^{n+n} (\sum_{i=0}^k a_i b_{k-i}) x^k$ .  
L'algorithme se déduit directement de la formule

---

*produit\_polynome*(*p* : tableau, *q* : tableau ) : tableau
 

---

**Debut**▷ *Variables Locales**i* : entier*R* : tableaupour *k* de 0 à 2*n* faire    *r*[*k*] ← 0    pour *i* de 0 à *k* faire        *r*[*k*] ← *r*[*k*] + *p*[*i*]\**q*[*k*-*i*]

fin pour

fin pour

retourner *R***Fin****Exercice 7** – D Taille d'un entier

1. Écrire un algorithme itératif et un algorithme récursif qui prend en entrée un entier *n* et renvoie sa taille en base 10. La taille d'un entier est le nombre de chiffres qu'il faut pour l'écrire dans sa représentation en base 10.
2. Quelle est la complexité de l'algorithme d'addition de deux nombres entiers que l'on apprend à l'école ?

**Correction exercice 7**

1. Si l'on prend l'entier 4563, l'algorithme doit nous renvoyer 4. Il faut faire ici des divisions successives par 10 jusqu'à obtenir un reste plus petit que 10 et compter le nombre de divisions que l'on fait. La taille sera donc le nombre de divisions effectuées + 1 (qui correspond au reste).

Voici un algorithme itératif :

---

*taille\_entier*(*n* : entier) : entier

---

**Debut**

▷ *Variables Locales*

*r, taille* : entier

*taille* ← 0

*r* ← *n*

**tant que** (*r* > 10) **faire**

*r* = *r* / 10;

*taille* ← *taille* + 1

**fin tant que**

retourner *taille* + 1

**Fin**

Voici un algorithme récursif :

---

*taille\_entier*(*n* : entier) : entier

---

**Debut**

▷ *Variables Locales*

*r, taille* : entier

**si** (*n* < 10)

    retourner 1

**sinon**

    retourner *taille\_entier*(*n*/10) + 1

**fin si**

**Fin**

2. L'addition de deux entiers est faite en additionnant, les chiffres des unités entre eux et ensuite des dizaines, puis des centaines, etc. Le nombre d'opérations élémentaires dépend donc de la taille des 2 entiers et plus précisément de la taille du plus grand des 2 entiers.

**Exercice 8 – D Tableau unimodal**

Un tableau *t* de *n* entiers est unimodal s'il est d'abord croissant puis décroissant. Dit autrement, il existe un indice *m*, avec  $0 \leq m \leq n - 1$  tel que

—  $t[i] \leq t[i + 1]$  pour  $0 \leq i \leq m - 1$  et

—  $t[i] \geq t[i + 1]$  pour  $m \leq i < n - 1$

En particulier,  $t[m]$  est l'élément maximal. Notez qu'un tableau trié est un tableau unimodal ( $m = n - 1$  si il est trié en ordre croissant).

Dans cet exercice, on suppose que toutes les valeurs du tableau sont distinctes.

1. Écrire un algorithme qui teste si un tableau est unimodal : votre algorithme doit retourner **vrai** s'il l'est, et **faux** sinon.
2. Écrire un algorithme qui calcule l'élément maximal du tableau unimodal *t* de complexité  $O(\log n)$ .



3. Pour la classe des tableaux triés, des tableaux unimodaux et des tableaux quelconques, comparer la complexité des algorithmes de la recherche du maximum et de la recherche d'une valeur.

### Correction exercice 8

1. Un exemple de tableau unimodal est le suivant :

1	3	5	7	10	6	4	2
---	---	---	---	----	---	---	---

Les éléments compris entre la position 0 dans le tableau et la position 4 sont croissants puis de la position 5 à 7 les éléments sont en ordre décroissant.

Dans l'algorithme suivant,  $t$  est le tableau,  $n$  est sa taille. La variable croissant vaut 1 quand le tableau est croissant et 0 sinon.

---

**Verif\_unimodal(  $t$  : tableau,  $n$  : entier ) : booléen**

---

**Debut**

▷ *Variables Locales*

*croissant* : entier

**croissant**  $\leftarrow$  1

**pour**  $i$  de 0 à  $n-2$  **faire**

**si** **croissant** = 1 et  $t[i] \leq t[i+1]$

**croissant** = 0

**si** **croissant** = 0 et  $t[i] \geq t[i+1]$

**retourner faux**

**fin pour**

**retourner vrai**

**Fin**

2. Pour atteindre une complexité en  $O(\log n)$  il faut couper le tableau pour n'explorer qu'une partie du tableau, comme on le fait avec une recherche dichotomique. Ici le principe va être le suivant :
- pour un tableau  $a$  un seul élément, l'élément maximal est cet élément.
  - Dans un tableau *unimodal* dont les éléments sont compris entre les indices  $g$  et  $d$ . On va calculer l'indice  $m$ , au milieu de ce sous-tableau. Si  $t[m] < t[m+1]$ , alors on va chercher l'élément maximal entre les indices  $m+1$  et  $d$ , sinon on va chercher entre les indices  $g$  et  $m$ . L'algorithme s'écrit donc :

---

**Max\_unimodal(  $t$  : tableau,  $g$  : entier,  $d$  : entier ) : booléen**

---

**Debut**

**si** ( $g > d$ )

$m \leftarrow (g+d) / 2$

**si** ( $t[m] < t[m+1]$ ) **retourner** Max\_unimodal( $t, m+1, d$ )

**sinon** **retourner** Max\_unimodal( $t, g, m$ )

**fin si**

**sinon** **retourner**  $t[g]$

**fin si**

**Fin**

3. Une synthèse pour la recherche du maximum et la recherche d'un élément dans différents tableaux de taille  $n$ .

## 1. EXERCICES

	Recherche d'un élément	Recherche du maximum
Tableau quelconque	$O(n)$	$O(n)$
Tableau unimodal	$O(n)$	$O(\log n)$
Tableau Trié	$O(\log n)$	$O(1)$

### Exercice 9 – D Doublons

1. Dans un tableau de  $n$  entiers, proposer un algorithme qui retourne un tableau dans lequel les doublons ont été supprimés.
2. Même question si le tableau est trié.
3. Pour la première question, est-il intéressant de le trier et d'appliquer ensuite l'algorithme sur les tableaux triés (la complexité du tri est  $O(n \log n)$ ) ?

#### Correction exercice 9

1. Supposons que nous ayons un tableau  $T$  en entrée et que nous voulions créer un tableau  $S$  en sortie qui contient les éléments de  $T$  sans les doublons. L'algorithme consiste à parcourir le tableau  $T$  et pour chaque élément on va vérifier s'il est déjà dans le tableau  $S$ , si oui on passe à l'élément suivant sinon on insère l'élément dans le tableau  $S$  et on passe à l'élément suivant dans le tableau  $T$ . Ce qui conduit à une complexité de  $O(n^2)$ .
2. Si le tableau est trié, c'est plus facile car les doublons se suivent dans le tableau, il suffit de parcourir le tableau  $T$ , et de rajouter les éléments de  $T$  dans  $S$  que si le dernier élément inséré dans  $S$  est différent de l'élément de  $T$  à insérer. On a une complexité en  $O(n)$ .
3. Il est donc plus efficace de trier le tableau avec un tri en  $O(n \log n)$  puis d'appliquer l'algorithme de la question 2 que d'appliquer l'algorithme de la question 1.

## 1.2 Les tris

Pour cette partie, nous allons faire un TD commun avec le module IN301. Le sujet du TD se trouve dans le polycop de TD d'IN301 dans l'espace moodle LSIN301.

