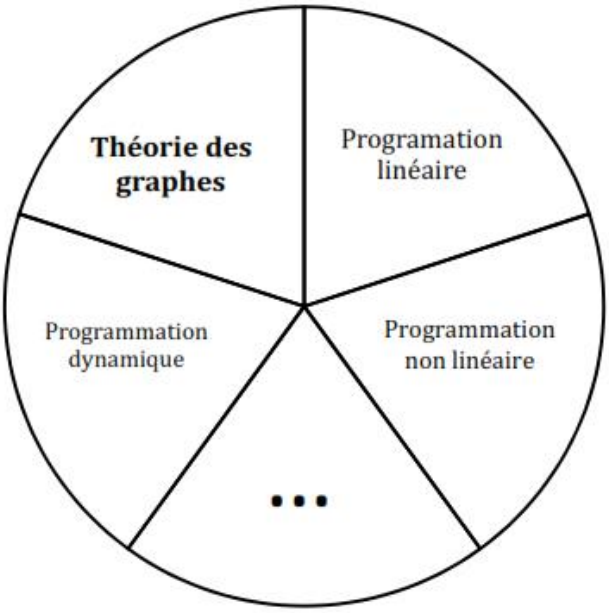
La recherche opérationnelle (RO) est la branche la plus importante de l'algèbre moderne. Elle regroupe un ensemble de méthodes et techniques rationnelles orientées vers la recherche de la meilleure façon d'opérer des choix en vue d'aboutir au meilleur résultat possible.

RECHERCHE OPÉRATIONNELLE



Les graphes sont actuellement l'outil privilégié pour modéliser des ensembles de structures complexes. Ils sont indispensables pour représenter et étudier des relations entre les objets.

Les graphes sont utilisés en :

- Economie (diagramme d'organisation)
- Electronique (circuits intégrés)
- Base de données (liens entre informations)
- Communications (réseaux de télécommunications)
- Transport (réseaux routiers)
- Ordonnancement (structure des projets)
- Etc.

A revoir le cours de la théorie des graphes dispensé en L2.
On s'intéressera principalement aux techniques d'optimisation (moindre coût, recherche du chemin optimal).

CHAPITRE 1 : NOTIONS FONDAMENTALES

Plan

1. Notion de graphe
2. Graphes particuliers
3. Chemins (chaines) et circuits (cycles)
4. Modes de représentation d'un graphe

1. NOTION DE GRAPHE

1.1. Graphe

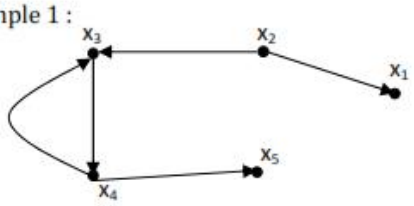
Un graphe (orienté) G est un couple $G(X, U)$ défini par :

- Un ensemble $X = \{x_1, x_2, \dots, x_n\}$ dont les éléments sont appelés **sommets** (ou **nœuds**).
- Un ensemble $U = \{u_1, u_2, \dots, u_m\}$ dont les éléments sont appelés **arcs.** Un arc est un couple ordonné de sommets ; l'ensemble U est donc obtenu par le produit cartésien :

$$U = X \times X = \{(x_i , x_j) / x_i \in X \; et \; x_j \in X\}$$

- L'arc (x_i , x_j) du graphe a x_i comme **extrémité initiale** et x_j comme **extrémité finale** (ou **terminale**).
- Le nombre n de sommets d'un graphe est dit **ordre d'un graphe**
- Graphiquement, les sommets d'un graphe sont représentés par des points et les arcs par des flèches reliant les sommets suivant un ordre.

Exemple 1 :



$X = \{x_1, x_2, x_3, x_4, x_5\}$

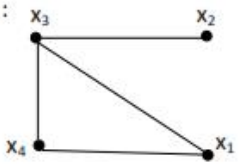
$U = \{(x_2, x_1), (x_2, x_3), (x_3, x_4), (x_4, x_3), (x_4, x_5)\}$

Ordre du graphe $n = 5$ *sommets*

1.2. Arête

Une arête est un arc non orienté.

Exemple 2 :

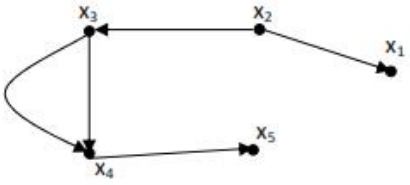


Une arête est représentée par un couple non ordonné noté avec des crochets comme suit :
$[x_1, x_3]$, $[x_2, x_3]$, ...

1.3. p-graphe

Un p-graphe est un graphe dans lequel il existe au maximum p arcs de la forme (x_i , x_j) entre deux sommets quelconques x_i et x_j pris dans cet ordre. Si $p = 1$, on parle alors de 1-graphe ou graphe.

Exemple 3 : (2-graphe)



CHAPITRE 2 : CONNEXITE

Plan
1. Transitivité
2. Connexité
3. Recherche de circuits et chemins hamiltoniens

La notion de connexité est liée à l'existence de chemins dans un graphe. Elle permet essentiellement de répondre à la problématique suivante : depuis un sommet, existe-t-il un chemin pour atteindre tout autre sommet ? Elle permet entre autre de déterminer les éléments du graphe (sommet et/ou arc) ayant une importance particulière selon le contexte.

La notion de connexité est importante dans plusieurs domaines :
- Réseaux : conception de réseaux fiables et résistants aux pannes.
- Topologie : pour repérer les espaces connexes ou non connexes.
- Sociologie : pour étudier l'interaction entre les individus.
- ...

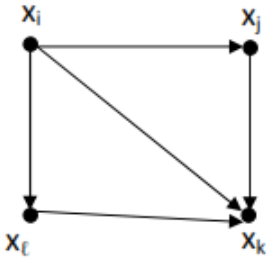
1. TRANSITIVITE

1.1. Graphe transitif

Un graphe transitif est un graphe tel que pour tout couple de sommets x_i et x_k reliés par un chemin de longueur 2, sont aussi reliés par un arc ; autrement dit, $G(X, U)$ est transitif si :

$$\forall\, \boldsymbol{x_i, x_j, x_k \in X},\ \ \boldsymbol{si}\ (x_i, x_j) \in \boldsymbol{U}\ \boldsymbol{et}\ (x_j, x_k) \in \boldsymbol{U}\ \ \boldsymbol{alors}\ (x_i, x_k) \in \boldsymbol{U}$$

Exemple 1 :



← *Graphe transitif*

Notation :

Notons par $\Gamma^1 = \Gamma^+$ (ensemble de successeurs) et $\Gamma^{-1} = \Gamma^-$ (ensemble de prédécesseurs). On a donc :

$$\Gamma^1(x_1, x_2, \dots, x_n) = \Gamma^1(x_1) \cup \Gamma^1(x_2) \cup \dots \cup \Gamma^1(x_n) = \bigcup_{i=1}^{n} \Gamma^1(x_i)$$

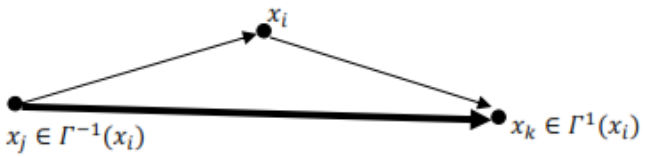
$$\Gamma^{-1}(x_1, x_2, \dots, x_n) = \Gamma^{-1}(x_1) \cup \Gamma^{-1}(x_2) \cup \dots \cup \Gamma^{-1}(x_n) = \bigcup_{i=1}^{n} \Gamma^{-1}(x_i)$$

- **Algorithme de recherche de la fermeture transitive stricte d'un graphe $G(X, \Gamma)$**

Cet algorithme permet d'obtenir $\hat{G}(X, \hat{\Gamma})$ connaissant $G(X, \Gamma)$. On pose $X = \{x_1, x_2, \dots, x_n\}$ l'ensemble des sommets du graphe G, $\Gamma^1(x_i)$ l'ensemble des successeurs du sommet x_i et $\Gamma^{-1}(x_i)$ l'ensemble des prédécesseurs de x_i .

Principe de définition de l'algorithme

$\theta_i$est une application qui consiste à enrichir G en joignant par un arc (lorsqu'il n'existe pas) chaque élément (sommet) de $\Gamma^{-1}(x_i)$ à chaque élément (ou sommet) de $\Gamma^1(x_i)$.



Si on considère un graphe représenté par une matrice booléenne (incidence sommet-sommet), θ_i consiste à reproduire tous les "1" existants en ligne "i" sur toute ligne "k" possédant un "1" en colonne "i".

	1	2	3	4	…	i	…	k	…
1									
2									
3									
…									
i	1	0	0	1	…		1	0	…
…	↓			↓			↓		
k	1			1		**1**	1	…	…
…									

Principe d'application de l'algorithme

Etant donné un graphe G :

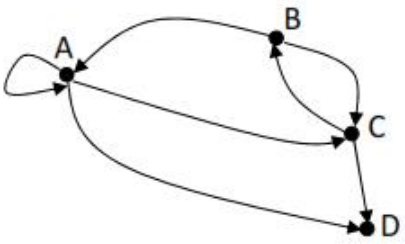
1. Numéroter ses sommets dans un ordre arbitraire.
2. Déterminer la matrice booléenne correspondante au graphe.
3. Appliquer θ_1($1^{\text{ère}}$ ligne de la matrice), puis θ_2 ($2^{\text{ème}}$ ligne de la matrice) au résultat obtenu, ensuite θ_3($3^{\text{ème}}$ ligne), … et enfin θ_n (dernière ligne).

