Lycée d'enseignement général et technologique international Victor Hugo
COLOMIERS

NSI
NUMÉRIQUE ET SCIENCES INFORMATIQUES

Algorithmique I

Cours & Activité Pratique

Nom : Prénom : Date :

Table des matières

1 Algorithmes sur les arbres binaires

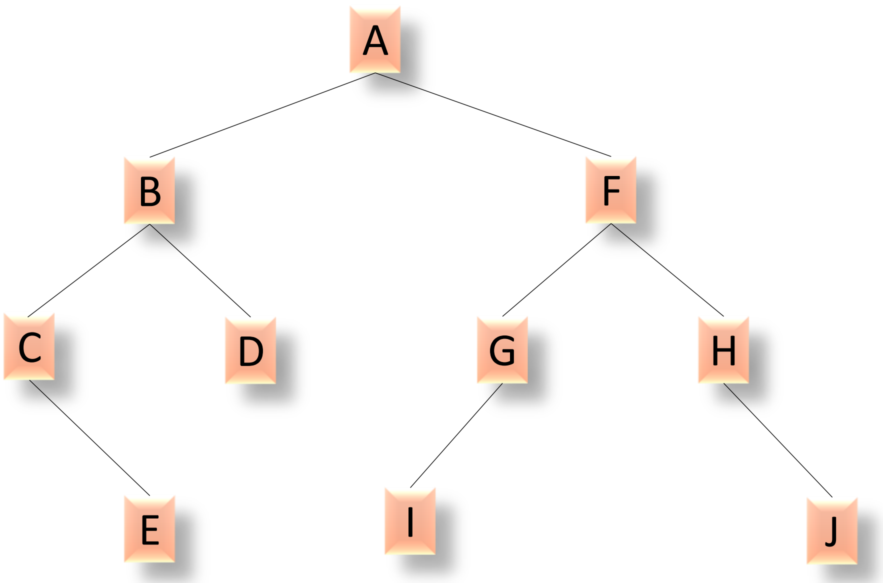
Prérequis : afin de profiter pleinement de cette activité, il est vivement conseillé de revoir les caractéristiques des arbres (cf. chapitres 3.1 et 3.2 du Support d'activité intitulé « SDD_2020.pdf »)

1.1 Rappels

Revenons sur le concept d'arbre binaire :

À chaque nœud d'un arbre binaire, on associe une clé ("valeur" associée au nœud), un "sous arbre gauche" et un "sous arbre droit"

Soit l'arbre binaire « Oak 2 » suivant :



Arbre Oak2

Si on prend le nœud ayant pour clé A (le nœud racine de l'arbre) on a :

- le sous arbre gauche est composé du nœud ayant pour clé B, du nœud ayant pour clé C, du nœud ayant pour clé D et du nœud ayant pour clé E
- le sous arbre droit est composé du nœud ayant pour clé F, du nœud ayant pour clé G, du nœud ayant pour clé H, du nœud ayant pour clé I et du nœud ayant pour clé J

Si on prend le nœud ayant pour clé B on a :

Le sous arbre gauche est composé du noeud ayant pour cle C, du noeud ayant pour cle E
Le sous arbre droite est composé du noeud ayant pour cle D

Un arbre (ou un sous arbre) vide est noté NIL (NIL est une abréviation du latin nihil qui veut dire "rien")

Si on prend le nœud ayant pour clé G on a :

```
Le sous arbre gauche est compose du noeud ayant pour cle I
Le sous arbre droite NIL
```

Il faut comprendre qu'un sous arbre (droite ou gauche) est un arbre (même s'il contient un seul nœud ou pas de nœud du tout (NIL)).

Soit un arbre Ar :

Ar.racine correspond au nœud racine de l'arbre T

Soit un nœud n :

- n.gauche correspond au sous arbre gauche du nœud n
- n.droit correspond au sous arbre droit du nœud n
- n.clé correspond à la clé du nœud n

Enfin, notons que si le nœud n est une feuille, n.gauche et n.droit sont des arbres vides (NIL).

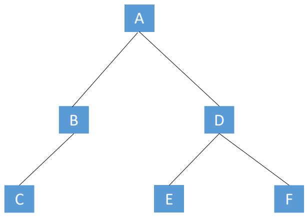
1.2 Calcul de la hauteur d'un arbre

Soit l'algorithme « hauteurArbreBinaire » suivant :

```
Variables
Ar : Arbre
n : Nœud
DEBUT
HAUTEUR(Ar) :
Si Ar <> Nil :
    n ← Ar.racine
    renvoyer 1 + max(HAUTEUR(n.gauche), HAUTEUR(n.droit))
    # la fonction « max » renvoie la plus grande valeur des 2 valeurs passées en paramètre
Sinon
    Renvoyer 0
FIN
```

A faire vous-même I

Faites tourner manuellement cet algorithme en l'appliquant à l'arbre binaire I « Arb1 » suivant :



Quelle est la hauteur de cet arbre ?

