# IN 303 – Structure de données

Exercices et Résumés de Cours

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

2020-2021

# 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 ;

# Chapitre 1

# Exercices

## 1.1 Complexité

**Exercice 1 – $\mathcal{P}$ 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 ?

**Exercice 2 – $\mathcal{P}$ 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
        i ← i + 1
    fin tant que
Fin
```

**Algorithme 1.5 E($m, n$ : entiers) : entier**

```
Debut
    i ← 1
    tant que (i ≤ m et i ≤ n) faire
        I ← i + 1
    fin tant que
Fin
```

**Algorithme 1.6 F($m, n$ : entiers) : entier**

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

Fin
```

**Algorithme 1.7 G**$(m, n$ **: entiers**$)$ **: entier**

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

      sinon
         j ← j + 1
      fin si

   fin tant que

Fin
```

**Algorithme 1.8 H**$(m, n$ **: entiers**$)$ **: entier**

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

      sinon
         j ← j + 1
         i ← 1

      fin si

   fin tant que

Fin
```

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

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

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

**Algorithme 1.9** $f(n$ **: entier** $)$ **: entier**

```
Debut
  r ← 0
   pour i de 1 à n  faire
     r ← 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** $g(n$ **: entier** $)$ **: entier**

```
Debut
   si (n = 0)
      retourner 0

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

Fin
```

**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.

**Exercice 5 – $\mathcal{D}$ 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-Ã -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 utilisez l'algorithme précédent.

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

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

**Exercice 7 – $\mathcal{D}$ 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 ?

**Exercice 8 – $\mathcal{D}$ 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 tous $0 \leq i \leq m-1$ et
— $t[i] \geq t[i+1]$ pour tous $m \leq i < n-1$
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.