## 1.3 Les piles et les files

On rappelle les primitives de manipulation des **piles** :

- `creer_pile()` : `Pile` : créé et retourne une pile vide ;
- `pile_vide(p: Pile)` : `booléen` : teste si une pile est vide ;
- `empiler(p: Pile, x: entier)` : insère l'entier  $x$  dans la pile ;
- `depiler(p: Pile)` : `entier` : retire l'entier sur le dessus de la pile et retourne sa valeur ;

et des **files** :

- `creer_file()` : `File` : créé et retourne une file vide ;
- `file_vide(f: File)` : `booléen` : teste si une file est vide ;
- `insere_file(f: File, x: entier)` : insère l'entier  $x$  dans la file ;
- `supprime_file(f: File)` : `entier` : retire un entier de la file et retourne sa valeur ;

### Exercice 10 – P Manipulations diverses

Pour traiter les questions suivantes, vous pouvez utiliser des variables locales de type `Pile` ou `File`.

1. Écrire un algorithme qui inverse l'ordre des éléments d'une pile.
2. Écrire un algorithme qui supprime un élément sur deux d'une pile. L'ordre des éléments doit être conservé.

3. Écrire un algorithme qui, étant donnée une pile, retourne deux piles qui contiennent respectivement tous les entiers pairs et tous les entiers impairs.
4. Écrire un algorithme qui étant donnée une pile  $P_1$  stocke dans une pile  $P_2$  les nombres pairs contenus dans  $P_1$ . Le contenu de  $P_1$  est inchangé après l'exécution de l'algorithme. Les nombres pairs stockés dans  $P_2$  doivent être dans le même ordre que dans  $P_1$ .

### Correction exercice 10

1. Un algorithme qui inverse l'ordre des éléments d'une pile.

---

**Inverse( P : Pile ) :Pile**

---

**Debut**

▷ *Variables Locales*

**x : entier ; P2 : Pile**

**P2 ← creer\_pile()**

**tant que pile\_vide(P) ≠ vrai faire**

**x ← depiler(P)**

**empiler(P2,x)**

**fin tant que**

**retourner P2**

**Fin**

2. Un algorithme qui supprime un élément sur deux de la pile.

---

**un\_sur\_deux( P : Pile ) :Pile**

---

**Debut**

▷ *Variables Locales*

**x : entier ; inter : Pile**

**inter ← creer\_pile()**

**tant que pile\_vide(P) ≠ vrai faire**

**x ← depiler(P)**

**empiler(inter,x)**

**depiler(P)**

**fin tant que**

**p ← Inverse(inter)**

**retourner P**

**Fin**

3. Un algorithme qui renvoie 2 piles : l'une avec les entiers pairs de la pile d'entrée, l'autre avec les entiers impairs.

---

**pair\_impair( P : Pile ) :Pile, Pile**

---

**Debut**

▷ *Variables Locales*

**x : entier ; pair, impair : Pile**

**pair ← creer\_pile()**

**impair ← creer\_pile()**

**tant que pile\_vide(P) ≠ vrai faire**

**x ← depiler(P)**

**si x modulo 2 = 0 alors**

**empiler(pair,x)**

**sinon**

```

        empiler(imppair,x)
    fin si
    fin tant que
    retourner pair, impair

```

**Fin**

4. un algorithme qui renvoie les entiers pairs contenus dans la pile d'entrée. La pile d'entrée est inchangée.

---

**pair( P1 : Pile ) :Pile**

---

**Debut**

▷ *Variables Locales*

```

    x : entier ; pair, inter,P2 : Pile
    pair ← creer_pile()
    inter ← creer_pile()
    P2 ← creer_pile()
    tant que pile_vide(P1) ≠ vrai faire
        x ← depiler(P1)
        si x modulo 2 = 0 alors
            empiler(pair,x)
        fin si
    fin tant que
    P2 ← Inverse(pair)
    P1 ← Inverse(inter)
    retourner P2

```

**Fin**

### Exercice 11 – P Implémentation d'une file par deux piles

Proposer une façon d'implémenter une file avec deux piles, puis écrire les primitives de manipulation des files à partir des primitives des piles. Donner la complexité de vos algorithmes.

#### Correction exercice 11

On commence par se définir une structure de données :

```

Enregistrement File {
    Pile e
    Pile s
}

```

Ensuite nous allons réécrire les primitives d'accès à une file à l'aide de cette nouvelle structure de données.

1. Création d'une file

---

**Creer\_File( ) :File**

---

**Debut**

▷ *Variable Locale*

```

    F : File
    F.e ← creer_pile()
    F.s ← creer_pile()

```

**Fin**

La complexité de cet algorithme est en  $O(1)$ .

2. La file est-elle vide ?

---

**File\_Vide( F : File ) : booléen**

---

**Debut**  
     **si** pile\_vide(F.e) = vrai et pile\_vide(F.s) = vrai **alors**  
         **retourner** vrai  
     **sinon**  
         **retourner** faux  
     **fin si**  
**Fin**

La complexité de cet algorithme est en  $O(1)$

3. Insérer un élément dans la file

---

**Inserer\_File(F : File, x : entier)**

---

**Debut**  
     **empiler** (F.e,x)  
**Fin**

La complexité de cet algorithme est en  $O(1)$

4. Supprimer un élément de la file

---

**Supprimer\_File ( F : File ) : entier**

---

**Debut**  
     F.s  $\leftarrow$  Inverse(F.e)  
     x  $\leftarrow$  depiler(F.s)  
     F.e  $\leftarrow$  Inverse(F.s)  
     **retourner** x  
**Fin**

La complexité de cet algorithme est en  $O(n)$ .

### Exercice 12 – $\mathcal{P}$ Implémentation d'une pile par deux files

Proposer une façon d'implémenter une pile avec deux files, puis écrire les primitives de manipulation des piles à partir des primitives des files. Donner la complexité de vos algorithmes.

#### Correction exercice 12

On commence par se définir une structure de données :

```
Enregistrement Pile {
  File e
  File s
}
```

Ensuite nous allons réécrire les primitives d'accès à une pile à l'aide de cette nouvelle structure de données.

1. Création d'une pile

---

**Creer\_Pile( ) :Pile**

---

**Debut**  
     ▷ *Variable Locale*  
     **P : Pile**  
     P.e  $\leftarrow$  creer\_file()

```
P.s ← creer_file()
```

**Fin**

La complexité de cet algorithme est en  $O(1)$ .

2. La pile est-elle vide ?

---

**Pile\_Vide( P : Pile ) : booleen**

---

**Debut**

```
si file_vide(P.e) = vrai et file_vide(P.s) = vrai alors
```

```
retourner vrai
```

```
sinon
```

```
retourner faux
```

```
fin si
```

**Fin**

La complexité de cet algorithme est en  $O(1)$

3. Insérer un élément dans la file

---

**Inserer\_Pile(P : Pile, x : entier)**

---

**Debut**

```
insere_file (P.e,x)
```

**Fin**

La complexité de cet algorithme est en  $O(1)$

4. Supprimer un élément de la file

---

**Supprimer\_Pile ( P : Pile ) : entier**

---

▷ *Variables Locales*

**x, elt : entier**

**Debut**

```
tant que file_vide(P.e) ≠ vrai faire
```

```
x ← supprimer_file(P.e)
```

```
si file_vide(P.e) = faux alors insere_file(f.s,x)
```

```
fin tant que
```

```
elt ← x
```

```
tant que file_vide(P.s) ≠ vrai faire
```

```
x ← supprimer_file(P.s)
```

```
insere_file(P.e,x)
```

```
fin tant que
```

```
retourner elt
```

**Fin**

La complexité de cet algorithme est en  $O(n)$ .

## 1.4 Les arbres

### Exercice 13 – Parcours d'arbres

La liste des valeurs suivantes :

15, 5, 9, 12, 2, 4, 11, 3, 7, 8, 1, 6, 10, 13, 14

a été mémorisée sous forme d'arbre binaire. Il s'agit de la liste des valeurs obtenues par un parcours en largeur dans un arbre binaire.