Le résultat obtenu (final) est la fermeture transitive stricte du graphe initial G.

Important :

Un "1" dans la case (x_i, x_j) de la matrice finale signifie qu'il existe un chemin de x_i vers x_j dans G.

Exemple 4 :
Soit à appliquer l'algorithme au graphe suivant pour déterminer sa fermeture transitive stricte.



	A	B	C	D
A				
B				
C				
D				

case(A , B) = ⇒

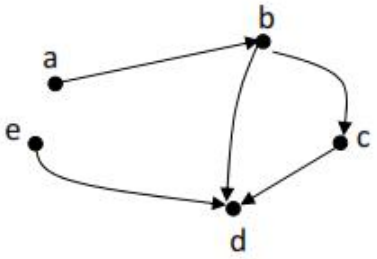
case(D , B) = ⇒

2. CONNEXITE

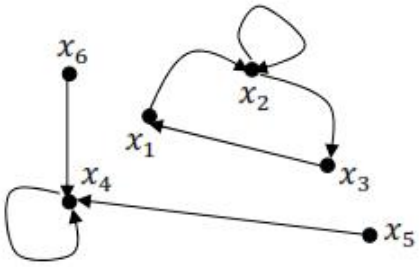
2.1. Graphe connexe (ou simplement connexe)

Un graphe connexe est un graphe tel qu'à partir de tout sommet x_i on peut atteindre tout autre sommet x_j en passant par une chaine du graphe.

Exemple 5 :



G1 est un graphe connexe



G2 est un graphe non connexe
(2 composantes connexes)

2.2. Composantes connexes d'un graphe

Soit x_i un sommet donné et C_{x_i} l'ensemble des sommets pouvant être reliés à x_i par une chaine y compris aussi x_i ; on appelle composante connexe du graphe, le sous-graphe engendré par un ensemble de la forme C_{x_i} .

Exemple 6 : (Graphe 2, Exemple 5)
Les deux sous graphes formés par (x_1 , x_2 , x_3) et (x_4 , x_5 , x_6) sont les deux composantes connexes du graphe $G2$.

Les différentes composantes connexes du graphe $G(X, \Gamma)$ constituent une partition de classes de X ; c'est-à-dire $si\ x_i \in X$:

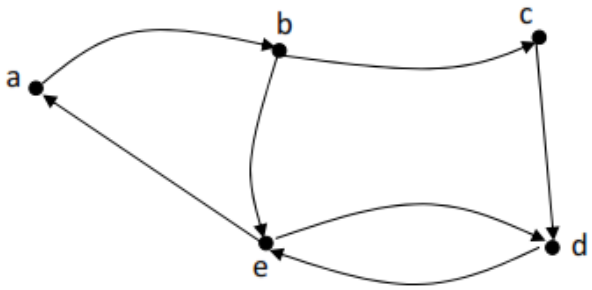
1. $C_{x_i} \neq \emptyset$
2. $C_{x_i} \neq C_{x_j} \Rightarrow C_{x_i} \cap C_{x_j} = \emptyset$
3. $\cup\, C_{x_i} = X$

Remarque :
Un graphe est connexe si et seulement s'il ne possède qu'une seule composante connexe (Graphe $G1$ de l'exemple 5).

2.3. Graphe fortement connexe

Un graphe $G(X,\Gamma)$ est fortement connexe si et seulement si : $\forall x_i \in X\ ,\ \hat{\Gamma}(x_i) = X$

Autrement dit, de tout sommet x_i on peut atteindre tout autre sommet x_j en suivant un chemin du graphe, comme c'est le cas du graphe suivant :



Remarque :
- Graphe fortement connexe ⇒ graphe connexe (et non l'inverse).
- Graphe connexe et symétrique ⇒ graphe fortement connexe.

2.4. Composantes fortement connexes

On considère un graphe $G(X,\Gamma)$ et la relation d'équivalence R définie par :

$$\forall x_i, x_j \in X \qquad x_i\ R\ x_j \Leftrightarrow \begin{cases} x_i = x_j \\ ou \\ Il\ existe\ un\ chemin\ de\ x_i\ vers\ x_j\ et\ réciproquement \end{cases}$$

Propriétés :
- R réflexive $\quad \forall x_i \in X\ ,\ x_i\ R\ x_i$
- R symétrique $\quad \forall x_i, x_j \in X \qquad x_i\ R\ x_j \Rightarrow x_j\ R\ x_i$
- R transitive $\quad \forall x_i, x_j, x_k \in X \qquad x_i\ R\ x_j\ et\ x_j\ R\ x_k \Rightarrow x_i\ R\ x_k$

R est donc une relation d'équivalence.

- $C_1, C_2, \dots, C_n$ est l'ensemble des classes d'équivalence de l'ensemble quotient X/R
- Chaque classe d'équivalence C_i est un sous graphe appelé composante fortement connexe.

Remarque :
- Les composantes fortement connexes constituent une partition de l'ensemble X.
- Un graphe fortement connexe n'a qu'une seule composante fortement connexe.

2.5. Algorithme de Malgrange : une méthode de décomposition d'un graphe en composantes fortement connexes.

Soit C_{x_i} une classe d'équivalence, alors $C_{x_i} = \hat{\Gamma}(x_i) \cap \hat{\Gamma}^{-}(x_i)$ puisqu'il doit exister un chemin de x_i vers les autres sommets de la classe et inversement ; c'est-à-dire $\forall x_j \in X$:

$$x_j \in C_{x_i} \Leftrightarrow \exists\ un\ chemin\ de\ x_i\ vers\ x_j\ et\ inversement$$

Principe de définition

- L'algorithme consiste à sélectionner arbitrairement un sommet x_i et à calculer $\hat{\Gamma}(x_i)$ puis $\hat{\Gamma}^-(x_i)$ et enfin $\hat{\Gamma}(x_i) \cap \hat{\Gamma}^-(x_i)$: ceci nous donne un sous-graphe fortement connexe contenant x_i .
- On supprime alors les sommets du sous-graphe obtenu et on recommence avec un autre sommet.
- Pratiquement, on part de la matrice booléenne représentant le graphe et on lui ajoute une colonne correspondant à $\hat{\Gamma}$ et une ligne correspondant à $\hat{\Gamma}^-$ d'un sommet considéré.

Principe d'application

	A	B	C	D	E	F	G	H	I	J	K	$\hat{\Gamma}(A)$
A				***1***		***1***			***1***			***0***
B				***1***				***1***	***1***	***1***		×
C						***1***						×
D							***1***					***1***
E							***1***					×
F	***1***			***1***								***1***
G							***1***				***1***	***2***
H								***1***	***1***		***1***	×
I							***1***					***1***
J		***1***									***1***	×
K									***1***			***3***

$\hat{\Gamma}^-(A)$	A	B	C	D	E	F	G	H	I	J	K
	0	×	***2***	×	×	***1***	×	×	×	×	×

- On choisit un sommet quelconque. Prenons par exemple le sommet A, pour remplir $\hat{\Gamma}(A)$, placer 0 dans la case A de $\hat{\Gamma}(A)$. La ligne A porte un 1 en colonne D, on placera donc un 1 dans la case D de $\hat{\Gamma}(A)$. De même, on placera 1 dans les cases F et I de $\hat{\Gamma}(A)$ (ainsi le plus court chemin de A à D, F, I est de longueur 1).
- On considère ensuite les lignes D, F, I, on placera 2 dans les cases de $\hat{\Gamma}(A)$ correspondantes aux colonnes portant des 1, dans le cas où ces cases ne sont pas encore remplies. Dans notre exemple, seulement un 2 pour la case G, car les autres (A et D) sont déjà remplies.
- On considère alors la ligne G, elle contient 1 en colonne G (déjà remplie) et 1 en colonne K ; on porte alors 3 dans K de $\hat{\Gamma}(A)$.
- On examine la ligne K qui contient 1 en colonne I (déjà remplie) on voit qu'on ne peut plus appliquer la procédure, ainsi pour terminer on met une croix dans les cases restantes vides de $\hat{\Gamma}(A)$ (il n'y a pas de chemin entre A et ces sommets).

- Pour remplir la ligne $\hat{\Gamma}^-(A)$, on opère de la même manière, mais en remplaçant les lignes par les colonnes. On trouve pour A :

$$\left.\begin{array}{l} \hat{\Gamma}(A) = \{A, D, F, G, I, K\} \\ \hat{\Gamma}^-(A) = \{A, C, F\} \end{array}\right\} \Rightarrow \hat{\Gamma}(A) \cap \hat{\Gamma}^-(A) = \{A, F\}$$

$\{A, F\}$ est la première composante (1er sous graphe) fortement connexe (maximal).

- On enlève les sommets du sous graphe obtenu, puis on choisit un sommet arbitraire parmi les sommets restants et on applique à nouveau la procédure.