<u>Aide :</u> vous pouvez présenter le déroulement de l'algorithme de la façon suivante :

`Appel : Hauteur(Arb1)`

`n = A, n.gauche = B, n.droit = D`

Renvoyer 1 + max(Hauteur(B), Hauteur(D))

`Appel : Hauteur(B)`

n = …, `n.gauche` = …, `n.droit` = …

Renvoyer …

Appel : …
…
…

Etc…

La hauteur est 3

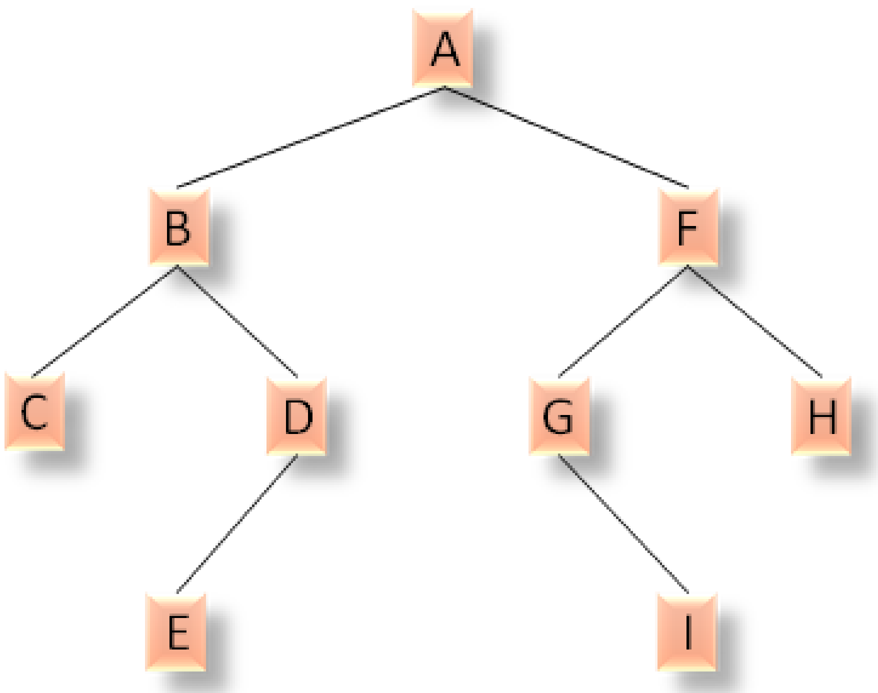
1.3 Calcul de la taille d'un arbre

La taille d'un arbre binaire est le nombre de nœuds qu'il contient.
Soit l'algorithme « tailleArbreBinaire » suivant :

```
Variables
Ar : Arbre
n : Nœud
DEBUT
TAILLE (Ar) :
Si Ar <> Nil :
        n ← Ar.racine
renvoyer 1 + TAILLE(n.gauche) + TAILLE(n.droit))
Sinon
        Renvoyer 0
FIN
```

A faire vous-même II

Faites tourner manuellement cet algorithme en l'appliquant à l'arbre « Oak1 » suivant :



Arbre Oak1

Quelle est la taille de « Oak1 » ?

```
Taille(Oak1) = 9
```

1.4 Parcours d’un arbre binaire

En fonction du problème traité, il existe plusieurs façons de parcourir un arbre. Nous pouvons parcourir l’arbre dans l’ordre :

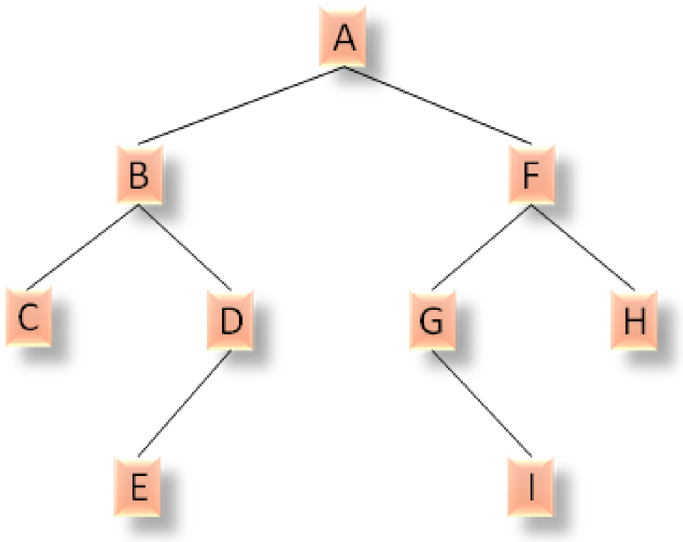
- Infixe
- Préfixe
- Suffixe

Soit l’algorithme « parcoursInfixe » de parcours d’un arbre dans l’ordre infixe :

```
VARIABLES
Ar : arbre
n : nœud

DEBUT
ordreInfixe(Ar) :
   si Ar <> NIL :
        n ← Ar.racine
        ordreInfixe (n.gauche)
        # affiche la valeur du nœud
        affiche n.clé
        ordreInfixe (n.droit)
   fin si
FIN
```