1. Dessiner l'arbre obtenu.

2. Quel est l'ordre des valeurs que l'on obtient quand on fait un parcours infixe ?
3. Et avec des parcours préfixe et postfixe ?
4. Quels sont tous les arbres qui ont pour parcours préfixe :1, 2, 3 ? Pour chacun des ces arbres, donnez le parcours postfixe et infixe correspondant.
5. Existe-t-il un arbre à 8 nœuds dont le parcours préfixe est 1, 2, 3, 4, 5, 6, 7, 8 et le parcours postfixe est 4, 3, 5, 2, 7, 8, 6, 1 ? Si oui le dessiner, si non le justifier.
6. Même question mais avec le parcours postfixe 5, 3, 2, 4, 7, 6, 8, 1.
7. Écrire un algo qui prend en entrée deux tableaux `t_pre` et `t_post` de même taille  $n$  et qui contiennent chacun toutes les valeurs de 1 à  $n$  (comme dans les 2 questions précédentes). Cet algorithme retourne `vrai` si ces deux tableaux peuvent être les parcours préfixe (pour `t_pre`) et postfixe (pour `t_post`) d'un même arbre binaire, et `faux` sinon.

#### Exercice 14 – Arbres Binaires

On considère la structure de données `arbre binaire`. Ecrire les fonctions suivantes :

- `Hauteur` qui calcule la hauteur d'un arbre.
- `NombreNoeuds` qui calcule le nombre de nœuds d'un arbre binaire
- `Descendants` qui calcule et stocke en chaque nœud de l'arbre, le nombre de ses descendants.
- `Un_Fils` qui calcule le nombre de sommets ayant exactement un fils
- `Complet` qui renvoie `vrai` si l'arbre est un arbre complet et `faux` sinon.

#### Exercice 15 – Bee Trees

Une abeille mâle est produite de manière asexuée à partir d'une abeille femelle. Par contre, une abeille femelle a deux parents : un mâle et une femelle.

1. Représenter l'arbre généalogique d'une abeille mâle jusqu'à la quatrième génération.
2. Combien une abeille mâle a-t-elle d'ancêtres de niveau 1 ? de niveau 2 ? de niveau  $n$  ?

#### Exercice 16 – Nombre de Strahler

On définit récursivement une fonction entière  $\phi$  sur l'ensemble des arbres binaires. Appelons  $sag(A)$  et  $sad(A)$  le sous-arbre gauche et le sous-arbre droit d'un arbre binaire  $A$  :

$$\phi(A) = \begin{cases} 0 & \text{si } A \text{ est l'arbre vide} \\ \max(\phi(sag(A)), \phi(sad(A))) & \text{si } \phi(sag(A)) \neq \phi(sad(A)) \\ \phi(sad(A)) + 1 & \text{si } \phi(sag(A)) = \phi(sad(A)) \end{cases}$$

Cette fonction  $\phi$  permet d'associer à tout arbre binaire un nombre dit de Strahler.

1. Quel est le nombre de Strahler d'un arbre binaire complet de hauteur  $n$  ? le nombre de Strahler d'un arbre binaire dégénéré de hauteur  $n$  ?
2. Donner une procédure récursive qui calcule le nombre de Strahler d'un arbre binaire.





---

## Chapitre 2

## Conseils

---

Vous trouverez dans ce document quelques conseils pour écrire correctement des algorithmes en pseudo-code, quelques rappels de mathématiques élémentaires (utiles pour les calculs de complexité) et enfin quelques techniques usuelles de preuves. J'espère que vous ferez bon usage de ce mémento !

### 2.1 Pseudo-Code

Le pseudo-code est un mélange de langage naturel et de concepts de programmation de haut-niveau qui décrit les idées générales d'un algorithme. Le but du pseudo-code est de s'abstraire des contingences d'un langage de programmation particulier. Chacun écrit le pseudo-code un peu à sa manière, ce n'est en rien un langage avec des règles claires (une grammaire) que l'on peut appliquer de manière automatique. Néanmoins, il reste que la plupart des gens s'accorde sur un certain nombre de règles de présentation. Ce sont celles-ci sur lesquelles je veux insister ici.

#### 2.1.1 Quelques règles générales

**Types élémentaires** Généralement, les types élémentaires de données sont les suivants : **Entier**, **Réel**, **Caractère**.

**Variables** Les variables sont typées et portent un nom facilement identifiable.

**Affectation** Il y a plusieurs façons de la noter soit  $\leftarrow$  soit  $=$  (à la manière du langage C) soit  $:=$  (à la manière de Pascal).

**Égalité** Là encore deux façons de l'écrire soit  $=$ , soit  $==$  (pour la différencier de l'affectation comme en C). On notera la différence (non égalité)  $<>$  ou bien  $\neq$ .

**Instructions** Ce que j'appellerai Instructions de manière générale regroupe 2 réalités : Soit une instruction simple terminée par un « ; » soit un bloc d'instructions (suite d'instructions généralement identifiable par l'indentation du programme).

**Conditionnelle** *Si condition alors Instructions si la condition est vraie sinon Instructions si la condition est fausse Fin Si*

**Boucles** Plusieurs types de boucles sont possibles :

- *Tant Que condition faire Instructions Fin Tant Que*
- *Répéter Instructions tant que condition Fin Répéter*
- *Pour variable de initialisation à valeur finale faire Instructions Fin Pour*

**Retour de valeur** *Retourner valeur*. Cette instruction stoppe l'exécution de la fonction en cours et retourne la valeur spécifiée.

**Tableaux** Un tableau  $A$  de type `tableau type1[n]` est une suite contiguë d  $n$  éléments (ou cases) notés  $A[1], A[2], \dots, A[n]$ . Chaque élément  $A[i]$  est une variable de type `type1`. Les tableaux à deux ou plusieurs dimensions sont définis de la même façon, c'est-à-dire `tableau type1[n1,n2, ..., n3]`. Par exemple pour un tableau bi-dimensionnel,  $A[i, j]$  donne accès au  $j$ ème élément de la  $i$ ème ligne du tableau  $A$  et est du même type que  $A$ .

**Enregistrements** Ils permettent de définir des types structurés. On les représentera généralement de la façon suivante : `enregistrement nom { champ1 : type1 ; champ2 : type 2 ; ... champn : typen }` L'accès au champ d'une variable structurée se fait par l'opérateur `.` (point).

**Pointeurs** Généralement, on ne descend pas au niveau des pointeurs pour les descriptions d'algorithmes, cela peut cependant se produire dans certains cas. Pour déclarer une variable de type pointeur on utilise le symbole  $\uparrow$  NIL représente un pointeur qui ne pointe sur rien.

**Entête d'une fonction** `Nom_de_la_fonction(liste de paramètres avec leurs types) : type renvoyé.` Il faut préciser ensuite quelles sont les variables modifiées et les variables non modifiées.

**Corps d'une fonction** Après l'entête de la fonction, on précise quelles sont les variables locales puis les Instructions de la fonction.

### 2.1.2 Quelques exemples

Voici quelques exemples d'algorithmes vus en cours et repris ici.

---

#### Algorithme 2.1 Somme des carrés des entiers entre m et n (version itérative)

---

**SommeCarrés**(`m` : entier, `n` : entier) : entier

▷ **Entrées** :  $m$  et  $n$

▷ **Sortie** : *somme des carrés des entiers entre m inclus et n inclus, si  $m \leq n$ , et 0 sinon.*

**Debut**

▷ **Variables locales**

`i` : entier ;

`som` : entier ;

`som`  $\leftarrow$  0 ;

**pour** `i` **de** `m` **à** `n` **faire**

`som`  $\leftarrow$  `som` + `i` \* `i` ;

**fin pour**

**retourner** `som`

**Fin**

---

**Algorithme 2.2 Somme des carrés des entiers entre m inclus et n inclus (version récursive)**


---

**SommeCarrés**(m : entier, n : entier) : entier

▷ Entrées : *m et n*

▷ Sortie : *somme des carrés des entiers entre m et n.*

▷ Pré-condition :  $m \leq n$

**Debut**

  si (m <> n)

    retourner (m\*m + SommeCarrés(m+1,n)) ;

  sinon

    retourner (m\*m) ;

  fin si

**Fin**

Il y a plusieurs choses à remarquer dans les algorithmes 1 et 2 :

- Tout d’abord, dans l’algorithme 2, on remarque que l’on a précisé les pré-conditions. Autrement dit, les conditions qui doivent être remplies avant l’utilisation de cet algorithme. En effet, si ces conditions ne sont pas remplies alors l’algorithme rend un résultat erroné.
- Dans l’algorithme 1, aucune pré-condition n’est précisée, toutefois la boucle **Pour**, ne s’exécutera que si la variable *i* n’a pas une valeur plus grande que *n*. Donc les conditions de validité de l’algorithme seront ici encore bien remplies.

