

Chapitre 1 - Introduction à la programmation fonctionnelle avec Ocaml

- I - Présentation du language Ocaml
 - A - Brève chronologie
 - B - Spécificités du language
- II - Elements du language
 - B - Retour sur certains types
 - 1 - Types produits
 - 2 - Type des fonctions
 - 3 - Le type option
 - C - Filtrage par motif ou pattern matching
 - 1 - Syntaxe générale
 - 2 - Les filtres
 - 3 - Clauses gardées
 - 4 - Filtrage sur les listes
- III - Introduction à la récursivité
 - A - Exemple introductif

I - Présentation du language Ocaml

A - Brève chronologie

- 1958 : LISP, 1^{er} language fonctionnel
- 1970 : ML
- 1987 : Caml, 1^{ère} implantation publiée par des chercheurs de l'Inria
- 1996 : Ocaml, 1^{ère} version
 - programmation par objet
 - typage statique avec inférence

B - Spécificités du language

⌚ Propriétés

Ocaml est un language :

- fonctionnel (principalement)
- compilé
- avec typage statique : utilisation d'un symbole pour représenter le type d'une expression
- fortement typé : le compilateur infère et vérifie les types

II - Elements du language

B - Retour sur certains types

1 - Types produits

✍ Définition

On appelle **type produit** le type des couples (paires) ainsi que des n-uplets.

ⓘ Notation

Ces types sont `'a * 'b * ... * 'c`. Pour une paire uniquement, on accède à ses éléments l'aide des fonctions `fst` et `snd`.

ⓘ Syntaxe

```
fst : 'a * 'b -> 'a  
snd : 'a * 'b -> 'b
```

```
let x = (1, 'b')  
fst x (* 1 *)  
snd x (* 'b' *)
```

💡 Conseil

Une expression par **filtrage** permet de **déconstruire** une expression de type produit.

2 - Type des fonctions

ⓘ Rappel syntaxe

```
let f = (fun x -> x * x)  
let f x1 ... xn = expression  
let f = function  
| motif1 -> val1  
| ... -> ...  
let f a = match a with  
| motif1 -> val1  
| ... -> ...
```

⚠ Règle

Le type d'une fonction est définie par le type de ses paramètres et celui de sa valeur de retour.

Si `'a` et `'b` sont des types alors `'a -> 'b` est le type d'une fonction qui prend un paramètre de type `'a` et renvoie une valeur de type `'b`.

Dans le cas d'une fonction à plusieurs paramètres, une écriture sous [forme curryfiée](#) permet des applications partielles.

On écrit `f x y = expr` et `f` a pour type `(type de x) -> (type de y) -> (type de expr)`

3 - Le type option

✍ Définition

Il existe un type `option`, défini comme un type somme.

```
type 'a option = None | Some of 'a
```

☰ Exemple

Pour obtenir le 2eme élément d'une liste, on peut utiliser le type **option** :

```
let deuxieme l = match l with (* 'a list -> 'a option *)
| [] | [_] -> None
| e1 :: e2 :: q -> Some e2
```

Ce choix est cohérent car on est pas toujours sûr que ce 2eme élément existe.

💡 Tip

La [documentation](#) OCaml officielle possède plein de fonctions pratiques sur le type **option** !

C - Filtrage par motif ou pattern matching

✓ Intérêt

Le filtrage par motif permet d'effectuer une *disjonction de cas* sur la forme d'une expression.

1 - Syntaxe générale

ⓘ Syntaxe

```
match expr with
| filtre1 -> valeur1
| filtre2 -> valeur2
...
...
```

⚠ Règles

Les expressions **valeur1**, ... **valeur2** doivent être **de même type**.

2 - Les filtres

ⓘ Information

Les filtres sont essayés un par un de haut en bas. Le premier qui accepte l'expression **expr** est utilisé. Le symbole **_** (*underscore*, tiret du 8) représente le motif universel.

☰ Exemple

```
match (x, y) with
| a, true -> not a
| _, false -> false
```

💡 Important

- Le compilateur effectue un **test d'exhaustivité** : il génère un warning si un `match` risque de ne pas traiter tous les cas.
- Un warning est également généré si un cas est masqué par un cas écrit plus haut et qui se déclenchera avant dans tous les cas.

⌚ Règle fondamentale

Toute situation générant des `warning` soit être **impérativement** corrigée. S'il reste un `warning`, il reste un problème (spécification, conception, utilisation, ...).

3 - Clauses gardées

📋 Information

Le filtrage syntaxique par motif peut être étendu par des conditions booléennes à l'aide du mot clé `when`.

ⓘ Syntaxe

```
match expr with
| motif1 when condition -> expr1
| ...
```

☰ Exemple : réécriture du match ci-dessus

```
match (x, y) with
| a, b when b -> not a
| _ -> false
```

4 - Filtrage sur les listes

ⓘ Rappel

Le filtrage par motif est très souvent utilisé sur des listes.

Une liste Ocaml est une structure de données représentant une liste chaînée et permettant de rassembler des éléments de même type.

Une liste contenant les éléments `e1`, `e2`, ..., `en` est notée `[e1; e2; ...; en]`.

Notation : Elle est construite avec le constructeur `cons`, noté `::` : `e1::e2::...::en::[]`. `[]` représente une liste vide.

☰ Exemple

```
match l with
| [] -> "liste vide"
| [e] -> "liste d'un élément"
| e :: q -> "liste avec au moins deux éléments"
```

La notation `[e]` est équivalente à `e::[]`.

De manière générale, lorsqu'on filtre sur `e::q`, `q` peut très bien être une liste vide, mais dans l'expression ci-dessus, si on arrive au troisième filtre, c'est bien qu'on a une liste d'au moins 2 éléments.

III - Introduction à la récursivité

A - Exemple introductif

```
let rec factorielle n = function
| 0 | 1 -> 1
| _ -> n * factorielle (n-1)

let rec binome n k =
  if b = 0 || b = a then 1
  else (binome (a-1) (b-1)) + binome (a-1) b
```

Chapitre 2 - Notion de complexité

- I - Introduction
- II - Notations de Landau
- III - Applications
 - Le problème fondamental du tri
- IV - Vocabulaire et outils

Définition

Un **algorithme** est une succession finie d'opérations permettant de résoudre un problème. En général on a des données d'entrée et des données de sortie.

I - Introduction

Situation

Evaluer la **complexité** d'un algorithme permet d'évaluer son efficacité.

On peut s'intéresser au **temps d'exécution** ou à l'occupation en **espace mémoire**.

Pour évaluer la **complexité** d'un algorithme, on compte le nombre d'**opérations primitives** en fonction de la taille de l'entrée.

Une **opération primitive** est une opération réalisable en temps constant.

Appartenance d'un élément à une liste

```
let rec appartient x l =
  match l with
  | [] -> false
  | e :: q -> e = x || appartient x q
```

Opérations :

- comparaison à la liste vide
- récupération de l'élément de tête
- test d'égalité
- appel de fonction

Pour une liste de taille n , on va avoir $4n$ opérations.

$$T(n) = 4 + T(n - 1)$$

$$T(n) = 4 \times n$$

Le taux de croissance du temps d'exécution de cette fonction est linéaire.

II - Notations de Landau

Définition

Soient deux fonctions $f(n)$ et $g(n)$. On dit que $f(n)$ est "**Grand O**" de $g(n)$ que l'on note $O(g(n))$ si il existe une constante positive et un entier $n_0 \geq 1$ tel que $f(g) \leq c \times g(n) \forall n \geq n_0$

Notation

On note $f(g) = O(g(n))$

Signification

Ecrire $f(n) = O(g(n))$ signifie que le taux de croissance de f n'est pas supérieur à celui de g .

⚠ Règles

- Si f est un polynôme de degré d , alors $f(n)$ est $O(n^d)$
 - on ne tient pas compte des facteurs constants
 - on peut supprimer les termes de degré inférieur à d
- On utilise la classe la plus basse possible / l'expression la plus simple ($2n$ est $O(n)$ même si c'est aussi $O(n^2)$)

📝 Définitions supplémentaires

- $f(n)$ est $\Omega(g(n))$ si $\exists c > 0, \exists n \in \mathbb{N}^*$ tel que $f(n) \geq c \times g(n) \forall n \geq n_0$
- $f(n)$ est $\Theta(g(n))$ si $\exists c' > 0, \exists c'' > 0, \exists n_0 \in \mathbb{N}^*$ tel que $c' \times g(n) \leq f(n) \leq c'' \times g(n) \forall n \geq n_0$

III - Applications

☰ Puissance d'un nombre entier

```
let rec puissance a n =
  if n = 0 then 1
  else a * puissance a (n-1)
```

Complexité : On effectue n appels à la fonction puissance. A chaque, on effectue un test, une multiplication, une soustraction et un appel de fonction. Ces opérations s'effectuant en temps constant, on a donc "La complexité de la fonction puissance est $O(n)$ ".

```
let rec exponentielle a n =
  if n = 0 then 1
  else if n mod 2 = 0 then exponentielle (a*a) (n/2)
  else a * exponentielle (a*a) (n/2)
```

Complexité : A chaque appel on effectue un nombre fini d'opérations primitive. Puisque on divise n par 2 à chaque appel, on aura de l'ordre de $\log_2 n$ appels à la fonction exponentielle. "La complexité de exponentielle est $O(\log_2 n)$ "

☰ Recherche du maximum d'une liste

Lorsque l'on recherche le maximum d'une liste d'élément, on compare l'élément de tête au maximum de la suite de la liste. "La complexité d'une recherche de maximum est $O(n)$ ".

Le problème fondamental du tri

📝 Enoncé

On dispose d'une collection d'éléments que l'on souhaite organiser en respectant un ordre.

☰ Le tri par sélection

🔗 Principe

A chaque étape, on va sélectionner le plus petit élément.

⌚ Algorithme

Tant qu'il y a plus d'un élément restant, choisir le plus petit, le placer en tête et recommencer sur la suite.

Complexité : La complexité du tri par sélection est $O(n^2)$

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

☰ Le tri par insertion

⌚ Algorithme

Pour chaque élément à partir du deuxième, insérer l'élément parmi les éléments déjà triés.

Complexité : Dans le pire des cas lorsque les éléments sont arrangeés en ordre décroissant au départ, il faut effectuer $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$ déplacement en mémoire.

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

☰ Le tri fusion

⌚ Algorithme

Séparer en deux parties de même taille, trier récursivement chaque partie puis fusionner les deux parties.

Complexité : La complexité du tri fusion est $O(n \log_2 n)$

☰ Le tri rapide

⌚ Algorithme

Choisir un élément qui sera utilisé comme pivot, mettre à gauche tous les éléments plus petits que le pivot et à droite tous les éléments plus grands et trier récursivement les deux sous-parties.

Complexité : Dans le meilleur cas, on choisit un pivot qui est systématiquement l'élément médian. Dans ce cas la complexité du tri rapide est $O(n \log_2 n)$. En revanche, si on choisit un pivot qui conduit à laisser un seul élément plus petit ou plus grand alors on effectue n comparaisons.

La complexité dans le pire des cas du tri rapide est en $O(n^2)$

IV - Vocabulaire et outils

Fonction	Complexité	Exemple
Constante	$O(1)$	Affectation d'une variable
Logarithmique	$O(\log n)$	Recherche d'un élément dans une liste triée
linéaire	$O(n)$	Parcours d'une liste de longueur n
N-log N	$O(n \log n)$	Tri fusion d'une liste de taille n
Quadratique	$O(n^2)$	Tri par selection, double boucle
Cubique	$O(n^3)$	Triple boucle, multiplications de matrices
Exponentielle	$O(2^n)$	Tous les sous-ensembles
Factorielle	$O(n!)$	Toutes les permutations, tri stupide

Chapitre 3 - Correction des algorithmes

- I - Définitions
 - A - Spécification d'un problème algorithmique
 - B - Terminaison et correction
- II - Exemples en programmation récursive

I - Définitions

A - Spécification d'un problème algorithmique

Définitions

Les **spécifications** d'un problème comportent deux parties.