A la fin, on arrive à décomposer le graphe G en 7 classes d'équivalence correspondant chacune à une composante fortement connexe maximale.

$$C_1 = \{A, F\}\ ,\ C_2 = \{B, J\}\ ,\ C_3 = \{G, I, K\}\ ,\ C_4 = \{H\}\ ,\ C_5 = \{C\}\ ,\ C_6 = \{D\}\ ,\ C_7 = \{E\}$$

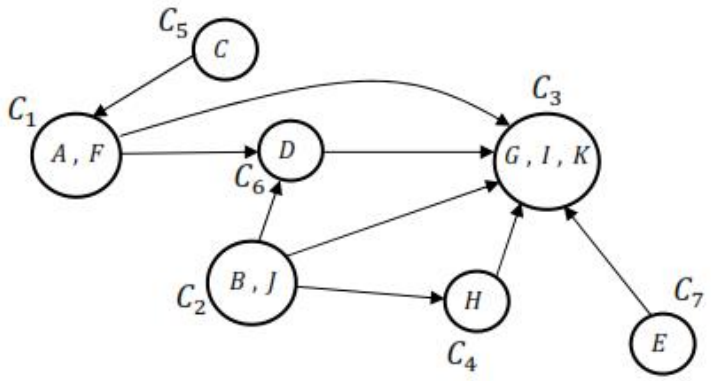
Graphe réduit

$G_r(C_f, U_r)$ est appelé graphe réduit de $G(X, U)$ s'il est défini par :

C_f : ensemble de composantes fortement connexe.

$(C_i, C_j) \in U_r$ s'il existe $\begin{Bmatrix} x_i \in C_i \\ x_j \in C_j \end{Bmatrix}$ *tel que* $(x_i, x_j) \in U$

Pour notre d'exemple d'application, on obtient le graphe réduit suivant qui permet de "résumer" le graphe initial, d'analyser ses "points faibles" et de déterminer quel est le minimum d'arcs à ajouter pour rendre le graphe fortement connexe.



3. RECHERCHE DE CIRCUITS ET CHEMINS HAMILTONIENS

3.1. Recherche de circuits d'un graphe

- Supprimer d'abord les sommets isolés.
- Commencer par chercher les boucles (circuits particuliers).
- Puis chercher les autres circuits du graphe par l'une des deux méthodes suivantes :

Méthode 1 : Sur le tracé du graphe (représentation sagittale)

a. Poser $i = 1$ (i : indice des sommets du graphe)
b. Repérer le sommet i
c. Les arcs du sommet i sont-ils tous aboutissants ou tous partants ?
   si oui aller à d
   sinon $i = i + 1$, aller à b
d. Effacer le sommet avec tous ses arcs ; poser $i = i + 1$; aller à b

La procédure s'arrête lorsque tous les sommets non supprimés ont à la fois des arcs partants et aboutissants. Si le graphe ne contient aucun circuit, la procédure supprime tout le graphe. Dans le cas contraire, seuls les circuits restent.

Méthode 2 : Sur la matrice booléenne du graphe

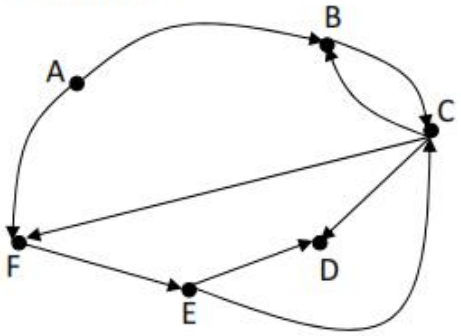
a. Poser $i = 1$ (indice des lignes ou colonnes de la matrice)
b. Repérer la case $(i\ ,i)$

c. La ligne i ou la colonne i contient-elle que des zéros (0) ?
   Si oui aller à d
   Sinon poser $i = i + 1$, puis aller à b
d. Barrer la ligne i et la colonne i ; poser $i = i + 1$; aller à b.

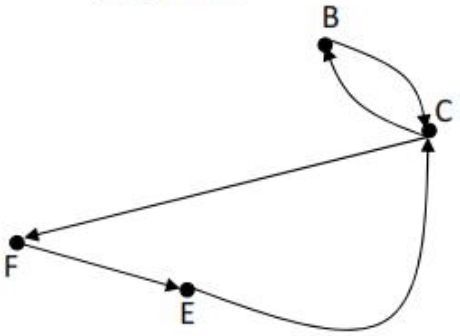
La procédure s'arrête d'elle-même, par épuisement.

Exemple 6 :

1ère méthode :



Etape finale



Les circuits sont :
CBC
CFEC
CBCFEC

2ème méthode :

	A	B	C	D	E	F
A	0	1	0	0	0	1
B	0	0	1	0	0	0
C	0	1	0	1	0	1
D	0	0	0	0	0	0
E	0	0	1	1	0	0
F	0	0	0	0	1	0

Résultat final

	B	C	E	F
B	0	1	0	0
C	1	0	0	1
E	0	1	0	0
F	0	0	1	0

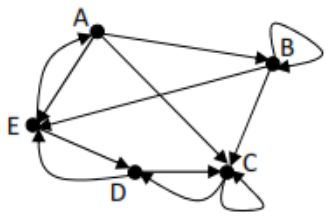
3.2. Recherche de chemins hamiltoniens

Longueur d'un chemin hamiltonien
La longueur d'un chemin hamiltonien (s'il existe) pour un graphe d'ordre n, est $\ell = n - 1$.

Méthode de recherche de chemins hamiltoniens
Principe de définition
Cette méthode est basée sur une multiplication matricielle d'un type particulier (matrice latine). Elle permet d'énumérer successivement tous les chemins élémentaires de longueur 1, 2, 3, ..., $n - 1$ (n étant l'ordre du graphe). On s'arrête une fois qu'on trouve tous les chemins de longueur $n - 1$ (chemins hamiltoniens) si le graphe en possède.

Principe d'application

Exemple 7 :
Soit à déterminer les chemins hamiltoniens du graphe suivant :



Nous commençons par représenter la matrice latine du graphe notée $[M]^1$ puisqu'elle nous donne tous les chemins élémentaires de longueur 1.

	A	B	C	D	E
A					
B					
C					
D					
E					

Nous déduisons de cette matrice une autre matrice latine notée $\left[\widetilde{M}\right]^1$ dans laquelle la première lettre sera enlevée :

	A	B	C	D	E
A					
B					
C					
D					
E					

Multiplions (d'une manière un peu différente) $[M]^1$ par $\left[\widetilde{M}\right]^1$ pour avoir $[M]^2$. On procède comme dans le calcul matriciel classique "ligne par colonne", mais au lieu d'effectuer des produits (ceci n'aurait pas de sens ici) ; lorsqu'une case $(i\,,j)$ sera en coïncidence avec une autre case $(j\,,k)$, on portera alors 0 dans la case $(i\,,k)$ de $[M]^2$. Si pour tous les j, l'une ou/et l'autre de ces cases contient un 0, on porte aussi 0, si pour toutes les cases traitées dans le "produit" ligne par colonne, on ne peut former une séquence ne contenant pas de lettres répétées en mettant à la suite d'une séquence de la case $(i\,,j)$ de $[M]^1$ une séquence de la case $(j\,,k)$ de $\left[\widetilde{M}\right]^1$, ceci pour tous les j (pour cet exemple les séquences de $\left[\widetilde{M}\right]^1$ se limitent chacune à 1 lettre, on généralisera par la suite). Enfin, si on met à la suite d'une séquence de la case $(i\,,j)$ de $[M]^1$ une séquence de la case $(j\,,k)$de $\left[\widetilde{M}\right]^1$, on trouve une plusieurs séquences sans répétition de lettre, on la (ou les) porte dans la case $(i\,,k)$ de $[M]^2$, et on procède ainsi pour tous les j.

Ce qui nous donne :

- $[M]^1 \boldsymbol{L} \left[\widetilde{M}\right]^1 = [M]^2$: qui nous donne la liste des chemins élémentaires de longueur 2.

	A	B	C	D	E
A					
B					
C					
D					
E					

- $[M]^2 \boldsymbol{L} \left[\widetilde{M}\right]^1 = [M]^3$: qui nous donne la liste des chemins élémentaires de longueur 3.

	A	B	C	D	E
A					
B					
C					
D					
E					

- $[M]^3 \boldsymbol{L} \left[\widetilde{M}\right]^1 = [M]^4$: qui nous donne la liste des chemins élémentaires de longueur 4.

	A	B	C	D	E
A					
B					
C					
D					
E					

On arrête à $[M]^4$ car $n = 5$. Il y a donc 7 chemins hamiltoniens :
(A , B , E , D , C) , (A , B , C , D , E) , (B , C , D , E , A) , (B , E , A , C , D) , (C , D , E , A , B) , (D , E , A , B , C) et (E , A , B , C , D).

Remarque :
La multiplication latine peut être étendue à des matrices $\left[\widetilde{M}\right]^q \quad q > 1$, définie comme suit :

$$[M]^p \boldsymbol{L} \left[\widetilde{M}\right]^q = [M]^{(p+q)}$$

Où $\left[\widetilde{M}\right]^q$ est la matrice $[M]^q$ dont on a enlevé à chaque séquence représentant un chemin élémentaire sa première lettre.