---

**Algorithme 2.3 Recherche dans une liste chaînée**


---

**Recherche**(L : ↑ Element, x : entier) : booléen

▷ Entrées : *x (élément recherché), L (tête de liste)*

▷ Sortie : *vrai si l’élément x a été trouvé dans la liste L, faux sinon.*

▷ Pré-condition : *La liste L est triée par ordre croissant*

**Debut**

▷ *Variable Locale*

  p : ↑ Element

  p ← L ;

  tant que (p <> NIL) faire

    si (p.val < x)

      p ← p.suiv

    sinon si (p.val = x)

      retourner vrai

    sinon

      retourner faux

  fin si

  fin tant que

Dans ce dernier algorithme, il faut noter que j’ai fait appel à un type structuré **Element** défini de la manière suivante :

Enregistrement `Element { val : entier; suiv : ↑ Element }`. Attention ici, j'ai explicitement fait apparaître une boucle `tant que` avec une seule condition qui est que la liste ne soit pas vide. On aurait pu écrire une boucle qui intègre les 2 conditions et donc avoir une boucle du type `tant que p <> NIL et p.val < x`), toutefois cette écriture ne fonctionne que si le `et` est paresseux, à savoir si la première condition est fausse la seconde condition n'est pas évaluée. Ceci est vrai dans un certain nombre de langages de programmation mais pas dans tous. Si le `et` est passif alors la seconde condition est évaluée et dans ce cas, votre programme écrit à l'aide de l'algorithme ne pourra pas s'exécuter correctement. Il faut donc être très prudent lorsque l'on écrit de tels algorithmes. C'est pourquoi j'ai choisi de présenter ici un algorithme sans le `et` comme cela j'évite le piège du `et` actif ou passif. Cela n'a ici de fait, plus aucune importance.

---

### Algorithme 2.4 Recherche dichotomique

---

**Dicho**( $x$  : entier,  $S$  : tableau d'entiers,  $g$  : entier,  $d$  : entier) : booléen

▷ **Entrées** :  $x$  (élément recherché),  $S$  (espace de recherche),  $g$  (indice de gauche),  $d$  (indice de droite)

▷ **Sortie** : vrai si l'élément  $x$  a été trouvé dans le tableau  $S$  entre les indices  $g$  et  $d$ , faux sinon.

▷ **Pré-condition** :  $g$  et  $d$  sont des indices valides du tableau  $S$ .

**Debut**

▷ *Variable Locale*

$m$  : entier ;

si ( $g < d$ )

$m \leftarrow \lfloor (g+d)/2 \rfloor$  ;

si ( $x = S[m]$ )

retourner vrai ;

sinon si ( $x < S[m]$ )

retourner (Dicho( $x, S, g, m-1$ )) ;

sinon

retourner (Dicho( $x, S, m+1, d$ )) ;

fin si

sinon

retourner faux ;

fin si

**Fin**

## 2.2 Mathématiques élémentaires

Voici quelques rappels de mathématiques élémentaires.

### logarithmes

- $\log_b(xy) = \log_b x + \log_b y$
- $\log_b(x/y) = \log_b x - \log_b y$
- $\log_b x^\alpha = \alpha \log_b x$
- $\log_b x = \frac{\ln x}{\ln b} = \frac{\log_{10} x}{\log_{10} b}$

### exposants

- $a^{(b+c)} = a^b a^c$
- $a^{bc} = (a^b)^c$
- $\frac{a^b}{a^c} = a^{(b-c)}$
- $b = a^{\log_a b}$
- $\sqrt[n]{x} = x^{\frac{1}{n}}$

## Parties entières

- Partie entière inférieure :  $\lfloor x \rfloor$  = le plus grand entier inférieur ou égal à  $x$ .
- Partie entière supérieure :  $\lceil x \rceil$  = le plus petit entier supérieur ou égal à  $x$ .
- $x - 1 \leq \lfloor x \rfloor \leq x \leq \lceil x \rceil \leq x + 1$

Par exemple,  $\lfloor 3.8 \rfloor = 3$  et  $\lfloor -3.8 \rfloor = -4$ . De même  $\lceil 3.8 \rceil = 4$  et  $\lceil -3.8 \rceil = -3$ .

## Somme

$$\sum_{i=1}^n i = 1 + 2 + \dots + (n-1) + n = \frac{n(n+1)}{2}$$

## 2.3 Techniques de preuves

### 2.3.1 Les techniques efficaces

#### Exemple

- Pour montrer qu'une propriété est fausse, il suffit de trouver un contre-exemple qui ne vérifie pas la propriété. Par exemple si la question est *Montrer que l'algorithme de coloration séquentielle est optimal* ce qui revient à dire que *pour tout graphe, cet algorithme permet de trouver une coloration optimale*, il suffit d'en trouver un pur lequel ça ne marche pas pour montrer que l'affirmation est fausse. Le contre-exemple du cours (un graphe biparti) montre que dans une famille de cas, l'algorithme est non optimal.
- On peut montrer qu'une propriété d'existence est vraie à l'aide d'un exemple. Par exemple, si la question est *existe-t-il des graphes tels que leur indice chromatique est strictement supérieur au degré du graphe*. Vous pouvez répondre en exhibant un exemple de graphe : le cycle à 5 sommets qui est de degré 2 et qui nécessite 3 couleurs pour colorier ses arêtes.

#### Absurde

Il s'agit de montrer que la négation de l'énoncé aboutit à une absurdité. Le schéma de la démonstration est donc le suivant : on prend comme hypothèse la négation de l'énoncé (pour éviter tout risque d'erreur, on écrit cette hypothèse). En l'utilisant, on arrive à un résultat que l'on sait faux. On en conclut que l'hypothèse était fausse et donc que sa négation, l'énoncé est juste.

Par exemple dans la démonstration du théorème suivant

**Théorème 1** *Un graphe orienté  $G$  est acyclique si et seulement si un parcours en profondeur de  $G$  ne rencontre jamais un arc  $(u, v)$  tel que  $u$  est un descendant de  $v$  dans une arborescence en profondeur.*

#### Preuve.

1. On va d'abord prouver que si on a un DAG alors il n'existe pas d'arc  $(u, v)$  tel que  $u$  est un descendant de  $v$  dans une arborescence en profondeur. Pour cela, on va supposer le contraire : il existe un arc  $(u, v)$  tel que  $u$  est un descendant de  $v$  dans une arborescence en profondeur. Dans ce cas là, le sommet  $v$  est un ancêtre du sommet  $u$  dans la forêt en profondeur. Ce qui implique qu'il existe un chemin de  $v$  à  $u$  dans  $G$ , donc avec l'arc  $(u, v)$  on a un circuit ! ce qui est absurde, donc il n'existe pas d'arc  $(u, v)$  tel que  $u$  est un descendant de  $v$  dans une arborescence en profondeur.
2. On va maintenant partir du fait qu'il n'existe pas d'arc  $(u, v)$  tel que  $u$  est un descendant de  $v$  dans une arborescence en profondeur pour montrer que le graphe  $G$  est sans circuit. De la même façon que dans la première partie de la preuve, on va partir de la supposition contraire, c'est-à-dire qu'il existe un circuit  $C$  dans le graphe. Soit  $v$  le premier sommet découvert par un parcours en profondeur dans le circuit  $C$  et soit  $(u, v)$  l'arc précédent  $v$  dans  $C$ . A la date  $d[v]$ , il existe un chemin composé de sommets blancs (non encore découverts) de  $v$  à  $u$ . D'après le théorème du chemin blanc, le sommet  $u$  devient un descendant de  $v$  dans la forêt en profondeur. L'arc  $(u, v)$  est donc un arc tel que  $u$

est un descendant de  $v$  dans l'arborescence en profondeur. C'est une absurdité puisque c'est contraire à notre hypothèse de départ, donc il ne peut pas exister de circuit dans le graphe  $G$ .