- la description des entrées admissibles
- la description du résultat ou des effets attendus (sorties)

On décrit les entrées admissibles par des contraintes appelées **préconditions**

On décrit les sorties attendues par des contraintes appelées **postconditions**

Une **assertion** permet de s'assurer qu'une condition est bien respectée à un endroit.

Syntaxe

En OCaml `assert` est une fonction de type `bool -> unit` qui lève une exception de type `Assert_failure` si la condition exprimée n'est pas évaluée à `true`.

Exemple

Vérifier qu'une liste est triée :

```
assert (est_croissante liste);
```

B - Terminaison et correction

Définition : terminaison

Une fonction **termine** si elle renvoie un résultat en un nombre fini d'étapes, quelles que soient les données d'entrée (en respectant les préconditions).

Définition : correction partielle

Une fonction est **partiellement correcte** si lorsqu'elle termine elle renvoie le résultat attendu.

Définition : correction totale

Une fonction est **totalement correcte** ou **correcte** si elle termine et elle est partiellement correcte.

Exemple

Cette fonction est partiellement correcte mais elle ne termine pas.

```
let rec fact_fausse n =
  if n = 1 then 1
  else n * fact_fausse (n+1)
```

II - Exemples en programmation récursive

Somme des n premiers entiers

```
let rec somme_n n =
  assert (n >= 0);
  if n = 0 then 0 else n + somme_n (n-1)
```

Terminaison :

- Si $n < 0$ la précondition matérialisée par le `assert` n'est pas respectée et la fonction lève une exception
- Si on a $n \geq 0$
- Le cas $n = 0$ vaut 0 d'après le code de la fonction
 - Lors de chaque appel récursif n décroît strictement de 1. Puisque \mathbb{N} n'admet pas de suite infinie on peut alors conclure que la suite des appels termine.

Correction :

Notons $P(n)$ la propriété. `somme_n n` vaut la somme des premiers entiers positifs, soit $\sum_{i=0}^n i$

Cas de base : Pour $n = 0$, d'après le code de la fonction on a $somme_n 0 = 0$ ce qui est égal à $\sum_{i=0}^0 i = 0$ Donc $P(0)$ est vraie.

Héritéité : Supposons que $P(n)$ est vraie. Alors on a, d'après le code de la fonction, $somme_n(n+1) = n + 1$

Par hypothèse de récurrence : $somme_n(n+1) = n + 1 + \sum_{i=0}^n i = \sum_{i=0}^{n+1} i$ Donc si $P(n)$ est vraie pour un n donné alors $P(n+1)$ est vraie.

Par principe de récurrence, $\forall n \geq 0$, $P(n)$ est vraie, ce qui prouve la correction de la fonction `somme_n`

Complexité :

Soit $T(n)$ le temps d'exécution de `somme_n n`. Lorsque $n = 0$ alors $T(n) = c$ où c est une constante représentant le temps nécessaire à renvoyer 0.

Sinon on a $T(n) = d + T(n-1)$ où d est une constante représentant le temps nécessaire pour effectuer une soustraction, une addition et un appel de fonction. $T(n)$ est une suite arithmétique de raison d . Alors pour tout $n \geq 0$ on a $T(n) = d \times n + c$ donc $T(n)$ est $O(n)$.

Insertion d'un élément dans une liste triée

```
let rec insere x l = match l with
| [] -> [x]
| y :: q -> if x <= y then x :: l else y :: (insérer x q)
(* 'a -> 'a list -> 'a list *)
```

Terminaison :

La fonction `insere` termine lorsque la liste passée en paramètre est vide ou lorsque x est inférieur ou égal à l'élément de tête de la liste.

Sinon la fonction effectue un appel récursif sur une liste de taille strictement inférieure.

Correction :

Soit $P(n)$: "Pour toute liste l croissante de longueur n et pour tout x , alors `insere x l` renvoie une liste croissante contenant les éléments de l et l'élément x "

Initialisation : Lorsque l est la liste vide, donc de longueur 0, `insere x []` renvoie `[x]` donc $P(0)$ est vraie.

Héritéité : Supposons que $P(n)$ est vérifiée pour toute liste de taille inférieur ou égale à $n \geq 0$.

Soit m la liste croissante de taille $n+1$ telle que $m = h :: t$

Par définition de `insere` nous avons

- si $x <= h$ alors `(insere x m) = x :: h :: t` qui est une liste croissante contenant les éléments de m et x .
- sinon `insere x m = h :: (insere x t)` or par hypothèse de récurrence `insere x t` est croissante et contenant tous les éléments de t et x .
Comme $h <= x$, h est aussi inférieur à tous les éléments de `insere x t`.
Par suite `h :: (insere x t)` est également croissante et contient tous les éléments de m et x .

Complexité :

- Dans le pire cas, l'élément x est plus grand que tous les éléments de la liste. Dans ce cas, on effectue n appels à la fonction `insere`. A chaque appel de `insere`, on a une reconnaissance de motif avec branchement, une comparaison, un ajout en tête et un appel de fonction.
On a donc $T(n) = a + T(n-1)$ et $T(0) = b$ avec a et b des constantes positives.
La complexité en pire cas de la fonction `insere` est $O(n)$.
- Dans le meilleur cas, l'élément x à insérer est inférieur ou égal à tous les autres éléments de la liste et la fonction renvoie `x :: l`
La complexité de `insere` dans le meilleur cas est $O(1)$.

☰ Tri d'une liste par insertion

```
let rec tri_insertion l = match l with
| [] -> []
| e::q -> insere e (tri_insertion q)
(* 'a list -> 'a list *)
```

Terminaison :

La fonction `tri_insertion` termine lorsque la liste est vide.

Sinon la fonction effectue un appel récursif sur une liste de taille `(n-1)` et un appel à `insere`. La fonction `insere` termine pour toutes les listes et les valeurs de `x`.

Alors pour toute liste `u` l'appel `tri_insertion` termine.

Correction :

Soit $P(n)$ la propriété "pour une liste `l` de taille `n` `tri_insertion` renvoie une liste constituée des éléments de `l` dans l'ordre croissant".

Initialisation : Lorsque la liste `l` est vide `tri_insertion []` renvoie la liste vide. Donc $P(0)$ est vraie.

Héritéité : Supposons que $P(n)$ est vraie pour toute liste `l` de taille `n`. Soit `m` une liste de taille `n+1` telle que `m=h::t`.

D'après le code de `tri_insertion` nous avons :

`tri_insertion m = insere h (tri_insertion t)`

On a par hypothèse de récurrence `tri_insertion t` est constituée des éléments `t` dans l'ordre croissant. Puisque la fonction `insere` est correcte, le résultat de `insere h (tri_insertion t)` est une liste croissante constituée des éléments de `m`.

Alors si $P(n)$ est vraie alors $P(n+1)$ est vraie.

Par principe de récurrence la fonction `tri_insertion` est correcte.

Complexité :

$$\begin{cases} T_{tri}(0) = a \\ T_{tri}(n+1) = b + T_{insere}(n) + T_{tri}(n) \end{cases}$$

En utilisant T_{insere} est $O(n)$:

$$\begin{cases} T_{tri}(0) = a \\ T_{tri}(n+1) = c + d \times n + T_{tri}(n) \end{cases}$$

Montrons par récurrence que $T_{tri} = d \sum_{i=0}^{n-1} i + n \times c + a$

Initialisation :

Pour $n = 0$, $T_{tri}(0) = a$

Pour $n = 1$, $T_{tri}(1) = c + d \times 0 + a = c + a$

$T_{tri}(1) = d \times \sum_{i=0}^0 i + 1 \times c + a = c \times a$

Donc $P(1)$ est vraie

Héritéité : $P(n)$ vraie pour $n \geq 1$

$T_{tri}(n+1) = c + d \times n + d \sum_{i=0}^{n-1} i + n \times c + a = d \sum_{i=0}^n i + (n+1) \times c + a$

Donc $P(n+1)$ est vraie.

$$T_{tri}(n) = d \frac{n \times (n+1)}{2} + (n+1) \times c + a$$

Donc la complexité de `tri_insertion` est $O(n^2)$.

Complexité dans le cas où la liste est déjà triée :

$$\begin{cases} T(0) = a \\ T(n) = b + c \times (n-1) + T(n-1) \end{cases}$$

Lorsque la liste es tdéjà triée, le temps passé dans est constant. On obtient les équations suivantes :

$$\begin{cases} T(0) = a \\ T(n) = e + T(n-1) \forall n \geq 1 \end{cases}$$

La complexité de `tri_insertion l` lorsque `l` est déjà triée est $O(n)$.

Chapitre 4 - Introduction

- I - Elements de langage
 - B - Retour sur certains points
 - 1. Les variables
 - 2. Les boucles
 - 3. Les fonctions
 - C - Les pointeurs
 - D - Les tableaux
- II - Elements avancés
 - A - Tableaux multidimensionnels
 - 1. Tableaux statiques
 - 2. Tableaux dynamiques
 - 3. Tableaux dynamiques contiguë à plusieurs dimensions

I - Elements de langage

B - Retour sur certains points

1. Les variables

🔗 Définition

Une **variable** est un espace mémoire auquel on associe un nom.

Représentation: `a[50]` où `a` est le nom de la variable et `50` sa valeur

ⓘ Syntaxe

```
int a = 50;  
// ou  
int a;  
a = 50;
```

C

🔗 Définition

Un **bloc** d'instruction est délimité par des accolades `{}`

ⓘ Syntaxe

```
int a = 50;  
{  
    a = 3;  
    int b = 25;  
}  
// a vaut 3  
// b n'est pas défini
```

C

2. Les boucles

⌚ Important

En C, `0` vaut faux et tout le reste vaut vrai. On renverra souvent `1` pour vrai et `0` pour faux.

🔗 Remarque

On peut utiliser les booléens `true` et `false` en important la librairie `stdbool.h`

```
#include <stdbool.h>  
bool a = true;  
bool b = false;
```

C

⚠ Règle (boucle `while`)

Il faut s'assurer qu'une situation pour laquelle **condition** sera évaluée à faux sera atteinte.

☰ Exemple

```
// Initialisation
int i = 0;
int n = 5;
int resultat = 0;
int p = 1;
// Boucle
while (i <= n) // condition
{
    resultat += p;
    p *= p;
    i++;
}
```

déroulement de la boucle :

n	i	p	resultat
5	0	1	0
5	1	2	1
5	2	4	3
5	3	8	7
5	4	16	15
5	5	32	32
5	6	64	63

i > 5 donc on s'arrête. $\sum_{i=0}^n 2^i = 2^{n+1} - 1$

⌚ Syntaxe

L'instruction `continue` permet de passer à l'itération suivante.
L'instruction `break` permet de stopper l'exécution d'une boucle.

☰ Exemples

```
int i = -1;
int n = 6;
int resultat = 0;
while (i <= n)
{
    i++;
    if (i % 2 != 0)
    {
        continue; // i est impair, la boucle passe à l'itération suivante
    }
    resultat += i;
}
// Ce code effectue la somme de n premiers entiers pairs
```

```
int i = 0;
int n = 5;
int resultat = 0;
while (1)
{
    resultat += i;
    i++;
    if (i >= n)
    {
        break; // i >= n donc on sort de la boucle, la boucle se termine
    }
}

// le code ci-dessus est équivalent au code ci-dessous, mais n'est pas recommandé

int i = 0;
int n = 5;
int resultat = 0;
while (i <= n)
{
    resultat += i;
    i++;
}
```

3. Les fonctions

Définition

On définit une **fonction** en écrivant son **type de retour**, son nom et, entre parenthèses, le type et le nom de chacun de ses arguments séparés pas des virgules.