**Exercice 9 – $\mathcal{D}$ 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 $O(n \log n)$) ?

## 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): booleen` : 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): booleen` : 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 – $\mathcal{P}$ 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. Ecrire 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 $P_1$.

**Exercice 11 – $\mathcal{P}$ 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.

**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.

# 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 $A[1], A[2], \ldots, A[n]$. 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 ↑ `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 ≤ n, et 0 sinon.
 Debut
▷ Variables locales
    i : entier ;
    som : entier ;

    som ← 0 ;
     pour i  de m  à n  faire
        som ← 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 ← ⌊(g+d)/2⌋ ;
     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**

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

## 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) montrer 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*[1] *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 externe.

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 :**

$$Min_{\mathcal{A}}(n) = \min\{Cout_{\mathcal{A}}(d), d \in D_n\}$$

— **Complexité dans le pire cas :**

$$Max_{\mathcal{A}}(n) = \max\{Cout_{\mathcal{A}}(d), d \in D_n\}$$

— **Complexité en moyenne :**

$$Moy_{\mathcal{A}}(n) = \sum_{d \in D_n} p(d) \times Cout_{\mathcal{A}}(d)$$

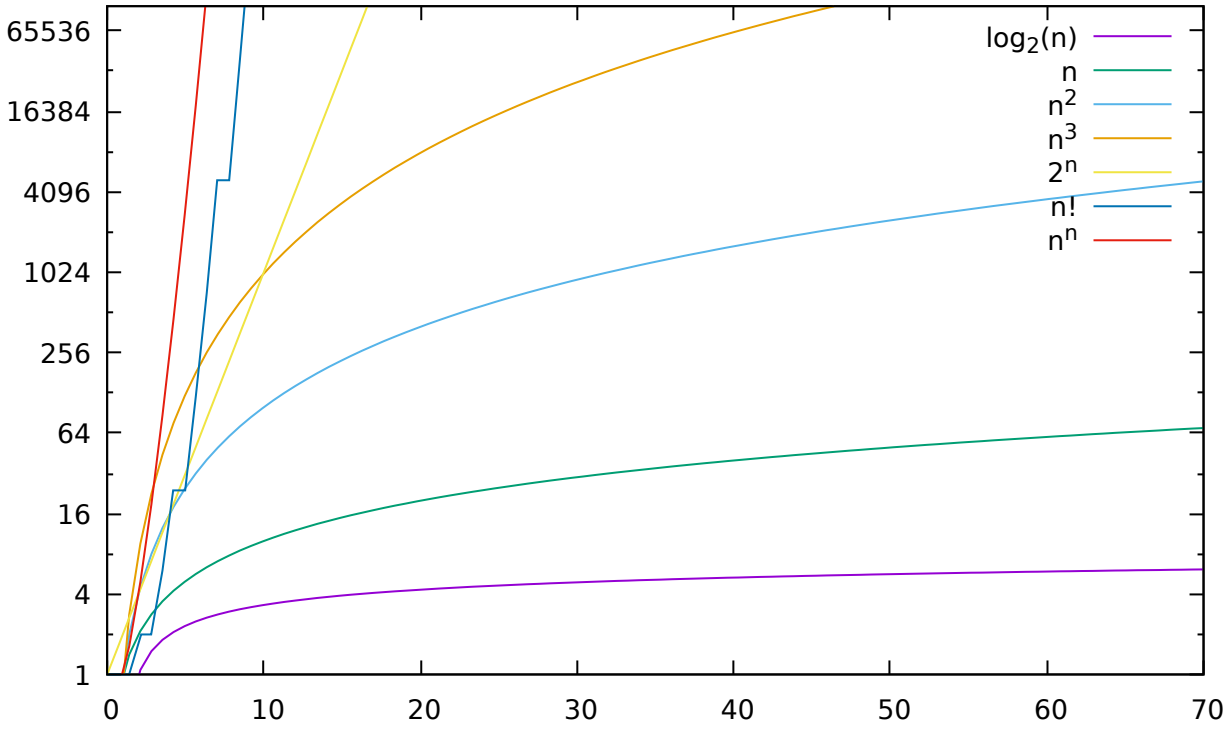


Figure 3.1 – Représentation des différentes fonctions $\log_2 n, n, n^2, n^3, 2^n, n!, n^n$ en échelle logarithmique

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

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

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

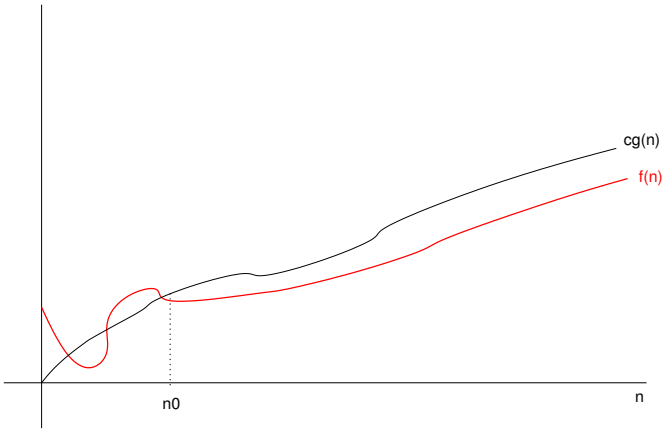


FIGURE 3.2 – $f(n) = O(g(n))$

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

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

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

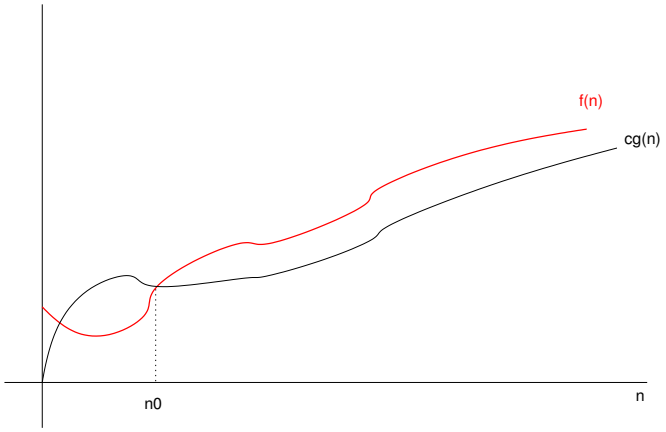


FIGURE 3.3 – $f(n) = \Omega(g(n))$

**Définition 4.** $\Theta$

$$f = \Theta(g) \text{ ssi } f = O(g) \text{ et } f = \Omega(g)$$

$\exists c, d \in \mathbb{R}_+, \exists n_0 \in \mathbb{N}$ tels que :

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

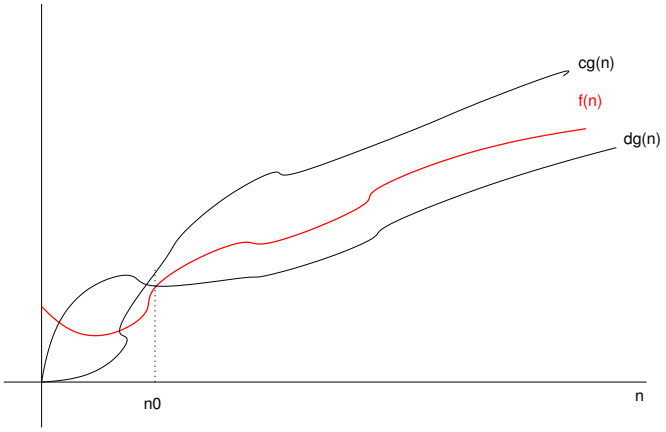


Figure 3.4 – $f(n) = \Theta(g(n))$

## 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'élements
- ...

**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 $\mathbf{O(n)}$.

**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 ← S.Fin + 1 ;
   S.T[S.Fin] ← x ;
Fin
```