A faire vous-même IV

Faites tourner manuellement cet algorithme en l'appliquant à l'arbre « Oak1 » suivant :



Arbre Oak1

<u>Aide</u> : vous pouvez présenter le déroulement de l'algorithme de la façon suivante :

```
Appel : ordreInfixe(Oak1)
Ar = Oak1, n = A, n.gauche = B, n.droit = F


Appel : ordreInfixe(n.gauche) → ordreInfixe(B)
Ar = …, n = …, n.gauche …, n.droit …


Appel : ordreInfixe(…) → ordreInfixe(…)
…
affiche(n.clé) → affiche(« C »)
```

C

Appel : …
…

Etc…

Quel est l'ordre de parcours de l'arbre ?

affichage (CBEDAGIFH)

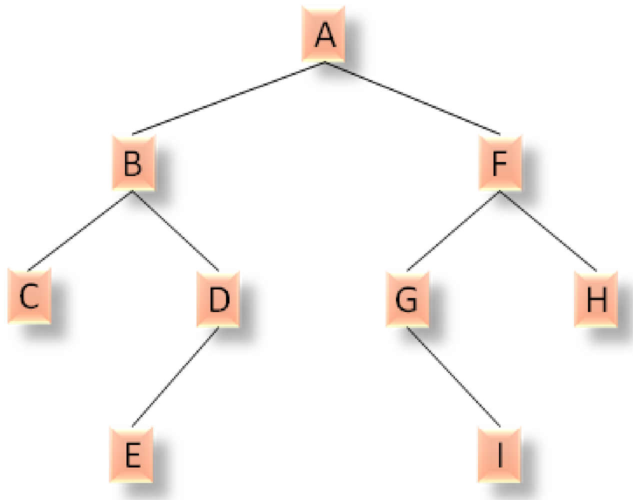
1.5 Parcours d'un arbre binaire en largeur d'abord

Soit l'algorithme « parcoursLargeurDab » de parcours d'un arbre binaire dans l'ordre largeur d'abord :

```
VARIABLES
Ar : arbre
ArG : arbre
ArD : arbre
n : nœud
f : file

DEBUT
f ← file()
parcoursLargeur(Ar) :
   enfiler(Ar.racine, f) # on place la racine de l'arbre dans la file f
   tant que f non vide :
       n ← defiler(f)
       affiche n.clé
       si n.gauche ≠ NIL :
           ArG ← n.gauche
           enfiler(ArG.racine, f)
       fin si
       si n.droit ≠ NIL :
           ArD ← n.droite
           enfiler(ArD.racine, f)
       fin si
   fin tant que
FIN
```

A faire vous-même V

Faites tourner manuellement cet algorithme en l'appliquant à l'arbre « Oak1 » suivant :



Arbre Oak1

Aide : vous pouvez présenter le déroulement de l'algorithme en complétant le tableau suivant :

Variable / étape	file	n	n.clé	n.gauche	ArG	n.droit	ArD
1	A						
2a	vide	A		B		F	
2b	vide	A	A	B		F	
2c	…	…	…	…	B(C(None, None),D(E,None))	…	
…	…	…	…	…	…	…	…
10b	…	…	…	…	…	…	…

Quel est l'ordre de parcours de l'arbre ? Selon vous, pourquoi parle-t-on de parcours en largeur ?

```
L'affichage sera "ABFCDGHEI"
L'algorithme affiche la racine puis les fils de cette racine puis les fils
de ces racines etc...
Cest un parcours en largeur car chaque ligne en largeur est afficher
```