Syntaxe

```
<type> <nom_de_la_fonction>(<type_arg_1> <nom_arg_1>, <type_arg_2> <nom_arg2>, ...)  
{  
    // code de la fonction  
}
```

Exemple

```
int somme(int a, int b)  
{  
    return a + b;  
}
```

Remarque

Tout code écrit après un **return** ne sera pas exécuté.
Si une fonction n'a pas de **return** et donc pas de type de retour, on utilisera le mot-clé **void**.

Exemple

```
void afficher_bonjour()  
{  
    printf("bonjour \n")  
}
```

Propriété

L'appel d'une fonction entraîne la copie des valeurs des variables.
Il s'agit de **passage par valeur**.

C - Les pointeurs

Définition

Un **pointeur** est une variable dont la valeur est une adresse en mémoire.

Exemple

$a[10] \leftarrow p$
 a est une variable entière dont la valeur est 10. Le type de a est **int**.
 p est un pointeur. p est une variable dont la valeur est l'adresse en mémoire de a . Le type de p est **int ***

Syntaxe

Définir un pointeur :

```
int *p; // pointeur de type int  
double *p; // pointeur de type double  
char *p; // pointeur de type char
```

Opérations :

Accès à l'adresse mémoire d'une variable : $\&$
Accès à la valeur contenu dans la case pointée par p : $*p$

☰ Exemples

```
C
int b = 25;
int *p = &b; // p contient l'adresse en mémoire de b
int c = *p; // ici, c prend la valeur de la case pointée par p, c'est-à-dire la valeur de b donc 25
*p = *p + 1; // b vaut maintenant 26, c vaut toujours 25
```

```
C
void mult_par_deux(int *p)
{
    *p = (*p) * 2;
}

int vitesse_tortue = 1;
mult_par_deux(&vitesse_tortue); // vitesse_tortue vaut maintenant 2
```

D - Les tableaux

🔗 Définitions

Un **tableau** est une suite de valeur stockées de manière consécutives en mémoire et telle que l'on puisse accéder à un élément à partir de son **indice** en temps constant. Un **indice** est le numéro d'un élément dans un tableau. En C, les indices d'un tableau de taille n vont de 0 à $n - 1$.

Représentation :

tab [0 2 4 6 8 10]
indice 0 1 2 3 4 5

🔗 Syntaxe

On déclare un tableau statique en donnant un type, son nom et sa taille.

```
<type> <nom_tableau>[<taille_tableau>];
```

Opérations : Accéder à la valeur d'un élément : **tab[i]** où **i** est un indice du tableau

☰ Exemple

```
C
int tab[6];
for (int i = 0; i < 6; i++)
{
    tab[i] = 2 * i;
}
// le résultat est le tableau dont la représentation est ci-précédente
```

⌚ Syntaxe

Initialiser les valeurs d'un tableau

```
C
int tab[6] = {0, 2, 4, 6, 8, 10};
```

⌚ Important

tab est un pointeur vers la première case du tableau. Le type de **tab** est donc **int *** dans l'exemple ci-dessus.
***tab** vaut donc **0**.

⌚ Syntaxe

Allouer un tableau dynamiquement

```
C
int *tab = (int*) malloc(sizeof(int)*6)
```

II - Elements avancés

A - Tableaux multidimensionnels

1. Tableaux statiques

Définition

Un tableau **statique** est un tableau dont on connaît la taille à la compilation.
Nous avons déjà vu comment déclarer un tableau statique.

Syntaxe

```
<type_du_tableau> <nom_du_tableau>[<nombre_elements_du_tableau>];  
  
// Exemple  
double a[5]; // a est un tableau de 5 éléments de type double  
int b[26]; // b est un tableau de 26 éléments de type int  
  
// Attention ici a et b ne sont pas initialisés.  
double c[] = {0.1, 0.2, 0.3};
```

C

Rappel

- si un tableau contient n éléments, les indices sont les valeurs entre 0 et $n - 1$ compris
- on accède à la valeur d'un élément d'un tableau avec la notation `tab[i]`

Règles

- Il n'est pas possible de comparer deux tableaux
- Il n'est pas possible d'affecter une valeur à un tableau

Syntaxe : tableaux 2D

Il est possible de déclarer des tableaux à plusieurs dimensions.

C

```
double c[50][10]; // c est un tableau de 50 tableaux de 10 doubles
```

Exemple

Passer un tableau en paramètre d'une fonction

C

```
void initialise(int tab[], int taille); // signature de la fonction
```

Pour un tableau à plusieurs dimensions seule la première dimension sera omise.

C

```
void initialise(int c[][10], int taille);
```

2. Tableaux dynamiques

Méthode

Pour allouer à l'exécution de l'espace mémoire, on utilise la fonction `malloc`.

`malloc` prend en paramètre la taille de l'espace mémoire à allouer en octets et renvoie l'adresse de la zone mémoire allouée.
On utilise `sizeof` pour connaître la taille d'un type.

Exemple

```
int* tableau = (int*) malloc(sizeof(int)*n); // où n est le nombre d'éléments du tableau
```

C

⚠ Gestion d'erreurs

En cas d'erreur, `malloc` renvoie le pointeur `NULL`. Il faut tester si `p == NULL` et arrêter l'exécution du programme si c'est le cas.

```
if (tableau == NULL)
{
    exit(EXIT_FAILURE);
}
```

C

3. Tableaux dynamiques contiguë à plusieurs dimensions

☰ Exemple

```
int m = 25;
int n = 10;
int* matrice = (int*) malloc(sizeof(int)* m * n);
if (matrice == NULL)
{
    exit(EXIT_FAILURE);
}
int i = 2;
int j = 3;
int valeur = matrice[i*n+j];
```

C

II - Éléments avancés

A - Tableaux multidimensionnels

Rapel Un *tableau* est une structure de données rassemblant des éléments de même type.

1. Allocation statique

voir cours déjà donné

2. Allocation dynamique - tableau 1D

voir cours déjà donné

Syntaxe Allouer à l'exécution un tableau contenant `n` éléments de type `int64_t` :

```
int64_t* p = (int64_t*)malloc(sizeof(uint64_t)*n);
// traitements
free(p);
```

Vocabulaire Effectuer un *cast*, c'est effectuer une opération de conversion d'un type vers un autre.

Dans l'exemple vu en cours, `malloc` renvoie l'adresse de la zone mémoire allouée sous la forme d'un `void *` que l'on *cast* en `int*`.

Pour allouer à l'exécution un tableau multidimensionnel, il faut déterminer sa représentation et sa taille en mémoire.

Règle toute zone mémoire allouée doit être libérée !

3. Allocation d'un bloc contigu

Comme on l'a vu en cours, la zone mémoire allouée par `malloc` est une zone linéaire dans la mémoire.

Pour représenter les données d'un tableau $M \times N$, le bloc de données prendra une taille en mémoire de : $M \times N \times s$ où s est la taille du type des éléments.

Ce sera au programmeur de gérer la manière dont il accède aux données.

Reprendons l'exemple d'une matrice :

```
int m = 25;
int n = 10;
double* matrice = (double*)malloc(sizeof(double)*m*n);
if (matrice == NULL)
{
    exit(EXIT_FAILURE);
}
// Accès à un élément d'indice i, j
double valeur = matrice[i*n+j];

// traitements
free(matrice);
```

Plus généralement, pour $i \in [0; m[$ et $j \in [0; n[$ toute fonction bijective qui prend un couple (i, j) et renvoie un entier a tel que $a \in [0; m \times n[$ convient comme fonction d'accès.

Exercice Si on utilise la fonction d'accès $f(i, j) = i + n \times j$, déterminer les coordonnées (i, j) lorsqu'on donne $a \in [0; m \times n[$.

4. Allocation de tableaux multidimensionnels avec indirection

Une deuxième possibilité est de déclarer un tableau d'indirection contenant des pointeurs qui contiennent chacun l'adresse de zones en mémoire différentes.

Reprenons le cas d'un tableau 2D contenant des `double` que l'on souhaite stocker en mémoire et accéder à l'aide d'un tableau de pointeurs.

Nous devons allouer :

- une zone en mémoire pour un premier tableau contenant les adresses des tableaux de second niveau
- pour chaque tableau du second niveau
 - une zone en mémoire pour contenir les données correspondant au tableau

```
int m = 25;
int n = 10;
double **matrice = (double**)malloc(sizeof(double*)*m);
if (matrice == NULL)
{
    exit(EXIT_FAILURE);
}
for(int i = 0; i < m; i++)
{
    matrice[i] = (double*)malloc(sizeof(double)*n);
    if (matrice[i] == NULL)
    {
        exit(EXIT_FAILURE);
    }
}

// traitements

// libération de la mémoire
for(int i = 0; i < m; i++)
{
    free(matrice[i]);
}
free(matrice);
```

I - Définitions

Définition Une *boucle* est une répétition d'instructions. Chaque passage dans la boucle est une *iteration*.

Rappel En C, deux mots clés permettent de définir des boucles : `for` et `while`. Pas d'inquiétude, il est aussi possible de définir des boucles en OCaml, comme nous le verrons plus tard.

Définition Un **variant** de boucle est une quantité : **entièr**e, **minorée**, qui décroît **strictement** à chaque passage dans la boucle.

Définition Un *invariant de boucle* est une propriété qui est vraie à l'initialisation d'une boucle et qui est conservée à l'issue d'un passage dans la boucle (si elle était vraie avant le passage).

II - Terminaison

Propriété Si une boucle admet un variant de boucle alors elle ne peut pas être infinie (donc elle termine).

A - Exemple de l'algorithme d'Euclide

Définissons en C une fonction utilisant l'algorithme d'Euclide pour calculer le PGCD de deux entiers a et $b > 0$.

```

1 int pgcd(int a, int b)
2 {
3     assert (b > 0);
4
5     int tmp;
6     while (b != 0)
7     {
8         tmp = b;
9         b = a % b;
10        a = tmp;
11    }
12    return a;
13 }
```

La **précondition** $b > 0$ est vérifiée ici à la **ligne 3**, en utilisant la fonction `assert`.

Montrons que b est un **variant** de la boucle `while`.
 b est :

une quantité entière ? b est un entier

minorée ? b est initialement strictement supérieur à 0 et l'exécution de la boucle s'arrête si $b = 0$

strictement décroissante ? à chaque passage dans la boucle, b prend la valeur de $a \% b$, c'est-à-dire le reste de la division euclidienne de a par b , quantité qui est, quelle que soit la valeur de a , **strictement** inférieure à b .

Ainsi, la boucle `while` admet un variant donc son exécution se termine, d'après la propriété ci-dessus.

B - Exemple de la recherche dichotomique

Nous nous intéressons à la recherche d'un élément entier dans un tableau contenant des entiers en ordre croissant.
Un exemple de code pour la fonction `recherche_dicho` est donné page suivante.

Les préconditions de la boucle sont :

- le tableau `tableau` contient des éléments en ordre trié
- l'indice `a` est un indice valide dans le tableau
- l'indice `a` est strictement inférieur à la valeur de `b` dont la valeur maximum possible est l'indice max du tableau + 1.

La fonction `recherche_dicho` contient une boucle `while`.

Montrons que la quantité représentée par $b - a$ est un variant de la boucle `while`.