- **Opération fondamentale :** affectation
- **Nombre d'opérations fondamentales** : 2 affectations.
- **Complexité :** $\mathbf{O(1)}$ (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 ← 0 ;
      S.T[S.Fin] ← x ;
   sinon
      i ← 0 ;
      tant que (i < S.Fin et S.T[i] < x)
         i ← i + 1 ;
      fin tant que
      si (i = S.Fin et S.T[i] < x)
         k ← S.Fin + 1 ;
      sinon
         k ← i ;
      fin si
      pour i de S.Fin + 1 à k en décroissant  faire
         S.T[i] ← S.T[i-1] ;
      fin pour
      S.T[k] ← x ;
      S.Fin ← 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 :** $\mathbf{O(n)}$ 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 $\mathbf{O}(\log_2 \mathbf{n})$ appels et la complexité est donc en $\mathbf{O}(\log_2 \mathbf{n})$.

### 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                    Chercher la position p
        p ← p + 1
    fin tant que
    pour i de TC en décroissant à p faire         Décaler les éléments
        T[i+1] ← T[i]
    fin pour
    T[p] ← temp
fin pour
Fin
```

**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] ↔ 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 ≤ r.
 Debut
     si p < r  faire
         q ← ⌊(p+r)/2⌋
         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 ≤ q ≤ 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
```

```
Debut
i ← p ; k ← p ; j ← q + 1 ;
tant que (i ≤ q et j ≤ r) faire
    si T[i] < T[j] faire
        B[k] ← T[i]
        i ← i + 1

    sinon
        B[k] ← T[j]
        j ← j + 1

    fin si
    k ← k + 1
fin tant que

tant que i ≤ q faire
    B[k] ← T[i]
    i ← i + 1
    k ← k + 1
fin tant que
tant que j ≤ r faire
    B[k] ← T[j]
    j ← j + 1
    k ← k + 1
fin tant que
T ← B
Fin
```

**Complexité pour $n$ éléments**

— Intuitivement il faut résoudre :

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

— $\Theta(n)$ : 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 ≤ r.
 Debut
     si p < r  faire
         q ← 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 ≤ r.
▷ Variables locales : i,j,pivot : entiers
 Debut
i ← p ; j ← r ; pivot ← T[p] ;
 tant que (i < j)  faire
     tant que (T[i] < pivot)  faire i ← i + 1  fin tant que
     tant que (T[j] > pivot)  faire j ← j - 1  fin tant que
     si (i < j)  faire
         T[i] ↔ T[j]
         i ← i + 1
         j ← 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 ← p.Sommet + 1 ;
    p.T[p.Sommet] ← 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 ← p.T[p.Sommet] ;
    p.Sommet ← p.Sommet - 1 ;
     retourner elt ;
sinon
    Afficher un message d'erreur
fin si
Fin
```

**Complexité :** $O(1)$