□

### Induction (récurrence)

La preuve se fait en 2 temps :

1. Prouver le cas de base
2. En considérant que la propriété est vérifiée pour  $n$ , il faut montrer que la propriété est aussi vraie pour  $n + 1$ .

Par exemple, pour montrer la propriété des arbres binaires suivante :

**Lemme 1** *Un arbre binaire localement complet<sup>1</sup> ayant  $n$  nœuds internes a  $(n + 1)$  nœuds externes.*

**Preuve.** La preuve va se faire par récurrence sur le nombre de nœuds internes de l'arbre.

**Cas de base** Le plus petit arbre localement complet est l'arbre réduit à un seul nœud. Cet arbre a 0 nœud interne et 1 nœud externe. La propriété est donc vérifiée.

**Récurrence** Supposons cette propriété vraie pour tous les arbres ayant moins de  $n$  nœuds internes c'est-à-dire  $\forall k \leq n$ , tout arbre binaire localement complet ayant  $k$  nœuds internes a  $k + 1$  nœuds externes.

Soit  $B$  un arbre constitué d'une racine  $o$  et de deux sous-arbres l'un gauche  $B_1$  et l'autre droit  $B_2$ .  $B$  a  $n$  nœuds internes. Soit  $n_1$  le nombre de nœuds internes de l'arbre  $B_1$  et  $n_2$ , le nombre de nœuds internes de l'arbre  $B_2$ . On a donc  $n = n_1 + n_2 + 1$ .

Par hypothèse de récurrence,  $B_1$  (resp.  $B_2$ ) a  $n_1 + 1$  (resp.  $n_2 + 1$ ) nœuds externes. Or les nœuds externes de  $B$  sont les nœuds externes de  $B_1$  et de  $B_2$ , donc le nombre de nœuds externes de  $B$  est  $n_1 + 1 + n_2 + 1 = n + 1$ . Ce qui termine la preuve puisque nous avons montré que si la propriété était vraie pour tous les arbres ayant moins de  $n$  nœuds internes alors elle était vraie pour tous les arbres ayant  $n$  nœuds internes. De plus, la propriété est vraie pour les arbres les plus petits (avec un seul nœud). Donc la propriété est vraie pour tous les arbres.

□

### 2.3.2 Les techniques absolument inefficaces :-)

- Donner un exemple pour une propriété générale. Exemple : *montrer que tous les entiers impairs sont premiers*. Réponse : C'est vrai pour 3 donc c'est vrai pour tous!!!!
- Preuve par excès d'agitations des mains (cela peut éventuellement être utile pour un oral, mais généralement c'est très peu convaincant).
- Preuve par diagramme incompréhensible (très en vogue lors des examens écrits mais est tout aussi inefficace que le précédent).
- Preuve par intimidation : *"Cette preuve est tellement évidente que seul un idiot serait incapable de la comprendre."* La notation en général est tout aussi évidente :-)

---

1. Un arbre binaire est localement complet s'il est binaire non vide et chaque nœud a 0 ou 2 fils. On appelle nœud externe un nœud ayant 0 fils et nœud interne un nœud ayant 2 fils.

---

## Chapitre 3

# Résumés de cours

---

### 3.1 Introduction à la complexité des algorithmes

- *Mesure intrinsèque* de la complexité de l'algorithme indépendamment de l'implémentation.
- Permet la *comparaison* entre différents algorithmes pour un même problème.

#### Définition 1. Différentes Mesures

- Complexité en temps
- Complexité en espace

**But :** « Sur toute machine, et quel que soit le langage utilisé, l'algorithme  $\alpha$  est meilleur que l'algorithme  $\beta$  pour des *données de grande taille*. »

#### 3.1.1 Quelques règles

$cout(x)$  : nbre d'op. élémentaires de l'ens. d'instructions  $x$ .

- **Séquence d'instructions** :  $x_1; x_2;$

$$cout(x_1; x_2;) = cout(x_1;) + cout(x_2;)$$

- **Les boucles simples** : tant que condition faire  $x_i$ ;

$$cout(boucle) = \sum_{i=1}^n (cout(x_i) + cout(condition))$$

- **Conditionnelle** : Si condition alors  $x_{vrai}$ ; sinon  $x_{faux}$ ;

$$cout(conditionnelle) \leq cout(condition) + \max(cout(x_{vrai}); cout(x_{faux}))$$

#### 3.1.2 Grandeurs

- Caractérisation du comportement d'un algorithme  $\mathcal{A}$  sur l'ensemble des données  $D_n$  de taille  $n$ .
- $Cout_{\mathcal{A}}(d)$  : coût de l'algorithme  $\mathcal{A}$  sur la donnée  $d$ .

— **Complexité dans le meilleur cas :**

$$\text{Min}_{\mathcal{A}}(n) = \min\{\text{Cout}_{\mathcal{A}}(d), d \in D_n\}$$

— **Complexité dans le pire cas :**

$$\text{Max}_{\mathcal{A}}(n) = \max\{\text{Cout}_{\mathcal{A}}(d), d \in D_n\}$$

— **Complexité en moyenne :**

$$\text{Moy}_{\mathcal{A}}(n) = \sum_{d \in D_n} p(d) \times \text{Cout}_{\mathcal{A}}(d)$$

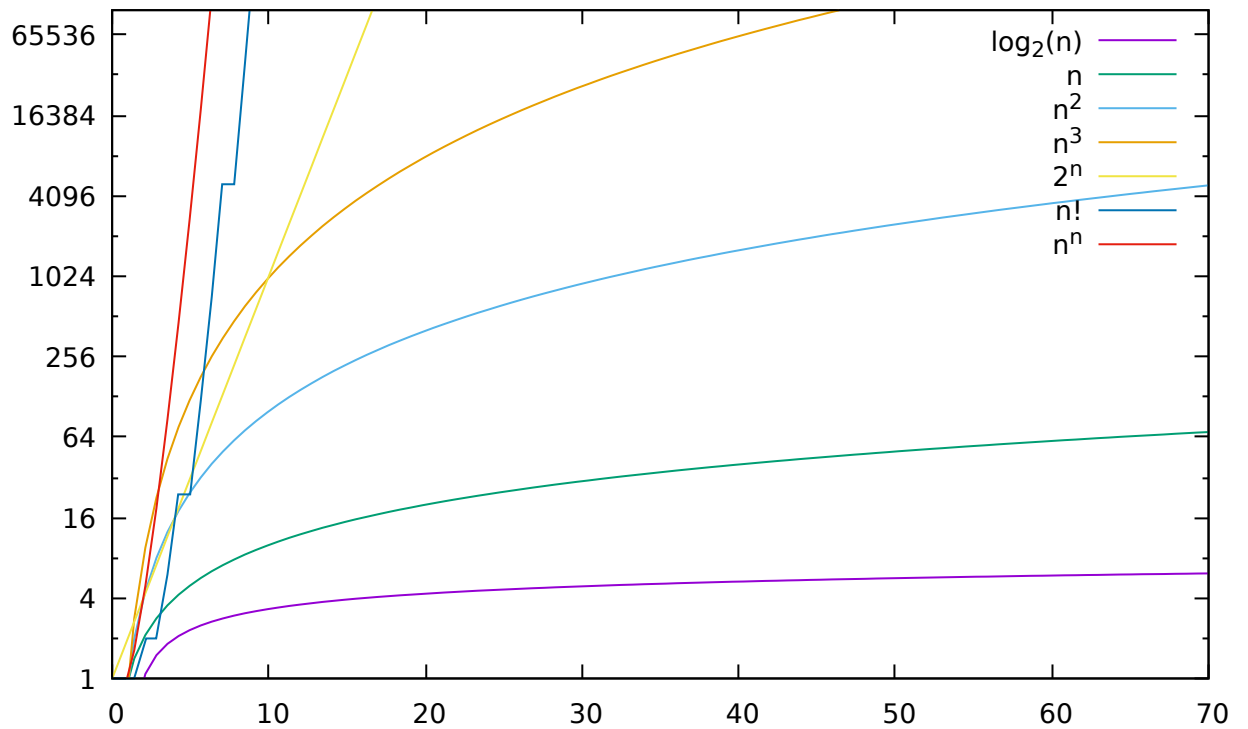


FIGURE 3.1 – Représentation des différentes fonctions  $\log_2 n, n, n^2, n^3, 2^n, n!, n^n$  en échelle logarithmique