- b et a sont des entiers, $b - a$ est un entier,
- la précondition impose que $a < b$, donc $b - a > 0$ et la boucle s'arrête si $a = b$, donc si $b - a = 0$, la quantité $b - a$ est donc minorée par 0,
- soit a et b vérifiant les préconditions : quelles sont les valeurs de a' et b' au prochain passage dans la boucle ?
D'après le code, on calcule $i = \left\lfloor \frac{a+b}{2} \right\rfloor$. Puisque $a < b$, alors $i < b$ et $i \geq a$. Dans le cas où la condition du test de la **ligne 10** n'est pas remplie, le test de la **ligne 15** implique deux cas possibles :
 - lorsque le test amène en **ligne 17**, alors $a' = a$ et $b' = i$. On a alors $b' - a' = i - a < b - a$,

- lorsque le test amène en **ligne 21**, alors $a' = i + 1$ et $b' = b$. On a alors $b' - a' = b - (i + 1) = b - i - 1$, or $i \geq a$ donc $b' - a' < b - a$.

Si la condition du test de la **ligne 8** est vérifiée, alors la boucle s'arrête.

Ainsi dans tous les cas, soit l'exécution de la boucle s'arrête, soit la quantité $b - a$ décroît strictement.

La quantité $b - a$ est donc un variant de la boucle. Cette dernière s'arrête.

```

1 int recherche_dicho(const int *tableau, int n, int e)
2 {
3     int res = -1;
4     bool trouve = false;
5     int a = 0;
6     int b = n;
7     while (a < b && !trouve)
8     {
9         int i = (a + b)/2;
10        if (tableau[i] == e)
11        {
12            trouve = true;
13            res = i;
14        }
15        else if (tableau[i] > e)
16        {
17            b = i;
18        }
19        else
20        {
21            a = i+1;
22        }
23    }
24    return res;
25 }
```

C. Remarques

L'analyse effectuée pour les boucles est très proche des raisonnements effectués pour garantir la terminaison d'une fonction récursive. Il est tout à fait possible de parler de *variant* lors de l'étude d'une fonction récursive si on a repéré une quantité entière, minorée et strictement décroissante... ou une quantité entière, majorée et strictement croissante.

III - Correction

Prouver la correction d'un algorithme comportant une boucle nécessite de montrer qu'une propriété intéressante est un *invariant* de la boucle.

Le raisonnement sera encore très proche de ce que nous avons fait pour prouver la correction des fonctions récursives. On démontre que la propriété \mathcal{I} est vraie juste avant la boucle, c'est l'initialisation.

Puis on démontre que si la propriété est vraie en début d'itération alors elle reste vraie à la fin de l'itération, c'est l'héritéité.

On conclut en utilisant l'invariant après arrêt de l'exécution de la boucle.

A. Exemple de l'algorithme d'Euclide

Rappelons les propriétés suivantes du `pgcd` pour deux entiers u et v positifs ou nuls et $u \bmod v$, le reste de la division euclidienne de u par v :

$$\text{pgcd}(u, 0) = u \tag{1}$$

$$\text{pgcd}(u, v) = \text{pgcd}(v, u \bmod v) \tag{2}$$

Invariant de boucle $\text{PGCD}(a, b) = \text{PGCD}(a_0, b_0)$ où a_0 et b_0 sont les valeurs de a et b à l'appel de la fonction.
Il est toujours possible de copier les valeurs d'entrées dans de nouvelles variables, pour faciliter le raisonnement.

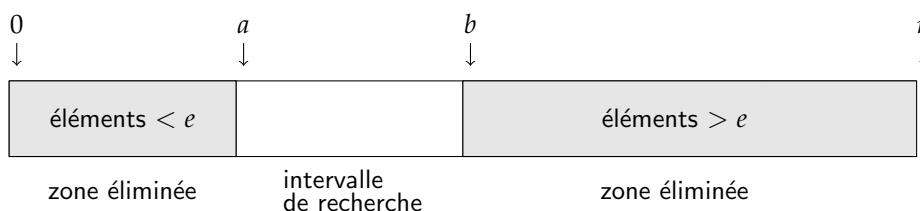
Initialisation Avant la boucle, a et b n'ont pas été modifiées et valent a_0 et b_0 donc $\text{PGCD}(a, b) = \text{PGCD}(a_0, b_0)$.

Héritéité Supposons que $\text{PGCD}(a, b) = \text{PGCD}(a_0, b_0)$ à l'entrée de la boucle. Alors, d'après le code, en sortie de boucle on a $a' = b$ et $b' = a \bmod b$. D'après la propriété (2), $\text{PGCD}(b, a \bmod b) = \text{PGCD}(a, b)$, donc $\text{PGCD}(a', b') = \text{PGCD}(a, b)$ et par hypothèse de récurrence, $\text{PGCD}(a', b') = \text{PGCD}(a_0, b_0)$.

Conclusion L'invariant étant conservé, à la suite de la dernière itération, on a $\text{PGCD}(a, b) = \text{PGCD}(a_0, b_0)$. Or puisque la boucle se termine, on a $b = 0$ et d'après la propriété (1), $a = \text{PGCD}(a_0, b_0)$.

B. Exemple de la recherche dichotomique

Voici une représentation sur un tableau de taille n de l'effet de la recherche dichotomique en cours d'exécution.



Si à la sortie de la boucle `res != -1`, alors `res` contient l'indice d'un élément du tableau dont la valeur est égale à l'élément recherché `e`.

Il faut montrer que lorsque le résultat est `-1`, la valeur de `e` ne se trouvait pas dans le tableau.

Nous allons prouver l'invariant suivant : « avant l'indice `a`, tous les éléments du tableau sont strictement inférieurs à l'élément `e` cherché et à partir de la position `b`, tous les éléments du tableau sont strictement supérieurs à l'élément `e` cherché ».

Initialisation Avant le premier passage dans la boucle, d'après le code on a `a = 0` et `b = n`. Dans ce tableau d'indices compris dans l'intervalle $[0; n - 1]$, il n'y a pas d'élément avant 0, ni d'élément à partir de n . L'élément cherché ne peut donc se trouver ni dans la partie d'indices inférieurs à `a` ni dans la partie de positions supérieures ou égales à `b`. Donc l'invariant est vérifié.

Héritéité Supposons l'invariant vérifié à l'entrée d'une boucle. Alors les éléments avant l'indice `a` sont strictement inférieurs à l'élément cherché `e` et les éléments à partir de l'indice `b` sont strictement supérieurs à l'élément `e` cherché. Supposons que l'élément au milieu `i` de l'intervalle ne soit pas égal à l'élément `e` cherché. Deux cas sont possibles :

- si l'élément `tableau[i]` est strictement supérieur à `e`, alors puisque le tableau est en ordre croissant, l'élément `tableau[i]` est inférieur ou égal à tous les éléments d'indice supérieurs à `i` et l'élément cherché ne peut pas se trouver dans la partie comprise entre l'indice `i` inclus et l'indice `b`. Par hypothèse de récurrence, l'élément cherché ne peut pas se trouver dans la partie située à partir de l'indice `b` donc il ne peut pas se trouver dans toute la partie située à partir de l'indice `i`. Or d'après le code, à la fin de l'itération on a `b' = i` et `a' = a`. Puisque les éléments dans la partie située avant l'indice `a` sont par hypothèse de récurrence strictement inférieurs à l'élément cherché, alors les éléments d'indice inférieur à `a'` sont strictement inférieurs à l'élément `e` et les éléments d'indice supérieur ou égal à `b'` sont strictement supérieurs à `e`. Donc les propriétés de l'invariant sont conservées.
- si l'élément `tableau[i]` est strictement inférieur à `e` alors puisque le tableau est en ordre croissant et par hypothèse de récurrence comme précédemment, l'élément cherché est strictement supérieur à tous les éléments situés de l'indice 0 à l'indice `i` inclus. À la fin de l'itération on a `a' = i + 1` et `b' = b`. Par hypothèse de récurrence, les éléments d'indice supérieur ou égal à `b` sont strictement supérieurs à l'élément cherché. Encore une fois, les éléments situés avant `a'` sont strictement inférieurs à `e` et les éléments situés après `b'` sont strictement supérieurs à `e` et les propriétés de l'invariant sont conservées.

Conclusion Si l'élément n'est pas trouvé, nous avons montré que l'exécution de la boucle s'arrête lorsque $a = b$. Dans ce cas, il n'y a plus d'éléments à considérer dans le tableau, l'intervalle $[[a, a]]$ étant vide. Et puisque l'invariant est vérifié, l'élément cherché est strictement supérieur à tous les éléments de l'intervalle $[0, a - 1]$ et strictement inférieurs à tous les éléments situés dans l'intervalle $[a, n - 1]$. On peut donc affirmer que l'élément cherché ne se trouvait pas dans le tableau, le résultat `-1` est correct.

Introduction

Ce TP permet de s'approprier de nouveaux éléments de langage, en OCaml et en C et de revenir sur certains éléments.

Plan de cours

Voici le plan du cours qu'il vous faudra avoir complété à l'issue du TP ou d'ici au prochain cours.

Chapitre 6 - Types structurés

I_ En OCaml

- A_ Nommer un type
 - + annotation de types dans les fonctions
- B_ Types sommes
 - 1. définition, notion de Constructeur
 - 2. filtrage
 - 3. types récursifs

I_ C_ Enregistrements à champs mutables

II_ En C

- A_ Renommer un type
- B_ Structures

1 Types structurés, OCaml

1.1 Retour sur les enregistrements

Nous avons déjà utilisé le mot-clé `type` pour définir des *enregistrements*... (premier TP !)

QUESTION 1

Définir un type `morceau` qui représente un morceau de musique et qui devra contenir les champs `titre`, `duree`, `auteur`, `annee` avec des types appropriés pour chacun des champs.

QUESTION 2

Utiliser ce type pour définir deux variables `m1` et `m2` représentant des morceaux de musique. Quel est le type de ces deux variables ?

Supposons que vous souhaitiez trier les éléments d'une liste contenant des morceaux de musique selon leur titre. Vous avez à votre disposition la documentation en anglais de `List.sort` :

*Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller.*¹

QUESTION 3

Créer une liste `catalogue` contenant les variables `m1` et `m2`. Utiliser `List.sort` pour effectuer ce tri en écrivant une fonction anonyme et stocker le résultat dans une variable `l2`.

QUESTION 4

Quel est le type de la fonction anonyme que vous avez écrite ?

QUESTION 5

Définir un nouveau type nommé `perturbation` qui contient au moins un champ dont le nom est `titre`.

1. <https://v2.ocaml.org/api/List.html>

QUESTION 5

Tester de nouveau l'appel permettant de trier la liste `catalogue`. Que se passe-t-il ? Quel est le type de la fonction anonyme utilisée ?

Le compilateur OCaml associe les éléments rencontrés au type le plus proche dans le contexte. Le type enregistrement `perturbation` a un champ `titre`. La fonction de comparaison est donc compatible avec un type `perturbation -> perturbation -> int`. Mais ce n'est pas le cas de la variable `catalogue` à laquelle la fonction est appelée !

1.1.1 Annotations de type

En OCaml, il est possible d'utiliser des annotations de types dans la déclaration des fonctions. Voici un exemple :

```
let f (a : int) (b: bool) = if b then a else (-a)
```

Pour chaque paramètre, on indique donc, entre parenthèses, son nom, suivi du symbole `:` puis du type du paramètre.

QUESTION 6

Corriger l'appel à `List.sort` en utilisant les annotations de types, cela doit fonctionner malgré la définition de `perturbation` ...

1.2 Renommer les types

En fait, il est également possible de définir des types pour donner un autre nom, un *alias* à n'importe quel type ou combinaison de type.

Voici un exemple en utilisant un type produit :

```
type coordonnees = int * int
```

QUESTION 7

Quel est le type de `coordonnees` ? Définir une variable `origine` en donnant simplement les coordonnées de l'origine d'un repère. Quel est le type de `origine` .