Exemple 8 :

3.3. Recherche de circuits hamiltoniens

Longueur d'un circuit hamiltonien

La longueur d'un circuit hamiltonien (s'il existe) pour un graphe d'ordre n, est $\ell = n$.

Exemple 9 :

Dans l'exemple 7 (A , B , C , D , E , A) , (B , C , D , E , A , B) , (C , D , E , A , B , C) sont des circuits hamiltoniens (équivalents car obtenus par permutation circulaire).

Méthode de recherche de circuits hamiltoniens

- Même méthode que celle de recherche de chemins hamiltoniens : il suffit de déterminer d'abord les chemins hamiltoniens du graphe donné, puis de fermer ces chemins obtenus par un arc du graphe si cela est possible.
- Si l'ordre du graphe est n, alors il suffit de calculer $[M]^{n-1}$ et de là on détermine $\left[\widetilde{M}\right]^{n-1}$.
- On calcule ensuite $[M]^{*}_{(n)} = \left[\widetilde{M}\right]^{n-1} L \left[\widetilde{M}\right]^{1}$, pour avoir les circuits hamiltoniens (s'il en existe), il suffit d'ajouter dans chaque séquence latine de $[M]^{*}_{(n)}$, comme lettre initiale (1ère lettre), la lettre constituant l'indice de la ligne de $[M]^{*}_{(n)}$.

Exemple 10 :

$\left[\widetilde{M}\right]^{n-1}$ L $\left[\widetilde{M}\right]^{1}$ $=$ $[M]^{*}_{(n)}$

	A	B	C	D	E
A	0	0	BEDC	0	BCDE
B	CDEA	0	0	EACD	0
C	0	DEAB	0	0	0
D	0	0	EABC	0	0
E	0	0	0	ABCD	0

L

	A	B	C	D	E
A	0	B	C	0	E
B	0	0	C	0	E
C	0	0	0	D	0
D	0	0	C	0	E
E	A	0	0	D	0

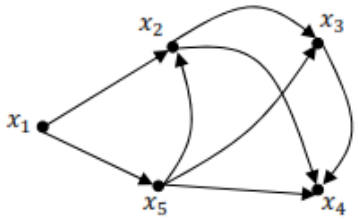
=

	A	B	C	D	E
A	ABCDEA	0	0	0	0
B	0	BCDEAB	0	0	0
C	0	0	CDEABC	0	0
D	0	0	0	DEABCD	0
E	0	0	0	0	EABCDE

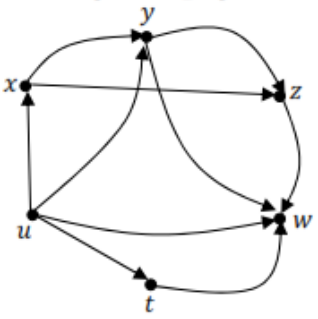
EXERCICES

CHAPITRE 2 –CONNEXITE

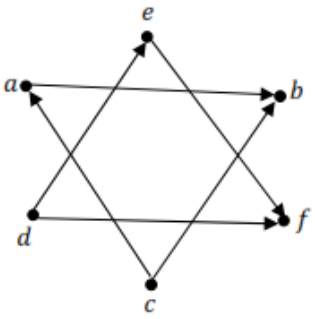
Exercice n°1 : Parmi les graphes suivants, indiquer les graphes transitifs :



G_a

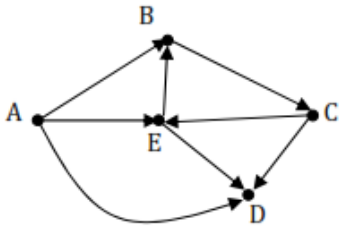


G_b

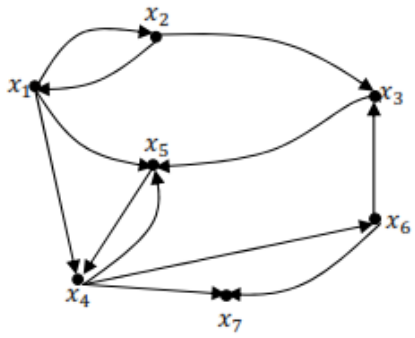


G_c

Exercice n°2 : Soient les 2 graphes suivants :



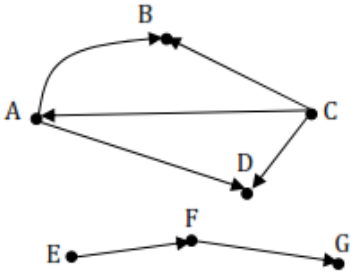
G_A



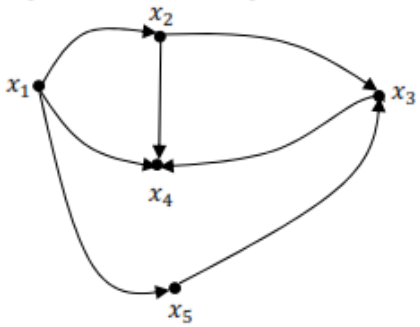
G_B

- Calculer pour G_A : $\Gamma^2(B)$, $\Gamma^3(C)$
- Calculer pour G_B : $\Gamma^{-2}(x_3)$, $\Gamma^{-3}(x_5)$
- Déterminer la fermeture transitive de G_A, puis celle de G_B

Exercice n°3 : Parmi les graphes suivants, indiquer ceux qui sont connexes, puis déterminer pour chacun d'eux, les composantes connexes.



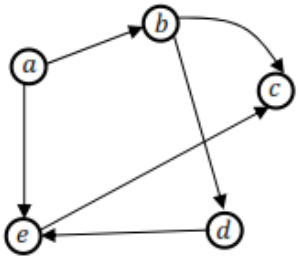
G_1



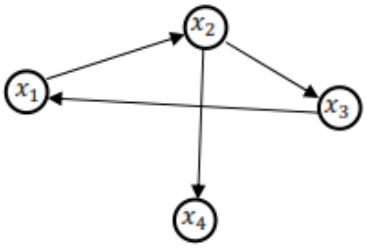
G_2

Exercice n°4 : Rechercher par deux méthodes différentes, les circuits s'il en existe sur les deux graphes de l'exercice précédent (exercice n°3). Pour le graphe G_2 remplacer l'arc (x_2, x_4) par (x_4, x_2).

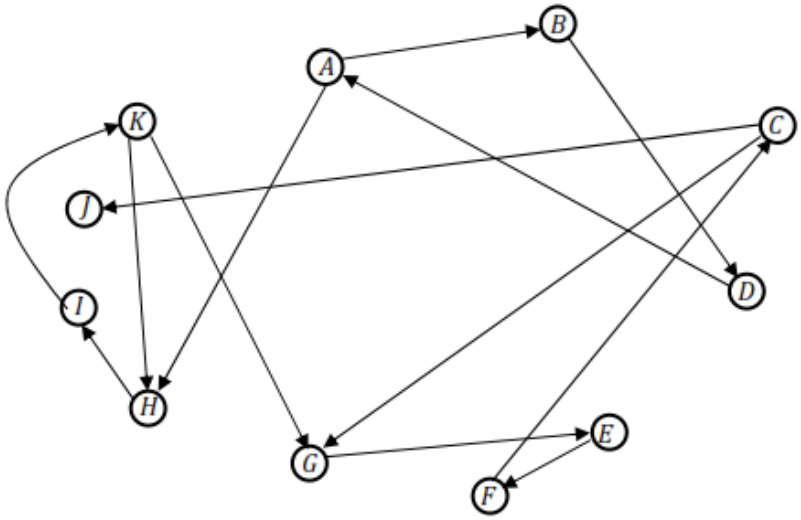
Exercice n°5 : Déterminer les composantes fortement connexes de chacun des graphes suivants, et tracer pour chaque graphe, son graphe réduit.



G1

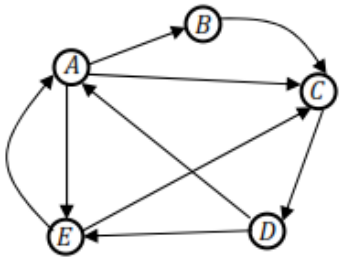


G2



G3

Exercice n°6 : Déterminer les chemins hamiltoniens, puis les circuits hamiltoniens du graphe suivant :

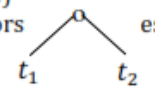


III. RECURSIVITE

III.1. Généralités

Définition
Un objet est dit récursif s'il est constitué en partie ou est défini en terme de lui-même. La récursivité est un moyen particulièrement puissant dans les définitions mathématiques. Quelques exemples bien connus sont :

- La fonction Factorielle $n!$ (définie pour les entiers non négatifs) :
  (a) $0! = 1$
  (b) Si $n > 0$, alors $n! = n \times (n-1)!$

- Les structures d'arbres :
  (a) o est un arbre (dit arbre vide)
  (b) si t_1 et t_2 sont des arbres, alors  est un arbre.

Les algorithmes récursifs sont principalement appropriés lorsque le problème à résoudre ou la fonction à calculer, ou la structure de données à traiter sont déjà définis en termes récursifs.