2 Arbre binaire de recherche

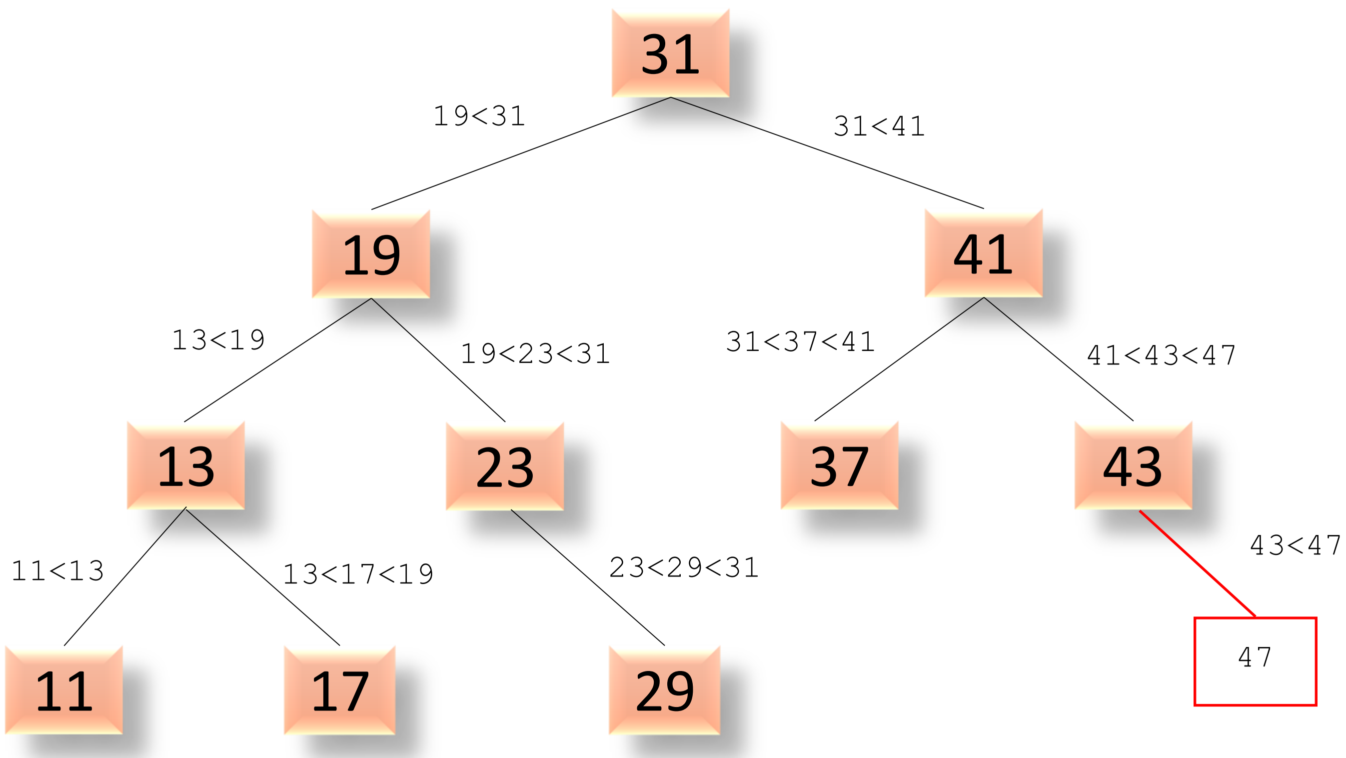
2.1 Qu'est-ce que c'est ?

Un arbre binaire de recherche est un cas particulier d'arbre binaire. Dans un arbre binaire de recherche :

- les clés de nœuds composant l'arbre sont ordonnables. Cela signifie qu'on doit pouvoir classer les nœuds, par exemple, de la plus petite clé à la plus grande
- soit n un nœud de l'arbre binaire de recherche. Si m est un nœud du sous arbre gauche de x, alors il faut que m.clé ⩽ n.clé. Si m est un nœud du sous arbre droit de n, il faut alors que n.clé ⩽ m.clé.

2.2 Parcours d'un arbre binaire de recherche

Soit l'arbre binaire de recherche suivant :



Arbre Oak4

A faire vous-même VI

A partir de la définition énoncée au paragraphe 2.1, vérifiez que l'arbre « Oak4 » est un arbre binaire de recherche.

A faire vous-même VII

Faites tourner manuellement l'algorithme de parcours infixe en l'appliquant à l'arbre « Oak4 ». Quel est l'ordre de parcours affiché ? Que remarquez-vous ?

```
11,13,17,19,23,29,31,37,41,43
C'est trier par ordre croissant
```

2.3 Insertion d'un nœud dans un arbre binaire de recherche

Soit l'algorithme « insertCléAbr » d'insertion d'un nœud m dans un arbre binaire « Ar » :

```
VARIABLES
Ar : arbre
n : nœud
m : nœud
DEBUT
insertClé(Ar, m) :
    n ← Ar.racine
    tant que Ar ≠ NIL :
        n ← Ar.racine
        si m.clé < n.clé :
            Ar ← n.gauche
        sinon :
            Ar ← n.droit
        fin si
    fin tant que
    si m.clé < n.clé :
            insérer m à gauche de n
    sinon :
            insérer m à droite de n
    fin si
FIN
```

A faire vous-même VIII

1. Faites tourner manuellement cet algorithme en l'appliquant à l'arbre « Oak4 » suivant.

2. Dessinez « Oak5 », l'arbre obtenu après l'insertion du nœud m. Vérifiez que « Oak5 » est bien un arbre binaire de recherche.

3 Algorithmes sur les graphes

3.1 Les parcours d'un graphe

"Parcourir" un graphe, c'est "visiter" tous les sommets du graphe en partant d'un sommet quelconque. Les algorithmes de parcours d'un graphe sont à la base de nombreux algorithmes très utilisés : routage des paquets de données dans un réseau, découverte du chemin le plus court pour aller d'une ville à une autre...

Il existe 2 méthodes pour parcourir un graphe :

- Le parcours en largeur d'abord
- Le parcours en profondeur d'abord

Le parcours en largeur d'abord

Soit un graphe G(V,E) avec V l'ensemble des sommets de ce graphe et E l'ensemble des arêtes de ce graphe. Un sommet u sera adjacent avec un sommet v si u et v sont reliés par une arête (on pourra aussi dire que u et v sont voisins). À chaque sommet u de ce graphe nous allons associer une couleur : blanc ou noir. Autrement dit, chaque sommet u possède un attribut couleur que l'on notera u.couleur. Nous aurons donc « u.couleur = blanc » ou « u.couleur = noir ». Quelle est la signification de ces couleurs ?