QUESTION 8

Définir une variable `point` représentant le point de coordonées (3, 2). Faire en sorte que `point` soit de type `coordonnees` .

1.3 Types construits

On parle de type *construit* lorsqu'on utilise un *constructeur*. Un constructeur est un nom que l'on donne à un type d'objet... Ce sera plus clair avec des exemples !

RÈGLE 1

Le nom d'un constructeur doit obligatoirement commencer par une lettre capitale. Ce sont les seuls objets autorisés à commencer par une lettre capitale en OCaml.

Un constructeur peut avoir un argument, et dans ce cas on donnera une valeur associée. Sinon il s'agit d'un constructeur *constant*.

Type construit constant

```
type unecouleur = Bleu
let b = Bleu
```

Ici `Bleu` est un constructeur de type `unecouleur`.

`l` est une variable de type `unecouleur` dont la valeur est `Bleu`.

Pour donner un argument à un constructeur on utilise le mot-clé `of` suivi du type du paramètre.

Type construit paramétré

```
type forme = Petales of int
let f = Petales 3
```

Ici `Petales` est un constructeur de type `forme`, paramétré par un entier.

`f` est une variable de type `forme` dont la valeur est `Petales 3`.

QUESTION 9

Tester les exemples précédents et vérifier ce qui est affiché par l'interpréteur OCaml.

1.4 Types sommes

Question Et si on souhaite avoir plusieurs valeurs pour un même type ?

Réponse on utilise un *type somme* (appelé aussi *type union*) !

On avait déjà les types produits, les types enregistrements, les types flèches... il nous manquait les sommes, non ?!

Type somme

```
type couleur = Bleu | Rouge | Jaune | Vert | Violet | Orange
```

`Bleu`, `Vert`, `Jaune` ... sont des constructeurs de type `couleur`.

QUESTION 10

Définir une variable de type `couleur` dont la valeur est `Jaune`.

QUESTION 11

Définir un type `fleur` comme le type produit d'une couleur et d'une forme.

QUESTION 12

Définir une variable `bouton_d_or` de type `fleur` sachant qu'un bouton d'or a 5 pétales de couleur jaune.

Il est possible de tester l'égalité d'une variable d'un type avec un constructeur.

```
let c = Rouge
let est_jaune = c = Jaune
```

Quelle est le type et la valeur de `est_jaune` ???

1.5 Filtrage

Il est bien sûr possible d'utiliser le filtrage par motif sur un type somme...

QUESTION 13

Écrire une fonction `melange` qui prend deux couleurs sous la forme d'un produit de couleurs et détermine la couleur obtenue si on les mélange comme on le ferait avec de la peinture (synthèse soustractive). On pourra ajouter une couleur `Inderterminée` au type `couleur` défini précédemment pour traiter les cas non évidents ou non définis.

1.6 Types sommes récursifs

Un type peut être récursif et faire appel à lui-même. Aucun mot-clé n'est nécessaire pour le spécifier.

Reprenons notre type couleur de la manière suivante

Couleur : type récursif

```
type couleur =
| Bleu
| Rouge
| Jaune
| Melange of couleur * couleur

let orange = Melange (Jaune, Rouge)
```

`orange` est une variable de type `couleur` dont la valeur est `Melange` du couple (`Jaune`, `Rouge`). Chaque élément du couple étant une variable de type `couleur`.

QUESTION 14

En utilisant le type précédent, définir une variable `emeraude` qui représente une couleur obtenue avec du vert et du bleu.

Il est bien sûr toujours possible d'utiliser le filtrage par motif sur ce type couleur.

QUESTION 15

Que réalise la fonction suivante ?

```
let rec mystere c = match c with
| Rouge | Jaune -> false
| Bleu -> true
| Melange (c1, c2) -> mystere c1 || mystere c2
```

2 En C

2.1 Définir de nouveaux types

En C aussi, il est possible de donner un nouveau nom, un *alias* à un type existant en utilisant le mot-clé `typedef` et la syntaxe `typedef <un type existant> <nouveau nom>`.

```
typedef int entier;
typedef int* ptr_entier;
typedef entier* ptr_entier_v2;
```

Après la première ligne, `entier` est un type synonyme de `int`. Puis `ptr_entier` est un nouveau nom pour `int*`, comme l'est aussi `ptr_entier_v2`.

2.2 Structures

En C, les *structures* permettent de rassembler des éléments de types différents.

Par exemple, on souhaite créer une structure contenant le prénom d'une personne et son âge.

Déclaration d'une structure

```
struct personne_s {  
    char prenom[256];  
    int age;  
};
```

Ceci déclare une structure, un objet dont le **type** sera `struct personne_s`.

Pour construire une variable du type `struct personne_s`, on peut déclarer une variable locale (`personne1`) ou allouer l'espace mémoire nécessaire avec `malloc` (`personne2`).

```
struct personne_s personne1;  
  
struct personne_s* ptr_personne = malloc(sizeof(struct personne_s));
```

Dans les deux cas ci-dessus, la structure n'est pas initialisée. Pour accéder aux différents champs à partir d'une variable du type de la structure, on utilise la notation `.` :

```
strncpy(personne1.prenom, "Toto", 256);  
personne1.age = 23;
```

Il est aussi possible d'initialiser les champs de la structure directement lors de la déclaration avec :

```
struct personne_s personne2 = { .prenom = "Titi", .age = 24};
```

QUESTION 16

Avec les éléments donnés jusqu'à présent, écrire les lignes de code pour que le prénom associé à la structure pointée par `ptr_personne` soit `"Tata"` et que son âge soit `50`.

Pour accéder au champ de la structure, il faut déréférencer le pointeur et donc écrire `(*ptr_personne).age`, par exemple. En pratique, il existe une autre syntaxe plus directe lorsque la variable est un pointeur vers une structure.

SYNTAXE

Si `p` est un pointeur vers une structure qui possède un champ nommé `b`, alors `(*p).b` s'écrit aussi `p->b`.

```
struct personne_s* ptr_pers2 = &personne2;  
ptr_pers2->age = ptr_pers2->age + 1;
```

Enfin, il est possible de donner un nouveau nom à un type structure :

```
typedef struct personne_s personne;
```

QUESTION 17

Écrire une fonction `afficher_personne` qui prend en argument un pointeur vers une structure de type `personne` et affiche le prénom de la personne suivi de son âge.

3 Enregistrements

À quoi vous font penser les structures du C ?

Que manque-t-il aux enregistrements du langage OCaml pour pouvoir être utilisés de la même manière que des structures C ?

QUESTION 18

Définir un enregistrement `personne` en OCaml qui aurait les mêmes champs que la structure C définie dans la partie précédente.

Comme on l'avait dit en début d'année, les champs d'un enregistrement OCaml ne peuvent pas changer de valeur, par défaut. Il s'agit d'une structure *immutable*, comme les listes OCaml.

En fait, il existe la possibilité de préciser qu'un champ peut être *mutable*, c'est-à-dire que sa valeur peut changer sans générer la création d'une nouvelle variable.

```
type personne = {nom : string; mutable age : int}
```

QUESTION 19

En utilisant la définition précédente du type enregistrement `personne`, et toujours en OCaml, définir une variable `personne1` pour représenter les informations d'une personne nommée Ada et dont l'âge est 20 ans.

Pour modifier la valeur d'un champ *mutable* d'un enregistrement, il faut utiliser l'opérateur `<-`. Pour augmenter l'âge d'une personne `p`, on écrirait `p.age <- p.age + 1 ...`

Mais ceci génère un effet de bord (modification d'une variable). Nous reviendrons sur les *traits impératifs* de OCaml ultérieurement !

4 Exercices

Lorsque vous avez tout bien terminé dans le TP et rédigé votre cours en suivant le plan proposé en introduction, vous pouvez vous entraîner avec les exercices proposés dans cette partie.

4.1 Type somme et révisions sur les listes

On définit le type somme suivant : `type t = B | N | R`.

QUESTION 20

Écrire une fonction `permute` qui étant donné une liste d'éléments du type `t` renvoie une nouvelle liste dans laquelle les valeurs B sont remplacées par N, les valeurs N par R et les valeurs R par B.

QUESTION 21

Écrire une fonction `compte` qui prend en paramètre une liste d'éléments du type `t` et un élément du type `t` et qui compte le nombre d'apparitions de l'élément dans la liste.

Annoter `compte` pour qu'elle soit de type `t list -> t -> int`.

QUESTION 22

Écrire une fonction `plus_grande_sequence` qui renvoie la longueur de la plus grande séquence de B dans une liste passée en paramètre. Par séquence on entend suite continue d'un élément.

QUESTION 23

Écrire toutes les fonctions précédentes en utilisant une fonction de l'api `List` (`List.map`, `List.fold_left` par exemple)... si ce n'est pas déjà le cas !

4.2 Entiers naturels de Peano

Voici une définition non exhaustive des entiers naturels dans l'arithmétique de Peano :

- le nombre appelé Zéro et noté 0 est un entier naturel
- si n est un entier naturel, alors son successeur $S(n)$ est un entier naturel.

Ainsi le nombre $S(S(S(0)))$ est un entier naturel, qui correspond à 4.

QUESTION 24

Définir un type somme **récursif** `nat` qui permettra de représenter les entiers naturels. On notera `Zero` le constructeur du nombre Zéro et `Succ` le constructeur d'un successeur.

QUESTION 25

Définir une fonction `est_nul` qui renvoie `true` si l'entier de Peano en paramètre est `Zero`, `false` sinon.

QUESTION 26

Écrire une fonction `valeur_entier` qui calcule la valeur entière d'un entier naturel de Peano. Donner son type.

QUESTION 27

Écrire la fonction réciproque, nommée `peano` de la fonction précédente. Donner son type.

QUESTION 28

En exploitant le fait que $(n + 1) + m = (n + m) + 1$, écrire une fonction `addition` qui effectue l'addition de deux entiers naturels de Peano sous la forme d'un entier naturel de Peano. Donner son type.

III - Retour sur l'allocation des structures en C

```
#include <string.h>

// défini la structure personne_
struct personnes_s {
    char nom[256],
    int age,
};

typedef struct personne_s personne; // personne est l'alias du type (on le renomme)
```

C

```
// Allocation d'une structure :
personne *p = (personne *) malloc(sizeof(personne));

// Accès à un élément :
(*p).age = 50;
p -> age = 50;
strncpy(p->nom, "Bidule", 256) // on doit utiliser strncpy pour affecter des chaînes de caractères
```

C

```
// Allocation dynamique d'un tableau de structure :
personne *tab = (personne *) malloc(sizeof(personne) * n);

free(tab); // ne pas oublier de libérer l'espace mémoire
```

C

```
void initialise(personne *tab, int n)
{
    for (int i = 0; i < n; i++)
    {
        tab[i].age = 20;
        sprintf(tab[i].nom, "Personne%d", i);
    }
}
```

C

Nous nous intéressons à la représentation des données en machine. L'informatique travaille classiquement avec des *bits*, des unités permettant de représenter deux états logiques. D'où leur nom qui provient de la contraction des mots anglais *binary digits*. Les symboles utilisés pour représenter la valeur de bits sont les deux premiers chiffres du système décimal : 0 ou 1.

I_ Représentation des entiers

A - Entiers naturels dans une base quelconque

1. Notation positionnelle

Théorème Soit $\beta \in \mathbb{N}$, $\beta > 1$, une base. Tout entier $n \in \mathbb{N}$ peut être représenté de manière unique par sa **représentation positionnelle en base β** :

$$(a_{p-1}a_{p-2}\dots a_1a_0)_{\beta} : n = \sum_{i=0}^{p-1} a_i \beta^i$$

avec $a_i \in 0, \dots, \beta - 1$ et $a_{p-1} \neq 0$.