En général, un algorithme récursif $\mathcal{A}$ peut être exprimé comme une composition C d'instructions de base S_i (ne contenant pas $\mathcal{A}$) et de $\mathcal{A}$ lui-même.

$$\mathcal{A} \equiv \mathrm{C}[S_i, \mathcal{A}]$$

Récursivité directe et indirecte
L'outil indispensable pour la conception de programmes récursifs est la procédure. Si une procédure $\mathcal{P}$ contient une référence explicite à elle-même, on parle alors de récursivité directe. Si une procédure $\mathcal{P}$ contient une référence explicite à une autre procédure $\mathcal{T}$ qui contient une référence (directe ou indirecte) à $\mathcal{P}$, on parle dans ce cas de récursivité indirecte.

Récursivité et variables locales
Si une procédure récursive contient des variables locales, à chaque appel récursif de cette procédure, un nouvel ensemble de ces variables locales est créé ; même si elles ont le même nom, leurs adresses mémoire et leurs valeurs sont différentes.

Terminaison des algorithmes récursifs
Comme pour les instructions répétitives (Boucles), la possibilité de calculs interminables (boucle infinie) existe dans les procédures récursives, et donc il y a nécessité de considérer le problème de la terminaison des procédures récursives. C'est pour cette raison qu'une procédure récursive $\mathcal{P}$ est soumise généralement à une condition $\mathcal{B}$, qui doit être à un moment donné non satisfaite. Le schéma général des algorithmes récursifs peut donc être exprimé plus précisément comme suit :

$$\mathcal{A} = if\ (\mathcal{B})\ \mathrm{C}[S_i, \mathcal{A}]$$

Une technique classique pour démontrer qu'une procédure récursive se termine est de définir une fonction $f(x)$ (x étant l'ensemble des variables et paramètres utilisés dans la procédure), telle que la condition $f(x) \leq 0$ signifie la condition d'arrêt de la procédure, puis de démontrer que $f(x)$ décroit à chaque appel récursif de la procédure. Ceci est nécessaire d'autant plus qu'à chaque appel récursif d'une procédure, un espace mémoire additionnel est requis pour les variables locales, mais aussi pour l'état actuel des calculs qui est enregistré (dans une pile) afin d'être accessible lorsque les prochains appels récursifs se terminent, et que l'appel actuel est repris.

III.2. Quand ne pas utiliser la récursivité ?

Les algorithmes récursifs sont particulièrement appropriés lorsque le problème sous-jacent où les données à traiter sont définis en termes récursifs. Cela ne signifie pas, toutefois, qu'une définition récursive du problème et/ou des données qui conduisent à un algorithme récursif, est la meilleure façon de résoudre le problème. Un exemple classique est le calcul de la suite de Fibonacci qui est définie comme suit :

$$Fib_n = Fib_{n-1} + Fib_{n-2} \quad pour\ n \geq 0$$

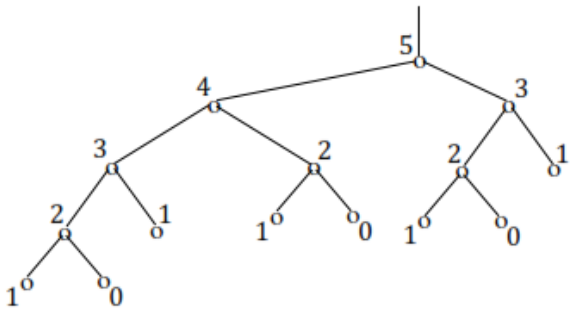
Et

$$Fib_1 = 1, \qquad Fib_0 = 0$$

Une approche récursive directe et naïve conduit à la procédure suivante :

```
public static int  Fib(int n)
{ if (n==0) return 0;
  else if (n==1) return 1;
       else return Fib(n-1)+Fib(n-2);
}
```

Le Calcul de Fib_n par un appel Fib(n) provoque une activation récursive de la fonction. Mais combien de fois ? Nous remarquons que chaque appel avec $n > 1$ mène à 2 autres appels, le nombre total d'appels augmentera ainsi de façon exponentielle. Un tel programme est clairement impraticable. La figure suivante montre qu'il faut 15 appels de la fonction récursive Fib(n) pour calculer Fib_5.



Cependant, il est clair que les nombres de Fibonacci peuvent être calculés par une méthode itérative qui permet d'éviter le recalcul des mêmes valeurs par l'utilisation de variables auxiliaires tels que $x = Fib_i$ et $y = Fib_{i-1}$.

```
i=1; x=1; y=0;
while (i<n) {
 z=x;
 i++;
 x=x+y;
 y=z;
}
```

Cet exemple ne devrait pas cependant conduire les programmeurs à se dérober de la récursivité à n'importe quel prix. Il existe plusieurs applications intéressantes à la récursivité comme vont le montrer les sections et chapitres suivants.

D'un autre coté, chaque algorithme récursif peut être transformé en un algorithme itératif. Toutefois, cela implique la manipulation explicite d'une pile, qui peut rendre plus difficile la compréhension du programme.

III.3. Règles de base pour la conception d'algorithmes récursifs

Lors de l'écriture de routines récursives, il est essentiel de garder à l'esprit les trois règles de base suivantes :

1. **Cas de base** : vous devez toujours avoir certains cas, de base, qui peuvent être résolus sans récursivité.
2. **Progression** : les appels récursifs doivent toujours progresser vers les cas de base.
3. **Ne jamais refaire le travail**, en résolvant la même instance d'un problème dans des appels récursifs séparés.

III.4. Exemples classiques d'algorithmes récursifs

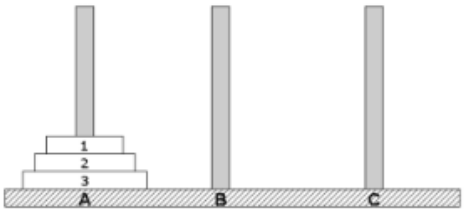
Il existe plusieurs problèmes classiques où l'utilisation de la récursivité contribue à une conception facile des algorithmes notamment lors de l'utilisation de stratégies dites « trial and error » (essais et erreurs) ou « backtracking » (retour arrière).

La tour de Hanoï

Le problème des tours de Hanoï est un jeu de réflexion qui consiste à déplacer des disques de diamètres différents d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire, et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ.



Le problème des tours de Hanoï est vu en algorithmique, où il offre un exemple de la puissance et de la lisibilité des programmes définis de façon récursive.

Les paramètres de la procédure *Hanoï* sont :

- *n* : nombre de disques utilisés,
- A : emplacement de départ,
- C : emplacement d'arrivée,
- B : emplacement intermédiaire.

```
Procedure Hanoï(n, A, C, B)
{ si n≠0 alors
    Hanoï(n-1,A,B,C);
    Déplacer le disque de A vers C
    Hanoï(n-1,B,C,A);
  finsi
}
```

IV. COMPLEXITE DES ALGORITHMES

On abordera essentiellement dans cette section la complexité temporelle ou en temps.

IV.1. Généralités

a. Définition

L'étude de la complexité des algorithmes a pour objectif l'estimation du coût d'un algorithme (assorti d'une structure de donnée). Cette mesure permet donc la comparaison de deux algorithmes sans avoir à les programmer.

Si l'on prend en compte pour l'estimation de la complexité les ressources de la machine telles que la fréquence d'horloge, le nombre de processeurs, le temps d'accès disque etc., on se rend compte immédiatement de la complication voir l'impossibilité d'une telle tâche. Pour cela, on se contente souvent d'estimer la **relation entre la taille des données et le temps d'exécution**, et ceci **indépendamment de l'architecture utilisée**.

Il s'agit d'un modèle simplifié qui tient compte des ressources technologiques ainsi que leurs coûts associés. On prendra comme référence un modèle de machine à accès aléatoire et à processeur unique où les opérations sont exécutées l'une après l'autre sans opérations simultanées.

On définit la complexité (temporelle) d'un algorithme comme étant la mesure du nombre d'opérations élémentaires qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données.

L'analyse de la complexité consiste généralement à mesurer **asymptotiquement** le temps requis pour l'exécution de l'algorithme sur une machine selon un modèle de calcul. Cette mesure permet juste d'avoir une idée imprécise mais **très utile** sur le temps d'exécution de l'algorithme en question. Elle nous permet, entre autres, d'estimer la **taille des instances** pouvant être traitées sur une machine donnée en un temps raisonnable. Cependant, l'analyse de la complexité peut s'avérer très difficile même pour des algorithmes relativement simples nécessitant parfois des outils mathématiques très poussés.

D'une façon formelle, on peut aussi définir la complexité d'un algorithme $\mathcal{A}$ comme étant tout **ordre de grandeur** du nombre d'opérations élémentaires effectuées pendant le déroulement de $\mathcal{A}$. Les notions d'opérations élémentaires et d'ordre de grandeurs ainsi que les notations les plus utilisées seront définis au fur et à mesure.

b. Opérations élémentaires

On appelle **opérations élémentaires** les opérations suivantes :

- un accès mémoire pour lire ou écrire la valeur d'une variable ou d'une case d'un tableau ;
- une opération arithmétique entre entiers ou réels telle que l'addition, soustraction, multiplication, division ou calcul du reste d'une division entière ;
- une comparaison entre deux entiers ou réels.