- si u.couleur = blanc => u n'a pas encore été "découvert"
- si u.couleur = noir => u a été "découvert"

Soit l'algorithme « largeurGraphe_algo » de parcours en largeur d'abord d'un graphe « Gr » :

```
# nom : largeurGraphe_algo.py
# rôle : Parcours un graphe en largeur d'abord
# propriété : appelle les fonctions enfiler() et defiler()

# Début algorithme

# Variables et constantes
# Gr : un graphe
# s : noeud (origine)
# u : noeud
# v : noeud
# f : file()


# On part du principe que pour tout sommet u du graphe "Gr",
# u.couleur <-- 'blanc' à l'origine

# s.couleur ← noir

# enfiler un nouvel élément s dans la file f
# enfiler (s,f)

# tant que f non vide :

    # récupérer l'élément situé en tête de f,
    # le mettre dans la variable u et le supprimer de f
    # u ← defiler(f)

    # pour chaque sommet v adjacent au sommet u :

        # si v.couleur <> 'noir' :
            # v.couleur ← 'noir'
            # enfiler(v,f)
        # fin si

    # fin pour

# fin tant que

# Fin de l'algorithme
```

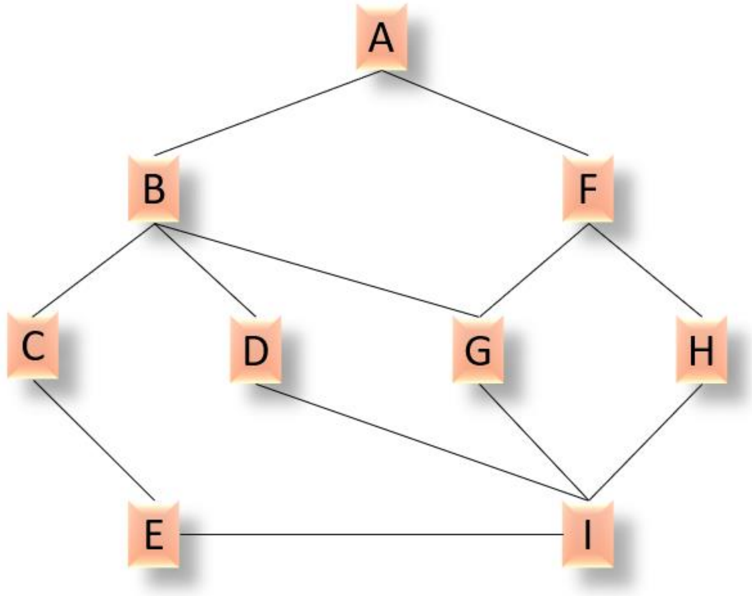
A faire vous-même IX

a. Faites tourner manuellement cet algorithme en l'appliquant au « Graf1 » suivant :

Les deux premières étapes sont les suivantes :

1 : file() ← Vide, s ← A, s.couleur ← noir (Sommet A découvert)

2 : file() ← A, s ← A, s.couleur ← noir (Sommet A découvert)



Graphe Graf1

Continuez le déroulement de l'algorithme

<u>*Aide :*</u> vous pouvez présenter le déroulement de l'algorithme en complétant le tableau suivant :

étape \ Variable	f	u	v	v.couleur, sommet découvert
3a tant que f <> vide	…			
3b u ← defiler(f)	…	…		
4a pour chaque sommet v	…	…	…	
4b si v.couleur <> noir	vide	A	B	
4c v.couleur ← noir	vide	A	B	noir, B
…	…	…	…	…
…	…	…	…	…
…	…	…	…	…
4g …	…	…	…	…
…	…	…	…	…
11a tant que f <> vide	E, H, G, D	C	E	noir, H
…	…	…	…	…
20f si v.couleur <> noir	Vide	I	G	noir, G

b. Quel ordre de parcours obtenez-vous ? Que remarquez-vous ? Pouvez-vous expliquer ce résultat ?

Ordre de parcours : ABFCDGHEI
L'ordre correspond a du plus proche au plus loin du point A

c. On définit la distance entre A et un sommet « destination » comme étant le nombre d'arêtes à parcourir depuis le sommet « origine » A pour arriver au sommet « destination ». En partant de cette définition, remplir le tableau suivant :

Sommets	A	B	F	C	D	G	H	E	I
Distance depuis A	0	1	1	2	2	2	2	3	3

d. En considérant la distance « d » qui sépare depuis le sommet « origine » A des autres sommets « s » du graphe, en déduire la règle de parcours (découverte) en largeur d'abord des sommets du graphe.