Les notations $(a_{p-1}a_{p-2}\dots a_1a_0)_{\beta}$ ou $\overline{a_{p-1}a_{p-2}\dots a_1a_0}^{\beta}$ permettent de représenter cette somme. p est le nombre de chiffres nécessaires pour écrire l'entier n .

Exemple : $(3205)_{10} = 3 \times 10^3 + 2 \times 10^2 + 0 \times 10^1 + 5 \times 10^0 = 3205$ en écriture décimale.

Vocabulaire : On appelle chiffre de *poids faible* le chiffre qui, dans la représentation positionnelle en base β , est associé à la plus petite puissance de β . À l'inverse, le chiffre de *poids fort* est le chiffre non nul associé à la plus grande puissance de β .

En base 2, on parlera donc de *bit de poids faible* et de *bit de poids fort*.

Le *chiffre de poids fort* est par convention représenté à gauche dans la représentation positionnelle usuelle.

2. Divisions euclidiennes successives

Le reste dans la division euclidienne d'un entier naturel n par β donne le chiffre de poids faible dans la représentation de n en base β .

En effet,

$$\begin{aligned} n &= a_{p-1}\beta^{p-1} + a_{p-2}\beta^{p-2} + \dots + a_2\beta^2 + a_1\beta^1 + a_0, \\ &= \underbrace{(a_{p-1}\beta^{p-2} + a_{p-2}\beta^{p-3} + \dots + a_2\beta^1 + a_1)}_{\text{quotient}} \beta + \underbrace{a_0}_{\text{reste}} \end{aligned}$$

avec, rappel, $0 \leq a_0 < \beta$. Donc $n \bmod \beta = a_0$.

3. Existence de la représentation positionnelle

Démontrons l'existence de la représentation positionnelle dans une base $\beta > 1$ pour tout nombre entier $n \in \mathbb{N}$ par récurrence forte sur n .

Initialisation Pour $n = 1$, $n = 1 \times \beta^0$ puisqu'on choisit $\beta > 1$. Remarque : pour $n = 0$, la représentation est vide par convention.

Héritéité Supposons l'existence de la représentation positionnelle pour tout nombre entier k tel que $0 \leq k < n$ et considérons le nombre n .

Puisque $n > 0$ et $\beta > 1$, il existe des entiers naturels q et r tels que $n = q\beta + r$ avec $0 \leq r < \beta$ et $0 \leq q < \beta$ ¹. Si $q = 0$, on a terminé : $n = r\beta^0$.

Sinon, puisque $q < n$, l'hypothèse de récurrence amène $q = \sum_{i=0}^{p-1} a_i \beta^i$, avec $a_{p-1} \neq 0$.

1. voir cours de maths

On obtient

$$\begin{aligned} n &= \beta \sum_{i=0}^{p-1} a_i \beta^i + r \\ &= \sum_{i=1}^p a_{i-1} \beta^i + r \\ &= \sum_{i=0}^p \bar{a}_i \beta^i \quad \text{avec } \bar{a}_0 = r \quad \text{et } \bar{a}_i = a_{i-1} \quad \text{pour } i \geq 1 \end{aligned}$$

Les \bar{a}_i sont tels que $0 \leq \bar{a}_i < \beta$ et $\bar{a}_p = a_{p-1} \neq 0$. \square

4. Nombres de chiffres représentables

En base β , un nombre écrit avec p chiffres pourra représenter β^p valeurs différentes. En choisissant la première valeur à 0, les nombres représentables seront donc compris entre 0 et $\beta^p - 1$.

5. Codage en machine

En machine, le nombre de bits p de la représentation positionnelle est fixé pour les différents *formats de codage* (entiers non signés de taille 8, 16, 32 ou 64 bits).

Lorsqu'on effectue une opération arithmétique entre entiers naturels (types `unsigned` en C) sur p bits, le résultat m est obtenu mod 2^p (mod représentant le modulo, c'est-à-dire le reste de la division euclidienne).

B - Entiers relatifs en base 2

Il existe plusieurs manières de représenter un entier relatif.

La représentation choisie est dite la *représentation en complément à 2 sur p bits*.

$$(c_{p-1}c_{p-2}\dots c_1c_0)_2 : -c_{p-1}2^{p-1} + \sum_{i=0}^{p-2} c_i 2^i$$

On peut montrer que :

$$\begin{cases} n \geq 0 & \text{ssi } c_{p-1} = 0, \\ n < 0 & \text{ssi } c_{p-1} = 1. \end{cases}$$

Opposé en complément à 2 sur p bits Le codage de $-n$ en complément à 2 sur p bits est le même que celui de l'entier naturel $(\bar{c}_{p-1}\bar{c}_{p-2}\dots\bar{c}_1\bar{c}_0)_2 + 1 \bmod 2^p$, avec \bar{c} valant 1 si $c = 0$ et 0 si $c = 1$, et n représenté par $(c_{p-1}c_{p-2}\dots c_1c_0)_2$.

Addition en complément à 2 sur p bits Soit $m = (c_m)_2$ et $n = (c_n)_2$. En l'absence de dépassement de capacité, le codage de $m + n$ en complément à 2 sur p bits est le codage de $(c_m + c_n) \bmod 2^p$ en tant qu'entier naturel. Si m et n sont de signes opposés, aucun dépassement n'est possible. Si m et n sont de même signe, il y a dépassement si et seulement si le signe du résultat calculé diffère du signe des opérandes.

II_ Vers les nombres flottants

A - Représentation de rationnels

Les rationnels sont les nombres de la forme $\frac{p}{q}$, avec $p \in \mathbf{N}$ et $q \in \mathbf{N} - \{0\}$.

L'écriture d'un rationnel en base β n'est pas forcément finie, mais si elle n'est pas finie, elle est nécessairement périodique. Soit un nombre x tel que $0 \leq x < 1$ dont on connaît l'écriture décimale en base 10. On veut déterminer son écriture en binaire de la forme :

$$x = (0, x_1 \dots x_q)_2$$

Puisque $2x = (x_1, x_2 \dots x_q)_2$ on a $x_1 = [2x]$.

Exemple convertir $1/10 = (0,1)_{10}$ en binaire.

II_ Vers les nombres flottants

A - Représentation de rationnels

RAPPEL 1

Les rationnels sont les nombres de la forme $\frac{p}{q}$ avec $p \in \mathbf{N}$ et $q \in \mathbf{N}^*$.

Certains rationnels ont une écriture finie en base 10. Les autres ont une écriture périodique.
On détermine une *forme normalisée* pour les nombres rationnels (fraction irréductible) de la manière suivante : $\frac{p}{q} = \frac{p/\text{pgcd}(p,q)}{q/\text{pgcd}(p,q)}$.
Cette forme normalisée permet de tester l'égalité de deux rationnels très simplement. Si deux rationnels sont représentés de manière normalisée par $\frac{a}{b}$ et $\frac{c}{d}$, tester l'égalité requiert de comparer les nombres a et c d'une part et les nombres b et d d'autre part.

RAPPEL 2

L'écriture scientifique d'un nombre réel dans une base $\beta > 1$ est l'écriture sous la forme $m \times \beta^e$ avec $e \in \mathbf{Z}$ et $1 \leq m < \beta$. Le nombre m est appelé la *mantisse*.

B - Nombres dyadiques

DÉFINITION 1

On définit l'ensemble des nombres dyadiques $\mathbf{Y} = \left\{ \frac{a}{2^b} \mid (a, b) \in \mathbf{Z} \times \mathbf{N} \right\}$.

PROPRIÉTÉ 1

On a :

- $\mathbf{N} \subset \mathbf{Y}$ (rappel : $\forall a \in \mathbf{N}, \left(\frac{a}{2^0} \right) \in \mathbf{Y}$)
- $\mathbf{Y} \subset \mathbf{Q}$
- mais $\mathbf{Q} \not\subset \mathbf{Y}$

Les nombres dyadiques sont les réels qui s'écrivent avec un nombre fini de chiffres après la virgule en base 2.

C - Notation scientifique en base 2

RAPPEL 3

De la même manière, l'écriture scientifique d'un nombre en base 2 aura la forme $m \times 2^e$ avec $e \in \mathbf{Z}$ et $1 \leq m < 2$.

Puisque dans l'écriture scientifique la position de la virgule (du point !) est connue, ce point n'est pas représenté. Sur 4 bits, la mantisse représentant 1.5 sera codée $(1100)_2$, le bit de poids fort étant écrit à gauche (comme d'habitude).

EXERCICE 1

Déterminer les réels représentables en écriture scientifique en base 2 si la mantisse est codée sur 3 bits et pour des valeurs d'exposant compris entre -2 et 3.

Les nombres ne sont donc pas répartis uniformément!!! Cela vient du fait que le dernier bit disponible (ici le troisième), n'a pas la même valeur en fonction de l'exposant.

Remarque Si on ajoute un bit de signe, on peut obtenir exactement tous les opposés des nombres positifs.

D - Nombres flottants en machine

Les nombres flottants sont usuellement représentés en machine en suivant la norme IEEE-754, introduite en 1985. Voici la représentation utilisée avec la manière dont sont utilisés les bits d'un type flottant pour respecter la représentation.

$c =$	s	code de l'exposant : c_e	code de la mantisse : c_m
1 bit	k bits		$p - 1$ bits

Dans la représentation ci-dessus, s vaut 1 si le nombre à représenter est négatif, 0 sinon.

Le code c_m de la mantisse utilisé est en fait la représentation de la partie fractionnaire de la mantisse m en binaire, c'est-à-dire sans le chiffre à gauche du point dans l'écriture scientifique en base 2. Le premier 1 est donc codé implicitement : $m = (1, c_m)_2$. Ainsi, si c_m est codé sur $p - 1$ bits alors le nombre sera représenté avec p bits de précision.

Le code de l'exposant c_e est une représentation sur k bits de la valeur de l'exposant e avec la signification suivante :

$$\begin{cases} (c_e)_2 = 0 \text{ et } (c_m)_2 = 0 & \text{le nombre représenté est zéro} \\ (c_e)_2 = 2^k - 1 \text{ et } (c_m)_2 = 0 & \text{le nombre représenté est } \pm \text{Inf} \\ (c_e)_2 = 2^k - 1 \text{ et } (c_m)_2 \neq 0 & \text{le nombre représenté est NaN} \\ 1 \leq (c_e)_2 \leq 2^k - 2 & e = (c_e)_2 - (\underbrace{2^{k-1} - 1}_{\text{biais}}) \end{cases}$$

Si le nombre représenté est « normal », il vaut alors $(-1)^s \times m \times 2^e$ avec m et e obtenus comme expliqué ci-dessus.

La norme définit la taille des différentes parties des nombres notamment pour les deux formats ci-dessous :

simple precision sur 32 bits, avec 8 bits pour le code de l'exposant, 23 bits pour le code de la mantisse

double precision sur 64 bits, avec 11 bits pour le code de l'exposant, 52 bits pour le code de la mantisse

Les flottants de type `float` en C respectent le format **simple precision**. Les flottants de type `double` en C, `float` en OCaml (et `float` en Python) respectent le format **double precision**.

EXERCICE 2

Déterminer le plus petit nombre flottant normalisé représentable dans le format simple.

EXERCICE 3

Déterminer le plus grand nombre flottant normalisé représentable dans le format simple.

Les nombres flottants positifs normalisés représentables le format **simple** sont compris entre environ 10^{-38} et 10^{38} environ. Les nombres positifs représentables avec le format **double** sont compris entre environ 10^{-308} et 10^{308} .

IMPORTANT 1

Les flottants sont une représentation **approchée** des réels.