**Définition 2.  $O$  « Borne Supérieure »**

Soient  $f$  et  $g : \mathbb{N} \rightarrow \mathbb{R}_+ : f = O(g)$  ssi  $\exists c \in \mathbb{R}_+, \exists n_0 \in \mathbb{N}$  tels que :

$$\forall n > n_0, f(n) \leq c \times g(n)$$

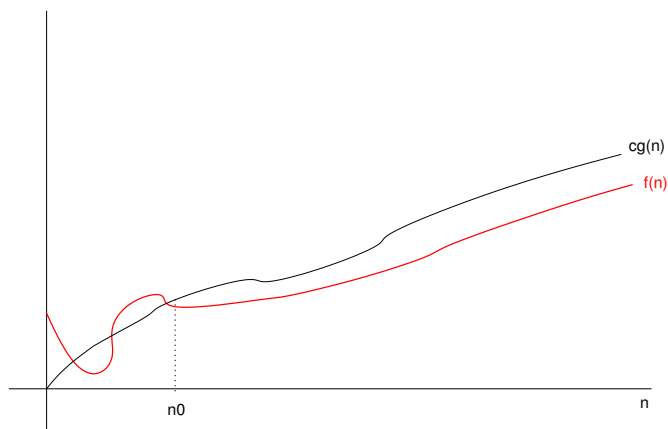


FIGURE 3.2 –  $f(n) = O(g(n))$

**Définition 3.  $\Omega$  « Borne Inférieure »**

Soient  $f$  et  $g : \mathbb{N} \rightarrow \mathbb{R}_+ : f = \Omega(g)$  ssi  $\exists c \in \mathbb{R}_+, \exists n_0 \in \mathbb{N}$  tels que :

$$\forall n > n_0, 0 \leq c \times g(n) \leq f(n)$$

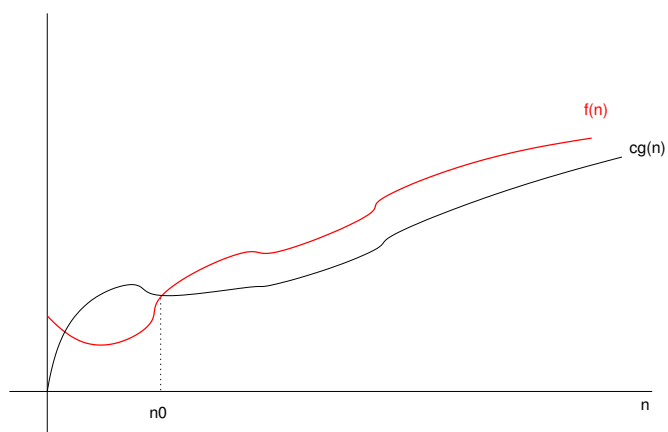
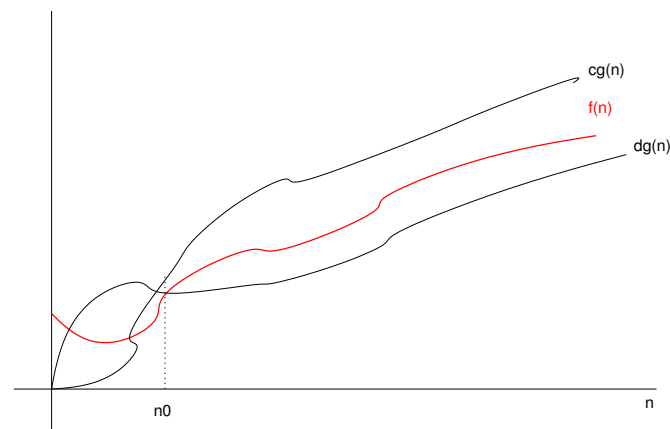


FIGURE 3.3 –  $f(n) = \Omega(g(n))$

**Définition 4.  $\Theta$** 

$f = \Theta(g)$  ssi  $f = O(g)$  et  $f = \Omega(g)$   
 $\exists c, d \in \mathbb{R}_+, \exists n_0 \in \mathbb{N}$  tels que :

$$\forall n > n_0, d \times g(n) \leq f(n) \leq c \times g(n)$$

FIGURE 3.4 –  $f(n) = \Theta(g(n))$ 

## 3.2 Etudes de complexité

### 3.2.1 Structures Linéaires

Eléments d'un même type stockés dans :

- un tableau
- une liste

**Opérations sur les structures linéaires**

- Insérer un nouvel élément
- Supprimer un élément
- Rechercher un élément
- Afficher l'ensemble des éléments
- Concaténer deux ensembles d'éléments
- ...

**Définition des structures**

- Un tableau  
Enregistrement Tab {  
    T[NMAX] : entier;  
    Fin : entier;  
}

### 3.2.2 Recherche et insertion dans les structures non triées

---

#### Algorithme 3.1 Recherche dans un tableau non trié

---

```
Recherche(S : Tab, x : entier) : booléen
▷ Entrées : S (un tableau), x (élément recherché)
▷ Sortie : vrai si l'élément x a été trouvé dans le tableau S, faux sinon.
Debut
▷ Variable Locale
  i : entier;
pour i de 1 à S.Fin faire
  si (S.T[i] = x)
    retourner vrai;
fin si
fin pour
retourner faux;
Fin
```

- **Opération fondamentale** : comparaison
- **A chaque itération** :

- 1 comparaison (Si ... Fin Si)
- 1 comparaison (Pour ... Fin Pour)
- **Nombre d'itérations maximum** : nombre d'éléments du tableau
- **Complexité** : Si  $n$  est le nombre d'éléments du tableau  $O(n)$ .

---

**Algorithme 3.2 Insertion dans un tableau non trié**


---

Insertion( $S$  : Tab,  $x$  : entier)  
 ▷ Entrées :  $S$  (un tableau),  $x$  (élément à insérer)  
 ▷ Sortie : le tableau  $S$  dans lequel  $x$  a été inséré.  
 Debut  
      $S.Fin \leftarrow S.Fin + 1$ ;  
      $S.T[S.Fin] \leftarrow x$ ;  
 Fin

- **Opération fondamentale** : affectation
- **Nombre d'opérations fondamentales** : 2 affectations.
- **Complexité** :  $O(1)$  (Temps constant).