Le parcours en largeur va parcourir les sommets par ordre croissant de distance avec l'origine

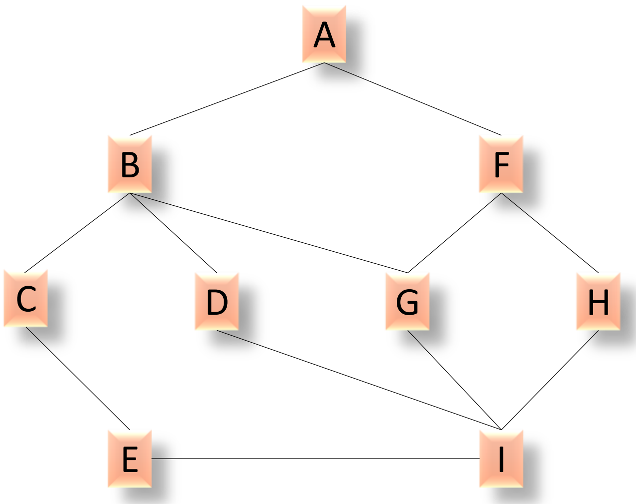
Le parcours en profondeur d'abord

Nous conservons le principe de signification des couleurs :

- si u.couleur = blanc => u n'a pas encore été "découvert"
- si u.couleur = noir => u a été "découvert"

Soit l'algorithme « profondeurGraphe_algo » de parcours en profondeur d'abord d'un graphe « Gr » :

```
VARIABLES
Gr : un graphe
u : nœud
v : nœud
# On part du principe que pour tout sommet u du graphe G, u.couleur = blanc à l'origine
DEBUT
profondeurGraf(Gr,u) :
    u.couleur ← noir
    affiche(u)
    pour chaque sommet v adjacent au sommet u :
        si v.couleur <> noir :
          profondeurGraf(G,v)
        fin si
    fin pour
profondeurGraf(Graf1,'A')
FIN
```



Graphe Graf1

A faire vous-même X

a. Faites tourner manuellement cet algorithme en l'appliquant au « Graf1 »

Aide : vous pouvez présenter le déroulement de l'algorithme en complétant le tableau suivant :

n° étape	instruction	u	u.couleur	v	v.couleur	Op.
1	profondeurGraf(G,u) :	A				
2	u.couleur ← noir	A	noir			
…	…	…	…	…	…	…
…	…	…	…	…	…	…
33	retour appelant 24 pour chaque sommet v adjacent u :	I	noir	G		
…	…	…	…	…	…	…
…	…	…	…	…	…	…
62	Fin pour 59	I	noir	H	noir	
…	…	…	…	…	…	…
…	…	…	…	…	…	…
72	Fin pour	A	noir	F	noir	

b. Quel ordre de parcours obtenez-vous ? Que remarquez-vous ? Pouvez-vous expliquer ce résultat ?

ABCEIHFGD

c. En partant du déroulement de l'algorithme, expliquez la règle de parcours (découverte) en profondeur d'abord des sommets du graphe.

L'algorithme parcours un chemin jusqu'au fond puis fait marche arriere pour trouver un autre chemin non decouvert a parcourir jusqu'au fond etc..

3.2 Présence d'un cycle dans un graphe

Rappel de quelques définitions :

Une « chaine » est une suite d'arêtes consécutives dans un graphe. On la désigne par les lettres des sommets qu'elle comporte. On utilise le terme de « chaine » pour les graphes non orientés et le terme de « chemin » pour les graphes orientés.

Un « cycle » est une chaine qui commence et se termine au même sommet.

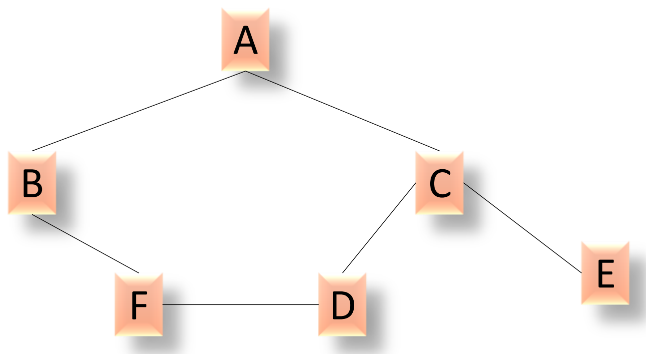
Pour différentes raisons, il peut être intéressant de détecter la présence d'un ou plusieurs cycles dans un graphe (par exemple pour savoir s'il est possible d'effectuer un parcours qui revient à son point de départ sans être obligé de faire demi-tour).