Les nombres qui ne sont pas des nombres dyadiques n'ont pas de représentation positionnelle finie en base 2 : ils seront approchées. Un nombre qui aurait une représentation finie mais trop longue pour le format utilisé serait également approché.

On parle d'*overflow* quand un nombre est trop grand pour être représenté, il y a alors dépassement et la représentation binaire utilisée correspond à celle de l'infini. On parle d'*underflow* pour les nombres trop petits : dans ce cas ils peuvent être représentés de manière non normalisée.

DÉFINITION 2

La **précision machine**, notée ϵ , est définie comme la quantité telle que $x = 1 + \epsilon$ est le plus petit flottant représentable strictement supérieur à 1.

Avec les représentations apportées par la norme IEE-754, le ϵ machine vaut

$$— \epsilon_{\text{simple}} = 2^{-23} \approx 1.2 \times 10^{-7}$$

$$— \epsilon_{\text{double}} = 2^{-52} \approx 2 \times 10^{-16}$$

La norme IEEE-754 définit 4 modes d'arrondi : au plus proche, vers 0, vers l'infini positif et vers l'infini négatif. Lorsqu'on enchaîne les calculs, les erreurs d'arrondis s'ajoutent les unes aux autres. De plus, même une opération s'effectuant sur deux nombres représentables de manière exacte en machine, peut conduire à un résultat faux !

III_ Types et opérations

RAPPEL 4

En C, il existe des types pour les entiers signés (`int`) et non signés (`unsigned int`). Les variantes permettant de manipuler des entiers non signés dont la taille est fixée sont `uint8_t`, `uint16_t`, `uint32_t` ou `uint64_t`, sur 8, 16, 32 ou 64 bits (`int8_t`, `int16_t`, `int32_t`, `int64_t` pour les signés).

Opérations sur des entiers

QUESTION 1

Quel est le nombre de bits nécessaire pour représenter l'addition de deux nombres entiers non signés représentés sur n et m bits ?

QUESTION 2

Quel est le nombre de bits nécessaire pour représenter la multiplication de deux nombres entiers non signés représentés sur n et m bits ?

Compléments

Nous avons vu qu'en C, il existe des opérateurs binaires utilisables sur des types d'entiers non signés : `|`, `&`, `^`.

L'opérateur `|` effectue un OU bit-à-bit. L'opérateur `&` effectue un ET bit-à-bit. L'opérateur `^` effectue un OU EXCLUSIF bit-à-bit.

Il est également possible d'effectuer des décalages des bits vers la gauche (ou vers la droite). Décaler de p bits vers la gauche revient à ajouter un nombre p de 0 à droite de la représentation binaire du nombre, c'est-à-dire à multiplier le nombre initial par 2^p .

En C, cet opérateur se note `<<`. Par exemple `1 << 2` vaudra $(100)_2$ soit 1×2^2 .

Pour un décalage à droite, utiliser `>>`. Par exemple, `111 >> 1` vaudra $(11)_2$ soit $[7/2]...$

RÈGLE 1

La valeur à droite de l'opérateur `<<` doit être inférieure à la précision du type utilisé !

Noms des opérateurs binaires en OCaml :

`|` s'écrit `lor`
`&` s'écrit `land`
`^` s'écrit `lxor`
`<<` s'écrit `lsl`
`>>` s'écrit `lsr`

Opérateur unaire : permet la négation d'un mot.

Cet opérateur s'écrit `~` en C, `lnot` en OCaml.

1 Introduction à l'organisation mémoire des architectures de calcul

Présentation générale	Un ordinateur est essentiellement composé de 2 type distincts de matériel : d'une part les unités de calcul et d'autre part les unités de stockage .
Unités de calcul	Fournir des résultats aux calculs élémentaires effectués : opérations arithmétiques et logiques ; fonctions élémentaires en mathématiques,
Unités de stockage	Permettre aux applications de mémoriser des données : données en entrées ou produites en cours et en fin de calcul.

1.1 Hiérarchie mémoire

Histoire	Ordinateur composé d'unités arithmétique et logique pouvant lire et écrire des données d'une mémoire principale .
Architecture moderne	<p>Une hiérarchie de mémoire</p> <ul style="list-style-type: none"> — des mémoires rapides voir très rapides mais coûteuses (en terme d'intégration). Proches des unités arithmétiques et logiques : registre du processeur ou mémoires caches. — une mémoire principale, appelée DRAM, de grande capacité, — des mémoires secondaires : mémoire non volatile, mémoire flash, disques, bandes magnétiques, ...

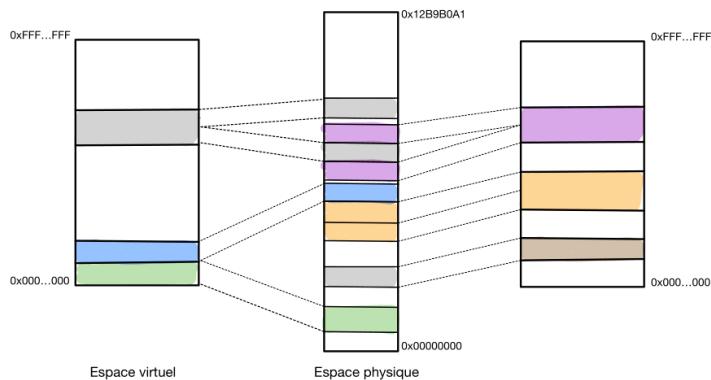
Type de mémoire	Temps d'accès indicatif
Registre	
Cache L1	
Cache L2	
Cache L3	
DRAM	
SSD	
HDD	

Utilisation et gestion	<p>registres gérés par le compilateur en fonction du programme à évaluer,</p> <p>mémoires caches sauvegardes des dernières données accédées par un programme,</p> <p>mémoire principale stocke les variables du programme,</p> <p>mémoires secondaires stocker les entrées et sorties, ou les données qui doivent être conservées.</p>
-------------------------------	--

1.2 Gestion de la mémoire par le système d'exploitation

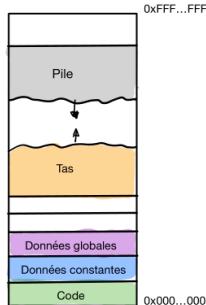
Utilisation par un programme	Le programme peut accéder aux données grâce à leurs adresses indiquant leurs positions dans la mémoire de l'ordinateur. Le système d'exploitation arbitre l'accès à la ressource « mémoire ».
Espace virtuel	Le système donne à chaque programme qui s'exécute, appelé processus , un espace virtuel de mémoire ou mémoire virtuelle . Cet espace virtuel de mémoire est propre à chaque processus et commence classiquement à l'adresse 0x000...000 (en hexadécimal) et s'arrête à 0xFFFF...FFF (que des bits à 1).

Illustration du mécanisme de projection de l'espace virtuel de mémoire de 2 processus vers une mémoire. Les zones colorées représentent les zones allouées dans l'espace virtuel et associées à des zones de la mémoire physique.



1.3 Organisation mémoire des processus

Illustration de l'organisation des différents types de mémoire d'un processus une fois créé par le système d'exploitation.



Ces quatre zones mémoires permettent de stocker différents type de données :

zone de code code exécutable du programme obtenu par compilation,

zone des données constantes données uniquement lues et jamais écrites,

zone des données globales deux sous parties : zones des données globales non initialisées, et celles initialisées,

tas données qui seront allouées dynamiquement,

pile manière très dynamique, données déclarées dans les fonctions et nécessaires à l'évaluation des expressions du programme.

2 Variables

2.1 Déclaration et définition des variables

Déclaration Donner le nom ou l'identifiant d'une variable

Définition Réserver ou allouer la zone mémoire correspondante.

Exemple `int a;` est la déclaration et la définition d'une variable de nom `a`.

2.2 Portée d'une variable

Règle La déclaration d'une variable doit précéder son utilisation.

Portée Le nom et la zone mémoire d'une variable ne sont valides qu'après l'instruction de déclaration, et jusqu'à la fin du **bloc d'instructions** qui englobe la déclaration.

2.3 Durée de vie d'une variable

La durée de vie d'une variable est associée à la durée de sa portée dans l'exécution du programme. On parle généralement de variable :

globale variable définie en dehors de toute fonction, connue du point de sa déclaration jusqu'à la fin du fichier compilé. Allouée dans la zone des données globales du processus,

automatique variable définie dans les fonctions ou les blocs d'instructions et n'est visible que dans son propre bloc d'instructions où elle est déclarée. Allouée dans la pile d'exécution.

manuel variable allouée dynamiquement dans le tas par le programme, et libérée par le programme.

semi-manuel variable allouée dynamiquement dans la pile par le programme (`alloca`) et libérée automatiquement par le programme lorsque le **bloc d'activation** est libéré.

3 Pile d'exécution

Vocabulaire Les variables locales (et certaines valeurs temporaires) sont stockées dans la **pile d'exécution** du processus. Le compilateur génère des instructions pour empiler (pousser dans la pile) des valeurs et des instructions pour dépiler (récupérer de la pile). La valeur dépiler sera toujours la dernière valeur empilée ! Cette zone mémoire dans la pile d'exécution fait partie que l'on appelle le **bloc d'activation** d'un appel de fonction.

3.1 Réalisation d'un appel de fonction

Considérons une fonction `f` qui appelle une fonction `g`.

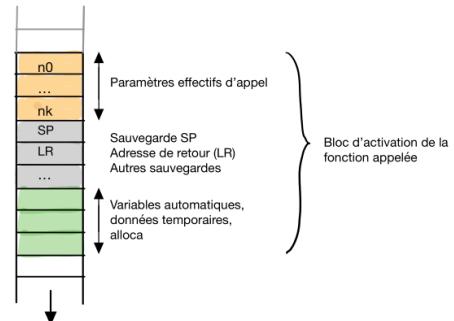
1. l'appelant (`f`) pousse sur la pile d'exécution n_0, n_1, \dots, n_k , les paramètres effectifs (les paramètres d'appel) pour `g`,
2. l'appelant sauvegarde des informations (typiquement le sommet de pile),
3. `f` appelle la fonction `g`,
4. l'appelé (`g`) peut sauvegarder d'autres informations (instruction à exécuter en retour de la fonction `g` générée lors de l'étape 3 précédente),
5. `g` alloue sur la pile un espace mémoire pour stocker ses variables locales et les données temporaires (connu à la compilation).

Pour retourner à l'appelant, `g` va restaurer les informations sauvegardées à l'étape 4 avant d'exécuter l'instruction suivante à l'appel à `g` dans le code de l'appelant `f`.

Bloc d'activation ou frame zone mémoire sur la pile d'exécution nécessaire à l'exécution de la fonction :

- déclaration et définition des variables automatiques,
- allocation des variables temporaires,
- allocation dynamique sur la pile (fonction `alloca`).

État de la pile pour un appel à la fonction `g`.



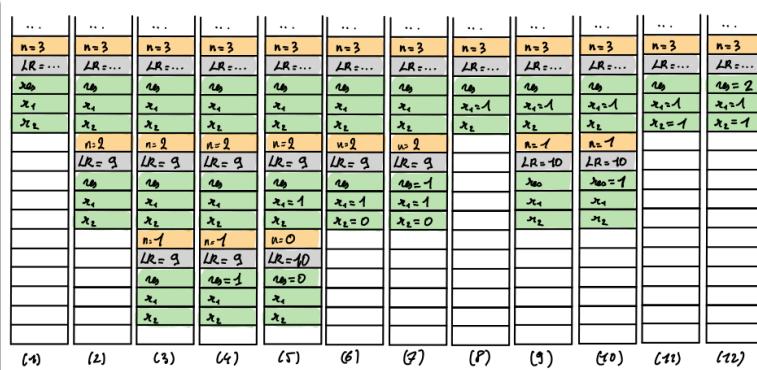
3.2 Illustration

Exemple projeté d'un code comportant deux fonctions `f`, illustration des blocs d'activation des fonctions.