Exemple : L'instruction « c ←a + b ; » nécessite les quatre opérations élémentaires suivantes :

1- un accès mémoire pour la lecture de la valeur de a,
2- un accès mémoire pour la lecture de la valeur de b,
3- une addition de a et b,
4- un accès mémoire pour l'écriture de la nouvelle valeur de c.

c. Remarques

- Il est recommandé de repérer les opérations fondamentales d'un algorithme. Leur nombre intervient principalement dans l'étude de la complexité. Voici quelques exemples :

Algorithme	Opérations fondamentales
Recherche d'un élément	Comparaisons
Tri	Comparaisons et déplacements (permutations)
Multiplication de matrices	Addition et multiplications

- Faire attention aux boucles et à la récursivité ; la complexité finale d'un algorithme provient généralement de ces deux types d'instructions.

IV.2. Evaluation des coûts

Suivant le type d'instruction, la complexité est évaluée comme suit :

a.Séquence

$\mathcal{A}$: | $\mathcal{J}$;
| $\mathcal{K}$;

$\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{J}}(N) + T_{\mathcal{K}}(N)$

b. Alternative

$\mathcal{A}$: Si $\mathcal{C}$ Alors $\mathcal{J}$ Sinon $\mathcal{K}$

$\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{C}}(N) + max\{T_{\mathcal{J}}(N), T_{\mathcal{K}}(N)\}$: Pire cas

$\Rightarrow T_{\mathcal{A}}(N) = T_{\mathcal{C}}(N) + min\{T_{\mathcal{J}}(N), T_{\mathcal{K}}(N)\}$: Meilleur cas

c. Boucles

$\mathcal{A}$: nb ↻ $\mathcal{B}$

$\Rightarrow T_{\mathcal{A}}(N) = nb \times T_{\mathcal{B}}(N)$

d. Récursivité

Etablir la formule de récurrence et en déduire la complexité.

IV.3. Mise en œuvre

a. Exemple d'illustration : Le Tri par insertion

Considérons l'algorithme de tri par insertion permettant de trier un tableau A de taille N. Considérons le modèle de complexité qui associe à chaque ligne i (du code) son coût (ou temps d'exécution) C_i. L'analyse peut se faire donc comme suit :

```
public static void triInsertion (int[]A,int N) // coût      Nombre de fois
    {                                          // ==========================
    int j, temp, i;
    for (j=2 ; j<=N ; j++)                     // coût C1 ........ N
       {
         temp = A[j];                          // coût C2 ........ N-1
         i = j-1;                              // coût C3 ........ N-1
         while (i>0 && A[i]>temp)              // coût C4 ........ 2+3+...+N
           {
             A[i+1]=A[i];                      // coût C5 ........ 1+2+...N-1
             i=i-1;                            // coût C6 ........ 1+2+...N-1
           }
         A[i+1]=temp;                          // coût C7 ........ N-1
      }
    }
```

b. Rappel des principales suites

- **Suite arithmétique :**

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

- **Suite géométrique :**

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + \cdots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

- **Suite harmonique :**

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = Ln(n)$$

La complexité de l'algorithme de tri par insertion est donc :

$$T(N) = (\frac{C_4 + C_5 + C_6}{2})N^2 + (C_1 + C_2 + C_3 + \frac{C_4 - C_5 - C_6}{2} + C_7)N - (C_2 + C_3 + C_4 + C_7)$$

Et puisqu'on s'intéresse aux valeurs de N très grandes (comportement asymptotique de la complexité), nous pouvons noter la complexité en utilisant les ordres de grandeurs (voir rappel ci-après) comme suit :

$$\mathbf{T(N) = O(N^2)}$$

Note :
T(N) représente la complexité dans le **pire des cas** car on n'a pas tenu compte dans notre calcul de la valeur de l'expression logique de la condition (A[i] > temp) qui détermine le nombre d'exécutions de la boucle interne. L'algorithme de tri par insertion peut donc prendre moins de temps pour des tableaux ayant une structure particulière et par conséquent, la complexité est une variable aléatoire. C'est pourquoi on parle de **complexité en moyenne** d'un algorithme qu'on peut calculer comme étant l'espérance de cette variable aléatoire.

c. Généralités sur les ordres de grandeurs

- Notation Θ (thêta) : soit g une fonction positive d'une variable entière n. **Θ(g(n))** désigne l'ensemble des fonctions positives de la variable n, pour lesquelles il existe deux constantes c_1, c_2 et un entier n_0, satisfaisant la relation :

$$\mathbf{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \qquad \forall n \geq n_0}$$

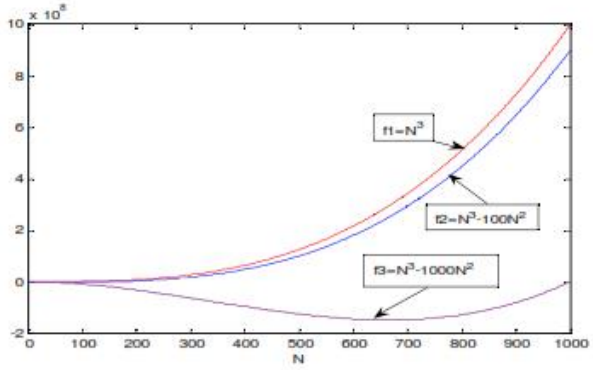
Par abus de notation on écrit f(n)=Θ(g(n)) pour exprimer que f ∈ Θ(g(n)) (malgré que Θ(g(n)) est un ensemble et f est un élément de celui-ci).

Exemple :
Soit $g(n) = n^2$ et $f(n) = 50n^2+10n$.
Il s'agit de trouver c_1,c_2 et n_0 tels que : $c_1 n^2 \leq 50n^2+10n \Rightarrow c_1 \leq 50$,
$50n^2+10n \leq c_2 n^2 \Rightarrow c_2 \geq 50$.

donc on a bien $f(n) = \Theta(g(n)) = \Theta(n^2)$

La figure suivante montre les graphes de trois fonctions f1, f2 et f3. On peut facilement vérifier que f2 = Θ(f1(N)) et aussi f3 = Θ(f1(N)).

On peut interpréter la relation f(n)=Θ(g(n)) comme suit : pour n assez grand, la fonction f est **bornée à la fois supérieurement et inférieurement** par la fonction g, c'est-à-dire que les fonctions f et g sont égales, à une constante prét.



- Notation O (grand o) : Lorsque la fonction f est bornée **uniquement supérieurement** par la fonction g on utilise la notation **O**. **O(g(n))** désigne donc l'ensemble des fonctions positives de la variable n, pour lesquelles il existe une constante c et un entier n_0, satisfaisant la relation :

$$\mathbf{0 \leq f(n) \leq c\,g(n) \qquad \forall\, n \geq n_0}$$

La relation f(n) = O(g(n)) indique que la fonction f est bornée supérieurement par la fonction g pour des valeurs suffisamment grandes de l'argument n. Donc f(n) = Θ(g(n)) implique que f(n) = O(g(n)) (par abus de notation). Formellement, on peut écrire Θ(g(n)) ⊆ O(g(n)).

Exemple : $100n^2 + 10n = \mathbf{O}(n^2)$ mais on peut aussi écrire **$100n = O(n^2)$!** Car ceci est équivalent à dire que 100n est asymptotiquement bornée supérieurement par n^2.

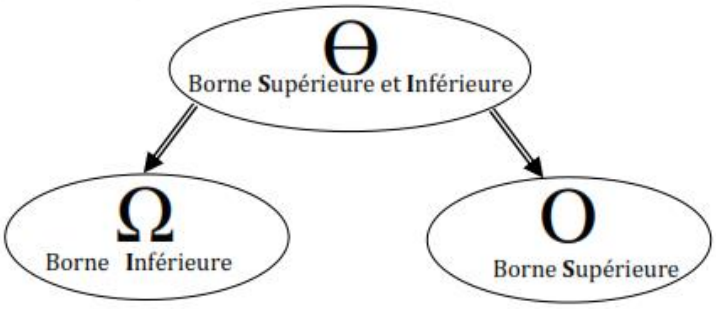
- Notation Ω (grand oméga): Ω(g(n)) désigne l'ensemble des fonctions positives de la variable n, pour lesquelles il existe une constante c et un entier n_0, satisfaisant la relation :

$$\mathbf{0 \leq cg(n) \leq f(n) \quad \forall\, n \geq n_0}$$

La relation f(n) = Ω(g(n)) indique que la fonction f est bornée inférieurement par la fonction g pour des valeurs suffisamment grandes de l'argument n. Par conséquent, **f(n) = Θ(g(n)) implique que f(n) = Ω(g(n))** (par abus de notation). Formellement, on peut écrire : **Θ(g(n)) ⊆ Ω(g(n))**. Le théorème suivant découle immédiatement des définitions données :

Théorème : f(n) = **Θ**(g(n)) si et seulement si : f(n) =**Ω**(g(n)) et f(n) = **O**(g(n))

Les trois notations précédentes peuvent être récapitulées par le schéma suivant :



Quelques propriétés de la notation O

1. Les facteurs constants peuvent être ignorés (Ex : $3x^2 = O(x^2)$)
2. Une grande puissance de n croît plus vite qu'une puissance inférieure ($n^r = O(n^s)$ si $0 \leq r \leq s$)
3. Le taux de croissance d'une somme de termes est le taux de croissance du terme le plus rapide en croissance. (Ex : $an^3 + bn^2 = O(n^3)$)
4. Les fonctions exponenielles sont plus rapides en croissance que les puissances (polynômes)
5. Tous les logarithmes ont un même taux de croissance.