### 3.2.3 Recherche et insertion dans les structures triées

---

**Algorithme 3.3 Insertion dans un tableau trié**


---

Insertion( $S$  : Tab,  $x$  : entier)  
 ▷ Entrées :  $S$  (un tableau),  $x$  (élément recherché)  
 ▷ Sortie : le tableau  $S$  dans lequel  $x$  a été inséré.  
 ▷ Pré-condition : le tableau  $S$  trié par ordre croissant.  
 ▷ Variable Locale  
      $i, k$  : entiers;  
 Debut  
     si ( $S.Fin = -1$ )  
          $S.Fin \leftarrow 0$ ;  
          $S.T[S.Fin] \leftarrow x$ ;  
     sinon  
          $i \leftarrow 0$ ;  
         tant que ( $i < S.Fin$  et  $S.T[i] < x$ )  
              $i \leftarrow i + 1$ ;  
         fin tant que  
         si ( $i = S.Fin$  et  $S.T[i] < x$ )  
              $k \leftarrow S.Fin + 1$ ;  
         sinon  
              $k \leftarrow i$ ;  
         fin si  
         pour  $i$  de  $S.Fin + 1$  à  $k$  en décroissant faire  
              $S.T[i] \leftarrow S.T[i-1]$ ;  
         fin pour  
          $S.T[k] \leftarrow x$ ;  
          $S.Fin \leftarrow S.Fin + 1$ ;  
     fin si  
 Fin

- **Opération fondamentale** : affectation
- **Recherche de la bonne position** :  $k$  affectations
- **Décaler à droite** :  $n - k$  affectations
- **Insérer élément** : 1 affectation
- **Total** :  $n + 2$  affectations
- **Complexité** :  $O(n)$  si  $n$  est le nombre d'éléments du tableau.

---

**Algorithme 3.4 Recherche dichotomique**


---

```

Recherche(x : entier, S : tableau, g : entier, d : entier) : booléen
▷ Entrées : x (élément recherché), S (espace de recherche), g (indice de gauche), d (indice de droite)
▷ Sortie : vrai si l'élément x a été trouvé dans le tableau S entre les indices g et d, faux sinon.
▷ Pré-conditions : g et d sont des indices valides du tableau S et S est trié par ordre croissant.
▷ Variable Locale
    m : entier ;
Debut
    si (g < d)
        m ← ⌊(g+d)/2⌋ ;
        si (x = S.T[m])
            retourner vrai ;
        sinon si (x < S.T[m])
            retourner (Recherche(x,S,g,m-1)) ;
        sinon
            retourner (Recherche(x,S,m+1,d)) ;
        fin si
    sinon
        retourner faux ;
    fin si
Fin
    
```

- **Opération fondamentale** : comparaison
- **A chaque appel récursif**, on diminue l'espace de recherche par 2 et on fait au pire 2 comparaisons
- **Complexité** : Au pire on fera donc  $O(\log_2 n)$  appels et la complexité est donc en  $O(\log_2 n)$ .

#### 3.2.4 Résumé

Complexité de l'insertion

	Eléments triés	Eléments non triés
Tableau	$O(n)$	$O(1)$

Complexité de la recherche

	Eléments triés	Eléments non triés
Tableau	$O(\log_2 n)$	$O(n)$

### 3.3 Les tris

#### 3.3.1 tri par insertion

---

##### Algorithme 3.5 Tri par insertion

---

**TriInsertion**(*T* : tableau d'entiers, *TailleMax* : entier)

▷ **Variables Locales**

*TC, i, p, temp* : entiers

**Debut**

**pour** *TC* **de** 1 **à** *TailleMax* - 1 **faire**

*temp* ← *T*[*TC* + 1]

*p* ← 1

**tant que** *T*[*p*] < *temp* **faire**

*p* ← *p* + 1

**fin tant que**

**pour** *i* **de** *TC* **en** décroissant **à** *p* **faire**

*T*[*i*+1] ← *T*[*i*]

**fin pour**

*T*[*p*] ← *temp*

**fin pour**

**Fin**

Chercher la position *p*

Décaler les éléments

**Complexité pour  $n$  éléments**

- Le corps de la boucle est exécuté  $n - 1$  fois
- Une itération :
  - Recherche de la position :  $p$
  - Décalage des éléments :  $TC - p$
  - Total :  $TC$
- Au total :

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

La complexité du tri par insertion est en  $O(n^2)$ .

**3.3.2 Tri par permutation****Algorithme 3.6 Tri par permutation**


---

**TriPermutation**( $T$  : tableau d'entiers,  $TailleMax$  : entier)  
 ▷ *Variables Locales*  
    $i, TC$  : entiers  
 Debut  
   pour  $TC$  de 2 à  $TailleMax$  faire  
     pour  $i$  de  $TailleMax$  en décroissant à  $TC$  faire  
       si  $T[i-1] > T[i]$  faire  
          $T[i-1] \leftrightarrow T[i]$   
       fin si  
     fin pour  
 fin pour  
 Fin

---

**Complexité pour  $n$  éléments**

- Boucle externe :  $n - 2$  fois
- Boucle interne :  $TailleMax - TC$  fois
- Total :  $\frac{(n-1)(n-2)}{2}$

La complexité du tri par permutation est en  $O(n^2)$ .

**3.3.3 Tri fusion****Algorithme 3.7 Tri Fusion**


---

**TriFusion**( $T$  : tableau d'entiers,  $p$  : entier,  $r$  : entier)  
 ▷  $p$  et  $r$  sont les indices entre lesquels on veut trier le tableau. On suppose  $p \leq r$ .  
 Debut  
   si  $p < r$  faire  
      $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$   
     **TriFusion**( $T, p, q$ )  
     **TriFusion**( $T, q+1, r$ )  
     **Fusion**( $T, p, q, r$ )  
   fin si  
 Fin

---

**Algorithme 3.8 Tri Fusion**


---

**Fusion**( $T$  : tableau d'entiers,  $p$  : entier,  $q$  : entier,  $r$  : entier)  
 ▷ *Entrées* :  $T$  : tableau d'entiers.  $p, q$  et  $r$  : indices entre lesquels on veut trier le tableau avec  $p \leq q \leq r$ .  
 ▷ *Sortie* :  $T$  : tableau trié entre les indices  $p$  et  $r$ .  
 ▷ *Pré-condition* :  $T$  tableau trié entre les indices  $p$  et  $q$  et  $T$  trié entre les indices  $q+1$  et  $r$   
 ▷ *Variables locales* :  $i, j, k$  : entiers et  $B$  : tableau d'entiers

---

<pre> <b>Debut</b> i ← p ; k ← p ; j ← q + 1 ; <b>tant que</b> (i ≤ q et j ≤ r) <b>faire</b>   <b>si</b> T[i] &lt; T[j] <b>faire</b>     B[k] ← T[i]     i ← i + 1    <b>sinon</b>     B[k] ← T[j]     j ← j + 1    <b>fin si</b>   k ← k + 1 <b>fin tant que</b> </pre>	<pre> <b>tant que</b> i ≤ q <b>faire</b>   B[k] ← T[i]   i ← i + 1   k ← k + 1 <b>fin tant que</b> <b>tant que</b> j ≤ r <b>faire</b>   B[k] ← T[j]   j ← j + 1   k ← k + 1 <b>fin tant que</b> T ← B <b>Fin</b> </pre>
--	---

#### Complexité pour $n$ éléments

— Intuitivement il faut résoudre :

$$Tri(n) = 2 \times Tri\left(\frac{n}{2}\right) + \Theta(n)$$

—  $\Theta(n)$  : complexité de la fusion

La complexité du tri fusion est en  $\Theta(n \log_2 n)$ .