Nous allons étudier un algorithme qui permet de "détecter" la présence d'au moins un cycle dans un graphe :

```
# nom : cycleIterGraphe_algo.py
# rôle : détection d'un cycle dans un graphe
# propriété : appelle les fonctions empiler() et depiler()

# Début algorithme
# Variables et constantes
# Gr : graphe
# s, u, v : nœud
# p : pile()

# Initialisation
# Gr <-- Graf2
# s <-- 'B'
# p <-- pile()
# On part du principe que pour tout sommet u du graphe "Gr", u.couleur <-- 'blanc' à l'origine

# Déclaration de la fonction searchCycle()
# searchCycle():

  # empiler un nouvel élément sur la pile p
  # empiler (s,p)

  # tant que p non vide :

    # récupérer l'élément situé au sommet de p, le mettre dans la variable u et le supprimer de p
    # u ← depiler(p)

    # pour chaque sommet v adjacent au sommet u :

      # si v.couleur <> 'noir' :
        # v.couleur ← 'noir'
        # empiler(v,p)
      # fin si

    # fin pour

    # si u.couleur = 'noir':
      #retourner Vrai
    # sinon:
      # u.couleur ← 'noir'
    # fin si

  # fin tant que

  # renvoyer Faux

# searchCycle()
# Fin de l'algorithme
```

A faire vous-même XI

a. Faites tourner manuellement cet algorithme en l'appliquant au « Graf2 » ci-dessous
b. Quelle valeur retourne la fonction searchCycle() ?



Graphe Graf2

VRAI

4 Méthode « Diviser pour régner »

4.1 Qu’est-ce que c’est ?

La méthode « diviser pour régner » est une méthode algorithmique basée sur le principe suivant :

> On prend un problème (généralement complexe à résoudre), on divise ce problème en une multitude de petits problèmes, l'idée étant que les "petits problèmes" seront plus simples à résoudre que le problème original. Une fois les petits problèmes résolus, on recombine les "petits problèmes résolus" afin d'obtenir la solution du problème de départ.

Cette méthode repose donc sur 3 étapes :

- DIVISER : le problème d'origine est divisé en un certain nombre de sous-problèmes
- RÉGNER : on résout les sous-problèmes (les sous-problèmes sont plus faciles à résoudre que le problème d'origine)
- COMBINER : les solutions des sous-problèmes sont combinées afin d'obtenir la solution du problème d'origine.

4.2 Le tri fusion

Afin d’illustrer l’approche « diviser pour régner », nous allons étudier un algorithme de tri : le tri fusion. En classe de première, nous avons étudié des algorithmes de tri : tri insertion, tri sélection.

4.2.1 L’idée

Il s’agit d’abord de décomposer le jeu de données en parties séparées et à trier ces parties. Ensuite, il s’agit de fusionner ces parties de manière à obtenir une séquence triée. Le tri et la fusion se poursuivent jusqu’à ce que l’ensemble du jeu de données forme à nouveau un jeu de données unique.

4.2.2 La fusion

A. Première approche

Puisque la fusion intervient sur des parties triées, nous allons prendre un exemple pour lequel les données en entrée de la « fusion » sont deux tableaux triés :

indices	0	1	2	3
Tableau T1	8	15	46	72

indices	0	1	2	3
Tableau T2	9	26	33	46

A faire vous-même XII

a. A partir des tableaux T1 et T2, écrivez un algorithme « fusionTableaux_algo.py » qui fusionne T1 et T2 et qui génère le résultat de la « fusion » dans le tableau T_{fu} suivant :

indices	0	1	2	3	4	5	6	7
Tableau T_{fu}	8	9	15	26	33	46	46	72

<u>Aide</u> : vous pourrez éventuellement compléter l'algorithme fourni « fusionTableaux_algo.py »

b. A partir de « fusionTableaux_algo.py », écrire et tester le programme python « fusionTableaux.py »

B. Une alternative

Au lieu d'utiliser T1 et T2, on pourrait utiliser deux sous-tableaux directement dans T_{fu} :

- Le premier sous-tableau commence à $T_{fu}[0]$ et se termine à $T_{fu}[3]$
- Le deuxième sous-tableau commence à $T_{fu}[4]$ et se termine à $T_{fu}[7]$

indices	0	1	2	3	4	5	6	7
Tableau T_{fu}	8	15	46	72	9	26	33	46
	Sous-tableau n°1				Sous-tableau n°2			

A faire vous-même XIII

a. Ecrivez un algorithme « fusionSousTableaux_algo.py » qui fusionne les Sous tableaux n°1 et n°2 du tableau T_{fu} ci-dessus.

<u>Aide</u> : vous pourrez partir de l'algorithme « fusionTableaux_algo.py » complété

b. A partir de « fusionSousTableaux_algo.py », écrire et tester le programme python « fusionSousTableaux.py »

4.2.3 Le tri

L'idée : nous allons utiliser l'algorithme de fusion précédent (Une alternative) afin de trier une liste non ordonnée.

Cette fois en entrée, nous allons prendre notre tableau T_{fu} mais désordonné :

indices	0	1	2	3	4	5	6	7
Tableau T_{fu}	9	46	26	72	15	46	33	8

Considérons chaque élément de Tfu comme un sous tableau de longueur 1. En effet, Tfu peut être vu comme un tableau de huit sous tableau d'un élément. Chaque sous tableau ne comportant qu'un seul élément, il est de fait ordonné.

Or, on se souvient que notre algorithme (Une alternative), lorsqu'il prend en entrée des sous tableaux triés, produit en sortie un tableau trié.