d. Autre méthode d'analyse de complexité

La complexité d'un algorithme peut être dans certains cas facilement trouvée si on arrive à analyser et comprendre l'algorithme et repérer ses opérations fondamentales. Ainsi pour le tri par insertion, on peut s'intéresser aux comparaisons comme suit :

Le nombre de comparaisons exécutées à l'itération i est au plus i. Le nombre total de comparaisons est :

$$\sum_{i=2}^{N} i = \frac{N(N+1)}{2} - 1 = O(N^2)$$

e. Exercices

1. Analyser la complexité de la recherche séquentielle puis de la recherche dichotomique.
2. Analyser la complexité de la fonction récursive de calcul de $n!$.
3. Développer la formule de récurrence suivante :

$$T(n) = T(n/2) + an + b \text{ et } T(1) = 1$$

IV.4. Différentes formes de complexité

Il est évident que la complexité d'un algorithme peut ne pas être la même pour deux jeux de données différents. Ceci peut avoir des conséquences sur le choix du meilleur algorithme pour un problème donné. En pratique, on s'intéresse aux formes suivantes de la complexité :

a. Complexité dans le pire des cas (Worst case)

Il s'agit de considérer le plus grand nombre d'opérations élémentaires effectuées sur l'**ensemble de toutes les instances** du problème ; on cherche une borne supérieure qui est atteinte même pour des instances ayant une très faible, voire zéro, probabilité.

Cette forme de complexité est plus simple à calculer mais peut conduire à un choix erroné d'un algorithme. En effet le pire cas peut être un cas très rare !

b. Complexité dans le meilleur des cas (Best case)

Il s'agit, cette fois-ci, de considérer le plus petit nombre d'opérations élémentaires ; c'est-à-dire on cherche un minorant de la complexité au lieu d'un majorant.

Cette forme de complexité peut être considérée comme complément de la complexité dans le pire des cas mais n'offre aucune garantie à l'utilisateur.

c. Complexité en moyenne (Average case)

Il s'agit de calculer la moyenne (espérance mathématique) des nombres d'opérations élémentaires effectuées sur la totalité des instances. Ce calcul est généralement très difficile et souvent même délicat à mettre en œuvre car il faut connaître la probabilité de chacun des jeux de données pour pouvoir calculer la complexité en moyenne.

Cette forme d'analyse fait actuellement l'objet de nombreux travaux de recherche et permet d'expliquer, d'une part le comportement de certains algorithmes en pratique ; et d'autre part, le choix d'algorithmes pour des problèmes ayant des tailles considérables tels que les problèmes d'apprentissage en intelligence artificielle.

IV.5. Algorithmes de tri & Complexité

Le tri est sans doute un des problèmes les plus fondamentaux de l'algorithmique. Après le tri beaucoup d'autres problèmes deviennent facile à résoudre tels que l'unicité ou la recherche.

Le tri consiste à réarranger une liste de n objets de telle manière :

$X_1 \leq X_2 \leq \cdots \leq X_n$: Tri par ordre croissant

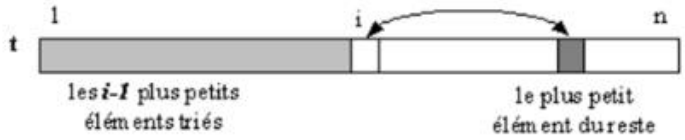
$X_1 \geq X_2 \geq \cdots \geq X_n$: Tri par ordre décroissant

Il existe plusieurs méthodes de tri, nous citons :

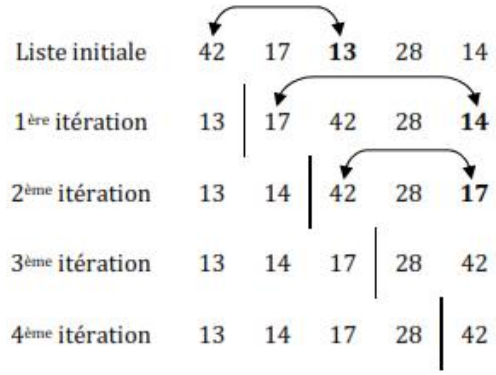
Tri par Selection (Selection Sort)

Principe

Itérativement, le tri par sélection consiste à chercher le plus petit élément puis de le mettre au début.



Exemple



Implémentation

```
public void selectionSort(int[] A, int n)
{
for (int i=0;i<n-1;i++)
 { int imin=i;
   for (int j=i+1;j<n;j++)
       if (A[j]<A[imin]) imin=j;
   swap(A,i,imin)
 }
}
```

Complexité

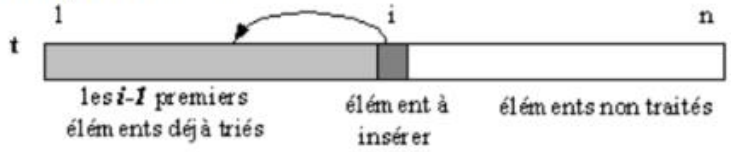
Le pire et le meilleur cas sont pareils, puisque pour trouver le plus petit élément, $(n-1)$ itérations sont nécessaires, pour le 2ème plus petit élément, $(n-2)$ itérations sont effectuées... jusqu'à l'avant dernier plus petit élément qui nécessite 1 itération. Le nombre total d'itérations est donc :

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

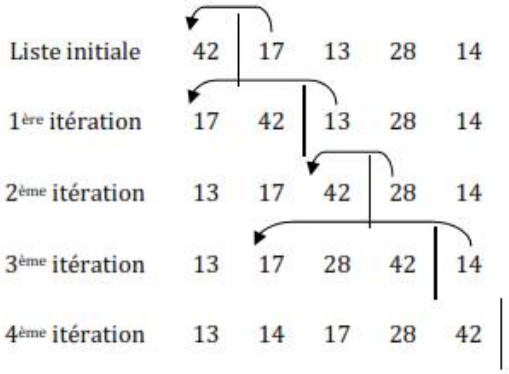
Tri par Insertion (Insertion Sort)

Principe

Itérativement, on insère le prochain élément dans la partie qui a été déjà triée précédemment. La partie de départ qui est triée est le premier élément.



Exemple



Implémentation

```
public void insertionSort(int[] A, int n)
{
 for (int i=1;i<n;i++)
 { int temp=A[i],j=i;
   while (j>0 && A[j-1]>temp)
   { A[j]=A[j-1];j--;}
   A[j]=temp;
 }
}
```

Complexité

Comme nous n'avons pas nécessairement à scanner toute la partie déjà triée, le pire et le meilleur cas sont différents.

Meilleur cas : si le tableau est déjà trié, chaque élément est toujours inséré à la fin de la partie triée ; nous n'avons à déplacer aucun élément. Comme nous avons à insérer $(n-1)$ éléments, chacun générant seulement une comparaison, la complexité est $O(n)$.

Pire cas : si le tableau est inversement trié, chaque élément est inséré au début de la partie triée. Dans ce cas, tous les éléments de la partie triée doivent être déplacés à chaque itération. La ième itération génère (i-1) comparaisons et échanges de valeurs :

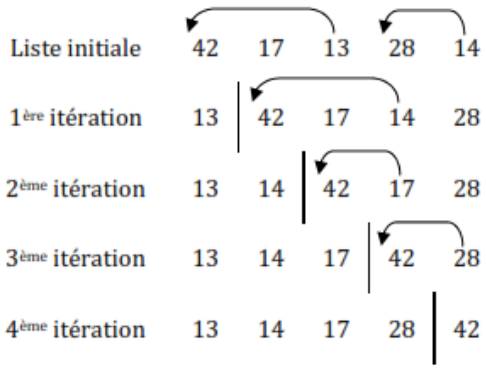
$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2} = O(n^2)$$

Tri à Bulles (Bubble Sort)

Principe

Parcourir le tableau en comparant deux à deux les éléments successifs et permuter s'ils ne sont pas dans l'ordre.

Exemple



Implémentation

```
public void bubbleSort(int[] A, int n)
{
for (int i=0;i<n-1;i++)
  for (int j=n-1;j>i;j--)
    if (A[j]<A[j-1]) swap(A,j,j-1);
}
```

Complexité

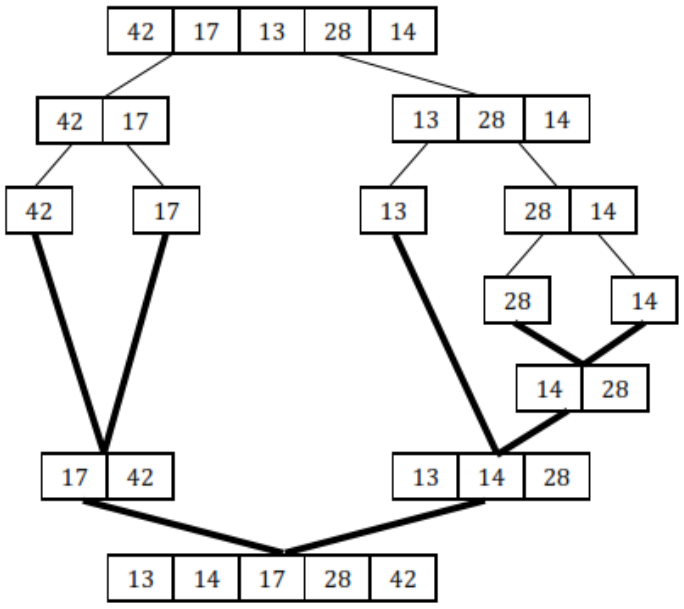
Globalement, c'est la même complexité que le tri par sélection. Le meilleur et le pire cas sont pareils avec une complexité de $O(n^2)$.