### 3.3.4 Tri rapide

---

#### Algorithme 3.9 Tri Rapide

---

```

TriRapide( $T$  : tableau d'entiers,  $p$  : entier,  $r$  : entier)
▷  $p$  et  $r$  sont les indices entre lesquels on veut trier le tableau. On suppose  $p \leq r$ .
Debut
  si  $p < r$  faire
     $q \leftarrow \text{partitionner}(T, p, r)$ 
    TriRapide( $T, p, q$ )
    TriRapide( $T, q+1, r$ )
  fin si
Fin

```

---

#### Algorithme 3.10 Partitionner

---

```

Partitionner( $T$  : tableau d'entiers,  $p$  : entier,  $r$  : entier)
▷  $p$  et  $r$  sont les indices entre lesquels on veut trier le tableau. On suppose  $p \leq r$ .
▷ Variables locales :  $i, j, \text{pivot}$  : entiers
Debut
 $i \leftarrow p$ ;  $j \leftarrow r$ ;  $\text{pivot} \leftarrow T[p]$ ;
tant que ( $i < j$ ) faire
  tant que ( $T[i] < \text{pivot}$ ) faire  $i \leftarrow i + 1$  fin tant que
  tant que ( $T[j] > \text{pivot}$ ) faire  $j \leftarrow j - 1$  fin tant que
  si ( $i < j$ ) faire
     $T[i] \leftrightarrow T[j]$ 
     $i \leftarrow i + 1$ 
     $j \leftarrow j - 1$ 
  fin si
fin tant que
retourner  $j$ 
Fin

```

---

#### Complexité pour $n$ éléments

- Partitionner  $n$  éléments coûte  $\Theta(n)$ .
- Temps d'exécution dépend de l'équilibre ou non du partitionnement :
  - S'il est équilibré : **aussi rapide que le tri fusion**
  - S'il est déséquilibré : **aussi lent que le tri par insertion**

## 3.4 Les structures de données abstraites

- Mise en œuvre d'un ensemble dynamique
- Définition de données (**structuration** et **propriétés**)
- Définition des opérations pour **manipuler** les données.

#### Quelques structures classiques

1. Pile
2. File
3. Tables de hachage
4. Dictionnaire
5. Tas
6. Files de priorité
7. Arbres
8. ....

### 3.4.1 Les Piles

Analogie avec une pile d'assiette :

*LIFO* (Last In First Out ou **Dernier Arrivé Premier Servi**)

- On ne peut rajouter un élément qu'au dessus de la pile
- On ne peut prendre que l'élément qui est au dessus de la pile (élément le plus récemment inséré).

Mise en œuvre à l'aide d'un tableau (*nombre maximum d'éléments dans la pile fixé*)

```
Enregistrement Pile {  
    T[NMAX] : entier;  
    Sommet : entier;  
}
```

---

#### Algorithme 3.11 La pile est-elle vide ?

---

```
PileVide(p : Pile) : booléen  
▷ Entrée : P (une pile)  
▷ Sortie : vrai si la pile est vide, faux sinon.  
Debut  
si (p.Sommet = -1)  
    retourner vrai;  
sinon  
    retourner faux;  
fin si  
Fin
```

Complexité :  $O(1)$

---

#### Algorithme 3.12 La pile est-elle pleine ?

---

```
PilePleine(p : Pile) : booléen  
▷ Entrée : P (une pile)  
▷ Sortie : vrai si la pile est pleine, faux sinon.  
Debut  
si (p.Sommet = NMAX-1)  
    retourner vrai;  
sinon  
    retourner faux;  
fin si  
Fin
```

Complexité :  $O(1)$

---

#### Algorithme 3.13 Insertion d'un élément

---

```
Insertion(p : Pile, elt : entier)  
▷ Entrée : p (une pile) et elt (un entier)  
▷ Sortie : la pile p dans laquelle elt a été inséré  
Debut  
si (PilePleine(p) = faux)  
    p.Sommet  $\leftarrow$  p.Sommet + 1;  
    p.T[p.Sommet]  $\leftarrow$  elt;  
sinon  
    Afficher un message d'erreur  
fin si  
Fin
```

Complexité :  $O(1)$

---

#### Algorithme 3.14 Suppression d'un élément

---

```
Suppression(p : Pile) : entier
```



```

▷ Entrée :  $p$  (une pile)  $e$ 
▷ Sortie : renvoie l'élément qui était au sommet de la pile  $p$  et supprime l'élément de la pile
▷ Variable locale :
  elt : entier ;
  Debut
  si (PileVide( $p$ ) = faux )
    elt  $\leftarrow$   $p.T[p.Sommet]$  ;
     $p.Sommet \leftarrow p.Sommet - 1$  ;
    retourner elt ;
  sinon
    Afficher un message d'erreur
  fin si
  Fin

```

Complexité :  $O(1)$

## 3.5 Les arbres

### 3.5.1 Définitions

#### Définition 5.

- Un **arbre** : un ensemble de nœuds reliés entre eux par des arêtes.
- Trois propriétés pour les arbres **enracinés** :
  1. Il existe un nœud particulier nommé **racine**.
  2. Tout nœud  $c$  autre que la racine est relié par une arête à un nœud  $p$  appelé **père** de  $c$ .
  3. Un arbre est **connexe**.

- Un nœud peut avoir 0 ou plusieurs fils.
- Un nœud a exactement un père.

#### Une définition récursive :

- **Base** :
  - Un nœud unique  $n$  est un arbre
  - $n$  est la racine de cet arbre.
- **Récurrence** :
  - Soit  $r$  un nouveau nœud
  - $T_1, T_2, \dots, T_k$  sont des arbres ayant pour racine  $r_1, r_2, \dots, r_k$ .
  - Création d'un nouvel arbre ayant pour racine  $r$  et on ajoute une arête entre  $r$  et  $r_1$   $r$  et  $r_2, \dots, r$  et  $r_k$ .

#### Définition 6.

- Les **ancêtres** d un nœud : Nœuds trouvés sur le **chemin unique** entre ce nœud et la racine.

#### Définition 7.

- Le nœud  $d$  est un **descendant** de  $a$  si et seulement si  $a$  est un ancêtre de  $d$ .

#### Définition 8.

- **Longueur** d'un chemin = nombre d'arêtes parcourues.

#### Définition 9.

- Les nœuds ayant le même père = **frères**.

#### Définition 10.

■ Un nœud  $n$  et tous ses descendants = **sous-arbre**

**Définition 11.**

Une **feuille** est un nœud qui n'a pas de fils. Un **nœud intérieur** est un nœud qui a au moins 1 fils. Tout nœud de l'arbre est :

- Soit une feuille
- Soit un nœud intérieur

**Définition 12.**

La **hauteur d'un nœud**  $n$ , notée  $h(n)$ , est la longueur du chemin depuis la racine jusqu'à  $n$ . La **hauteur de l'arbre**  $T$ , notée  $h(T)$  :

$$h(T) = \max_{x \text{ nœud de l'arbre}} h(x)$$

**Définition 13.**

■ **Taille** de l'arbre  $T$ , notée  $taille(T)$  = nombre de nœuds.

**Définition 14.**

■ **Nombre de feuilles** noté  $nf(T)$ .

**Définition 15.**

**Longueur de cheminement** de l'arbre  $T$ , notée  $LC(T)$  = somme des longueurs de tous les chemins issus de la racine.

$$LC(T) = \sum_{x \text{ nœud de } T} h(x).$$

**Longueur de cheminement externe** de l'arbre  $T$ , notée  $LCE(T)$  = somme des longueurs de tous les chemins aboutissant à une feuille issus de la racine.

$$LCE(T) = \sum_{x \text{ feuille de } T} h(x).$$

**Définition 16.**

**Profondeur moyenne** de l'arbre  $T$ , notée  $PC(T)$  = moyenne des hauteurs de tous les nœuds.

$$PC(T) = \frac{LC(T)}{taille(T)}$$

**Profondeur moyenne externe** de l'arbre  $T$ , notée  $PCE(T)$  = moyenne des longueurs de tous les chemins issus de la racine et se terminant par une feuille.

$$PCE(T) = \frac{LCE(T)}{nf(T)}$$

### 3.5.2 les arbres binaires

**Définition 17.**

■ Tous les nœuds d'un arbre binaire ont 0, 1 ou 2 fils.

- **Fils gauche** de  $n$  = racine du sous-arbre gauche de  $n$ .

— **Fils droit** de  $n$  = racine du sous-arbre droit de  $n$ .

**Définition 18.**

**Bord gauche** de l'arbre = le chemin depuis la racine en ne suivant que des fils gauche.

**Bord droit** de l'arbre = le chemin depuis la racine en ne suivant que des fils droits.