Donc, si nous fusionnons, à la manière de notre algorithme (Une alternative), deux à deux les sous tableaux de Tfu, comme ceci :

indices	0	1	2	3	4	5	6	7
Tableau T_{fu}	9	46	26	72	15	46	33	8

Nous obtenons alors 4 sous tableaux ordonnés de deux éléments :

indices	0	1	2	3	4	5	6	7
Tableau T_{fu}	9	46	26	72	15	46	8	33

Puis nous recommençons et fusionnons deux à deux les quatre sous tableaux de Tfu, comme ceci :

indices	0	1	2	3	4	5	6	7
Tableau T_{fu}	9	46	26	72	15	46	8	33

Nous obtenons alors 2 sous tableaux ordonnés de quatre éléments :

indices	0	1	2	3	4	5	6	7
Tableau T_{fu}	9	26	46	72	8	15	33	46

A partir de là, nous sommes dans une configuration analogue à celle que nous avons défini en entrée de notre algorithme B. Nous connaissons le résultat :

indices	0	1	2	3	4	5	6	7
Tableau T_{fu}	8	9	15	26	33	46	46	72

Il s'agit bien là d'un tableau ordonné.

A faire vous-même XIV

a. Ecrire l'algorithme « triFusion_algo.py »

Aide : une boucle parcourt Tfu et fusionne des sous tableaux de longueur L en sous tableaux de longueur 2*L.

Au début, L ← 1

Deux variables x et y stockent les indices des sous tableaux à fusionner

Soit p, une variable qui sert à déterminer la position du premier et celle du dernier élément du sous tableau à fusionner. Au début de la boucle, L ← 1. On fusionne donc les deux sous tableaux :

- [p, p + L – 1]. L ← 1, le premier et le dernier élément du premier sous tableau ont ici le même indice (0 pour L ← 1)
- [p + L, p + 2*L]. L ← 1, le premier et le dernier élément du deuxième sous tableau ont ici le même indice (1 pour L ← 1)

A la manière de l'algorithme « fusionSousTableaux_algo.py », vous choisirez (en testant une condition sur la taille maximale des sous tableaux à fusionner et sur l'ordre des éléments), les éléments alternativement dans un sous tableau et dans l'autre, en augmentant d'une unité l'indice du sous tableau choisi, jusqu'à ce que l'indice ait atteint la taille maximale du sous tableau.

On fusionne les deux sous tableaux dans un tableau temporaire « Temp »

Puis, on teste la valeur de l'indice courant des sous tableaux :

- x est comparé à p + L
- y est comparé à p + 2*L

Lorsque les sous tableaux ont été entièrement parcourus, on traite les deux sous tableaux suivants de Tfu :

- p ← p + 2*L
- x ← p
- y ← p + L

Quand cette boucle est achevée, le tableau temporaire « Temp » est composé de sous tableaux triés de taille 2*L. On le recopie dans Tfu

On multiplie L par 2 et on recommence à fusionner les segments de taille supérieure. Cette opération peut être réalisée par une boucle externe (au début L ← 1 puis L ← 2 puis L ← 4).

b. A partir de « triFusion_algo.py », écrire le programme « triFusion.py »

A faire vous-même XIV

a. Nous allons écrire un nouvel algorithme en respectant les consignes suivantes :
   - Le tableau en entrée du tri fusion est généré de façon aléatoire et contient 128 (2^7) éléments compris entre 0 et 200
   - L'algorithme inclut la déclaration et l'appel de la fonction « fusion() »

   En partant de « triFusion_algo.py », écrire l'algorithme « triFusionNelts_fonction_algo.py » dans le respect des consignes précédentes.

b. A partir de « triFusionNelts_fonction_algo.py », écrire le programme « triFusionNelts_fonction.py »

A faire vous-même XV

a. En partant de « triFusionNelts_fonction_algo.py », écrire l'algorithme récursif « triFusionRecursif_algo.py ».
b. A partir de « triFusionRecursif_algo.py », écrire le programme « triFusionRecursif.py »

Pour aller plus loin, A faire vous-même XVI

Remarquons que jusqu'à présent, nous avons trié des tableaux de N entiers avec N correspondant à une puissance de 2. A présent, nous allons considérer le tri de tableaux dont le nombre d'éléments n'est pas une puissance de 2.

a. En partant de « triFusionRecursif_algo.py », écrire l'algorithme récursif « triFusionRecursifNelts_algo.py » qui réalise le tri fusion d'un tableau de N éléments générés aléatoirement avec N impair.

Aide : Voici une astuce impliquant une simple adaptation de l'algorithme précédent.

Si $N \neq 2^n$, nous complétons le tableau à un nombre N' tel que $N < N' <= 2^n$. La valeur des éléments choisis pour compléter le tableau doit être très supérieure à la plus grande valeur des éléments du tableau initial.

Une fois le tri fusion effectué, nous n'avons plus qu'à supprimer les éléments qui ont été ajoutés avant le tri. Nous obtenons ainsi en sortie, le tableau trié de N éléments.

b. A partir de « triFusionRecursifNelts_algo.py », écrire le programme « triFusionRecursifNelts.py »

***** Fin du document *****