Tri par fusion (Merge Sort)

Principe

Cet algorithme divise en deux parties égales le tableau. Après que ces deux parties soient triées (de manière généralement récursive), elles sont fusionnées pour l'ensemble des données.

Exemple



Implémentation

```
public void mergeSort(int[] A, int deb , int fin)
{
 int mil;
 if (deb<fin) {
   mil=(deb+fin)/2;          (* DIVIDE *) ..... O(1)
   mergeSort(A,deb,mil);     (* CONQUER *) ..... T(n/2)
   mergeSort(A,mil+1,fin);   (* CONQUER *) ..... T(n/2)
   fusion(A,deb,mil,fin);    (* COMBINE *) ..... O(n)
  }
}

public void fusion(int[] A, int deb, int mil, int fin)
{
  int n1,n2,i,j; int[] R = new int[50]; int[] L = new int[50];
  n1=mil-deb+1;
  n2=fin-mil;
  for (i=1;i<=n1;i++) L[i]=A[deb+i-1];
  for (j=1;j<=n2;j++) R[j]=A[mil+j];
  L[n1+1]=9999;  //Nombre très grand
  R[n2+1]=9999;
  i=j=1;
  for (intk=deb;k<=fin;k++) {
  if (L[i]<=R[j]) {A[k]=L[i];i++;}
  else {A[k]=R[j];j++;}
 }
}
```

Complexité

La complexité peut être exprimée par récurrence :

$$T(n) = \begin{cases} \text{O(1)} & \text{si n} = 1 \\ 2\text{T(n/2)} + \text{O(n)} & \text{si n} > 1 \end{cases} \Rightarrow T(n) = O(n log_2 n)$$

Le tableau A sera divisé par 2 jusqu'à obtention de tableaux de taille 1, ainsi :
Taille de A : n , $/2$, $n/4$, ..., $n/2^p$ avec $n/2^p = 1 \Rightarrow p = log_2(n)$
Puisqu'à chaque étape, une (ou plusieurs) opération(s) de fusion de l'ordre $O(n)$ est exécutée sur les sous tableaux obtenus, on obtient donc $O(n log_2 n)$.

Tri rapide (Quick Sort)

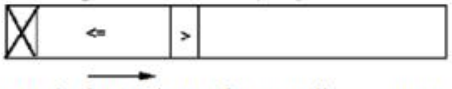
Principe

L'idée de cet algorithme est de diviser le tableau en deux parties séparées par un élément appelé pivot de telle manière que les éléments de la partie gauche soient tous inférieurs ou égaux à cet élément et ceux de la partie droite soient tous supérieurs à ce pivot. Cette étape fondamentale du tri rapide s'appelle le partitionnement.

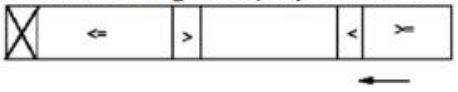
Choix du pivot : Le choix idéal serait que ça coupe le tableau exactement en deux parties égales, mais cela n'est pas toujours possible. On peut prendre le premier ou le dernier ou de manière aléatoire,...

Partitionnement :

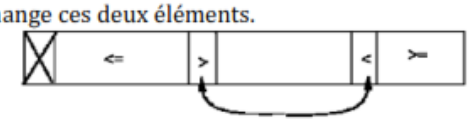
- On parcourt de **gauche à droite** jusqu'à rencontrer un élément **supérieur** au pivot.



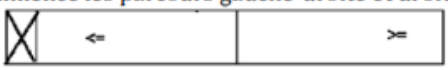
- On parcourt de **droite à gauche** jusqu'à rencontrer un élément **inférieur** au pivot.



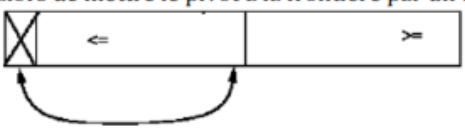
- On échange ces deux éléments.



- On recommence les parcours gauche-droite et droite-gauche jusqu'à avoir :

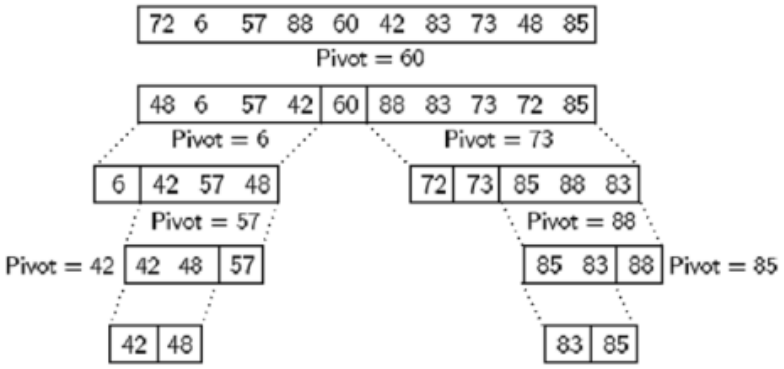


- Il suffit alors de mettre le pivot à la frontière par un échange



Dans ce qui suit (Exemple & implémentation) on choisit l'élément se trouvant au milieu du tableau comme pivot.

Exemple



Implémentation

```
public void quickSort(int[] A, int deb, int fin)
{
 int ipivot;
 if (deb<fin)
  {
   ipivot=partition(A,deb,fin);
   quickSort(A,deb,ipivot);
   quickSort(A,ipivot+1,fin);
  }
}

public int partition(int[] A, int deb, int fin)
{
 int pivot,aux,i,j;
 pivot=A[(deb+fin)/2];
 i=deb;j=fin;
 while (i<j)
  {
   while (pivot>A[i]) i++;
   while (pivot<A[j]) --j;
   if (i<j) swap(A,i,j);
  }
 return j;
}
```

Complexité

- **Cas favorable :**

La meilleure chose qui puisse arriver, c'est qu'à chaque fois que la fonction Partition()est appelée, elle divise exactement le tableau (ou le sous-tableau) en 2 parties égales.

À la première passe, les n éléments du tableau sont comparés avec la valeur pivot pour les balancer à la droite ou à la gauche. Il y a donc n comparaisons. À la seconde passe il y a 2 fonctions Partition() qui effectuent leur rôle chacune sur leur moitié de tableau. Chaque fonction doit comparer les $n/2$ éléments du sous-tableau pour effectuer le balancement. Donc de fait, il y a encore n comparaisons pour cette passe. Il en sera de même pour les autres itérations.

Nous pouvons exprimer la complexité sous forme de récurrence :

$$T(n) = \begin{cases} O(1)\ si\ n = 1 \\ 2T\left(\frac{n}{2}\right) + O(n)\ si\ n > 1 \end{cases} \Rightarrow T(n) = O(nlog_2 n)$$

- **Cas défavorable :**

La pire chose qui puisse arriver, c'est qu'à chaque appel à la fonction Partition(), à cause des circonstances de la disposition des données, celle-ci place la totalité du sous-tableau à droite ou à gauche (excluant bien sûr l'élément pivot qui est alors à sa place définitive). Dès lors le tri rapide se transforme en un tri à bulles.

À la première passe, il y aura donc n comparaisons. À la seconde passe il y a déjà une valeur ordonnée et un sous-tableau de $n-1$ éléments, il y aura donc $n-1$ comparaisons effectuées par la fonction Partition(). À la 3ème passe, il y aura $n-2$ comparaisons, etc.

Pour effectuer le tri au complet, il aura donc fallu en tout n passes. D'où la complexité:

$$n + (n-1) + \ldots + 2 + 1 = n(n+1)/2$$

Soit donc $O(n^2)$ = pire cas de Quicksort. Mais il reste que la complexité en moyenne est $O(n\log_2 n)$.

IV.6. Complexité polynomiale et complexité exponentielle

Un algorithme est considéré pratique, relativement à une classe de problèmes, s'il est capable de résoudre n'importe qu'elle instance (dans le pire des cas) ou en moyenne (complexité moyenne) en temps polynomial c'est-à-dire, sa complexité est $O(N^k)$, où k est un entier quelconque.

En réalité une complexité est dite polynomiale si elle peut être bornée par un polynome, on retrouve donc :

$O(1)$	(Complexité constante)
$O(log(N))$	(Complexité logarithmique)
$O(\sqrt{N})$	(Complexité racinaire)
$O(N)$	(Complexité linéaire)
$O(NlogN)$	(Complexité quasi-linéaire)
$O(N^2)$	(Complexité quadratique)
$O(N^3)$	(Complexité cubique)
…	

D'autre part une complexité est dite exponentielle ($O(k^N)$ en général) si elle ne peut pas être bornée par un polynome, on retrouve par exemple :

$O(N^{logN})$	(Complexité sous exponentielle)
$O(2^N), O(3^N)$	(Complexité exponentielle)
$O(N!)$	(Complexité factorielle)
$O(2^{2^N})$	(Complexité double exponentielle)