3.3 Gestion des appels récursifs

```

1 int fibo(int n)
2 {
3     int res;
4     if (n<2) res = n;
5     else
6     {
7         int r1, r2;
8         r1 = fibo(n-1);
9         r2 = fibo(n-2);
10        res = r1+r2;
11    }
12    return res;
13 }
```



3.4 Comment est gérée la pile d'exécution ?

Gestion de la pile Le processus reçoit à sa création l'information d'une zone mémoire (un tableau) qui servira à stocker les données de la pile d'exécution. Le lanceur de processus initialise la pile lors de l'appel à la première fonction du processus.
La gestion de la pile revient à manipuler un pointeur, le pointeur **SP** pour **Stack Pointer**, qui pointe sur la prochaine adresse qui sera retournée lors de l'empilement d'une donnée.

III - Pile d'exécution

5 - Risque de l'utilisation d'une fonction récursive

⚠️ Attention

L'exécution d'une fonction récursive peut entraîner un **débordement de la pile d'exécution** (*stack overflow*).

6 - Récursivité terminale

☰ Exemple

Exemple de fonction somme récursive :

```
let rec somme n =
  if n = 0 then 0
  else n * somme (n-1)
```

Même fonction mais en "**récursif terminal**" :

```
let somme_rt n =
  let rec aux n s =
    if n = 0 then s
    else aux (n-1) (s+n)
  in aux n 0
```

✍ Définition

Une fonction est **récursive terminale** si le résultat renvoyé par un (potentiel) appel récursif n'est pas utilisé dans une opération.

⌚ Propriété du compilateur Ocaml

Si une fonction est récursive terminale, le compilateur Ocaml la traduit de manière itérative.

Chapitre 9 - Structures de données

- I - Les listes chaînées
 - A - Présentation
 - B - Représentation
 - C - Opérations
 - D - Algorithmes
 - E - Exemples d'implémentation
- II - Structures de données
 - A - Définitions
 - B - Type abstrait tableau
 - C - Structure abstraite d'ensembles
- III - Structures de données séquentielles
 - A - Structure de pile
 - B - Structure de file
 - C - Exemple d'algorithmes
 - D - File à double entrée (dequeue)
 - E - File à priorité
- IV - Implémentations classiques
 - A - Piles et files dans un tableau
 - B - File avec deux piles
 - C - Tableaux redimensionnables
 - 1 - Spécifications
 - 2 - Implémentation
 - 3 - Insérer un élément
 - 4 - Calcul de la complexité amortie pour n opérations

I - Les listes chaînées

A - Présentation

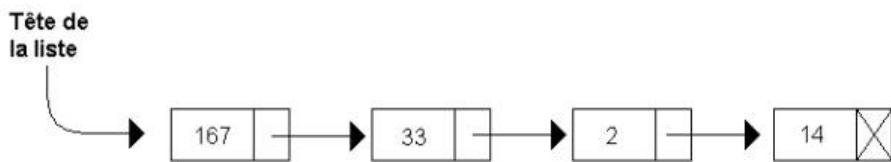
Définition

Une **liste chaînée** est une structure de données dans laquelle les éléments sont accessibles de manière linéaire.

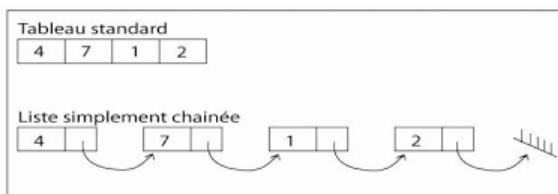
B - Représentation

Définition

Une **cellule** est une paire contenant un élément et l'adresse de la cellule suivante.



La valeur **Nil** représente "nulle part", elle est utilisée dans la dernière case de la liste chaînée.



C - Opérations

Remarque

Les opérations sont écrites en pseudo-code, T représente la tête de la liste et Nil la queue de la liste.

Algorithm Créer une liste vide

T ← Nil

Algorithm Insérer un élément en tête

Allouer(AC)
Contenu(AC).Succ ← T
T ← AC

Algorithm Supprimer l'élément en tête

AC ← T
T ← Contenu(AC).Succ
Libérer(AC)

D - Algorithmes

Algorithm Compter le nombre d'éléments positifs d'une liste chaînée

```
AC ← T
N ← 0
while AC != Nil do
    if Contenu(AC).E >= 0 then
        | N ← N + 1
    AC ← Contenu(AC).Succ
```

Algorithm Construire une liste contenant les entiers de 1 à n, par ajout en queue

```
T ← NIL
Q ← T
for i=1 to n do do
    | Allouer(AC)
    | Contenu(AC).E ← i
    | Contenu(AC).Succ ← Nil
    | Contenu(Q).Succ ← AC
    | Q ← Contenu(Q).Succ
T ← Contenu(T).Succ
return T
```

Algorithm Construire une liste contenant les entiers de 1 à n, par ajout en tête

```
T ← NIL
while Contenu(T).E != 1 do
    | Allouer(AC)
    | Contenu(AC).E ← n
    | Contenu(AC).Succ ← T
    | n ← n - 1
    | T ← AC
return T
```

Algorithm Écrire une fonction est croissante (T)

```
Q ← T
while Contenu(Q).Succ != Nil do
    | if Contenu(Contenu(Q).Succ).E < Contenu(Q).E then
        | | return false
    | Q ← Contenu(Q).Succ
return true
```

E - Exemples d'implémentation

En Ocaml :

```
type 'a list =
| Nil
| Cons of 'a * 'a list
let T = Cons(1, Cons(2, Nil))
```

En C :

```
struct Cellule {
    int element;
    struct Cellule *succ;
};
typedef struct Cellule Cellule_t;
```

Définition des algorithmes vus précédemment en C :

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int valeur;
    struct node_t *succ;
};
typedef struct node node_t;

int nb_positifs(node_t *t)
{
    int n = 0;
    node_t *temp = t;
    while (temp != NULL)
    {
        if (temp->valeur >= 0)
        {
            n++;
        }
        temp = temp->succ;
    }
    return n;
}
```

```

node_t *creer_liste_queue(int n)
{
    node_t *head = malloc(sizeof(node_t));
    head->valeur = 1;
    head->succ = NULL;
    node_t *queue = head;
    for (int i = 2; i <= n; i++)
    {
        node_t *temp = malloc(sizeof(node_t));
        temp->valeur = i;
        temp->succ = NULL;

        queue->succ = temp;
        queue = queue->succ;
    }
    return head;
}

node_t *creer_liste_tete(int n)
{
    node_t *head = NULL;
    while (n != 0)
    {
        node_t *temp = malloc(sizeof(node_t));
        temp->valeur = n;
        temp->succ = head;

        head = temp;
        n--;
    }
    return head;
}

int est_croissante(node_t *head)
{
    node_t *queue = head;
    while (queue->succ != NULL)
    {
        node_t *prochain = queue->succ;
        if (prochain->valeur < queue->valeur)
        {
            return 0;
        }
        queue = queue->succ;
    }
    return 1;
}

void afficher_liste(node_t *head)
{
    node_t *temp = head;
    while (temp != NULL)
    {
        printf("%d - ", temp->valeur);
        temp = temp->succ;
    }
}

void liberer_liste(node_t *head)
{
    node_t *temp = head->succ;
    while (temp != NULL)
    {
        free(head);
        head = temp;
        temp = temp->succ;
    }
    free(head);
}

```

Definition des algorithmes vus précédemment en OCaml :

```

type 'a cellule =
| Nil
| Cons of 'a * 'a cellule

let T = Cons(1, Cons(2, Nil))

let nb_positif (c : int cellule) : int =
  let rec aux cell acc = match cell with
  | Nil -> acc
  | Cons (v, succ) -> if v >= 0 then aux succ (acc+1) else aux succ acc
  in aux c 0

```

```

let creer_liste_tete n =
  let rec aux a c = if a=1 then Cons(a,c) else aux (a-1) (Cons(a,c))
  in aux n Nil ;;

let creer_liste_queue (n : int) : int cellule =
  let rec aux count = if count = n then Cons(n, Nil) else Cons(count, aux (count + 1))
  in aux 1

let rec est_croissante (c : int cellule) : bool = match c with
| Nil -> true
| Cons (v1, succ) -> match succ with
| Nil -> true
| Cons (v2, _) -> (v1 <= v2) && est_croissante succ

```

II - Structures de données

A - Définitions

Définitions

La **spécification d'une structure de données** est la définition formelle de son comportement.

Une **structure de données abstraites** est un ensemble d'éléments muni d'opérations agissant sur ses éléments.

La **spécification complète** est composée de :

- la *définition* des opérations avec la signature
- des *axiomes* qui permettent de définir le comportement
- des *préconditions* si les opérations sont partiellement définies

B - Type abstrait tableau

Définition : tableau

tableau(T, n)
T : le type des éléments du tableau
n : un entier représentant le nombre d'éléments du tableau

Opérations

fonction	type	documentation	préconditions
create	(n : int) -> (x : T) -> (t : tableau(T, n))	crée un tableau de taille n et initialise les éléments à la valeur de x	
get	(t : tableau) -> (i : int) -> (x : T)	renvoie x l'élément à la position i dans la tableau t	i doit être un indice valide
set	(t : tableau) -> (i : int) -> (v : T) -> (t' : tableau)	met à jour la valeur de l'élément à l'indice i du tableau t en lui assignant la valeur v	i doit être un indice valide

Axiome

get(set(t, i, x), i) = x

C - Structure abstraite d'ensembles

Définition : ensemble

ensemble(T)
T : le type des éléments de l'ensemble

Opérations

fonction	type	documentation	préconditions
cardinal	(E : ensemble) -> (n : int)	$n = Card(E)$	
add	(x : T) -> (E : ensemble) -> (E' : ensemble)	$E' = \{x\} \cup E$	
member	(E : ensemble) -> (x : T) -> bool	vrai si $x \in E$	
remove	(E : ensemble) -> (x : T) -> (E' : ensemble)	$E' = E \setminus \{x\}$	l'ensemble n'est pas vide
iter	(T : unit) -> (E : ensemble) -> unit		

III - Structures de données séquentielles

A - Structure de pile

Principe

Le dernier objet ajouté est le premier à être retiré.
C'est la principe **LIFO** (Last In First Out).

Spécifications fonctionnelles

opérations	type	comportement
pile_vide	'a pile	renvoie une pile vide
est_vide	'a pile -> bool	renvoie vrai si la pile est vide
empiler	'a -> 'a pile -> 'a pile	renvoie une pile contenant le nouvel élément
depiler	'a pile -> ('a * 'a pile)	renvoie l'élément du sommet de la pile et le reste de la pile

Préconditions

On peut dépiler seulement si la pile n'est pas vide

B - Structure de file

Principe

Le premier entré est le premier sorti.
C'est la principe **FIFO** (First In First Out).

Spécifications fonctionnelles

opérations	type	comportement
file_vide	'a file	renvoie une file vide
est_vide	'a file -> bool	vrai si la file est vide
enfiler	'a -> 'a file -> 'a file	fait entrer un élément dans la file
defiler	'a file -> ('a * 'a file)	renvoie l'élément le plus proche de la sortie et la file sans cet élément

C - Exemple d'algorithmes

Algorithm Inverser l'ordre des éléments d'une file en utilisant une pile

```
P ← PileVide()
while not EstVide(F) do
| e, F ← Défiler(F)
| P ← Empiler(e, P)
while not EstVide(P) do
| e, P ← Dépiler(P)
| F ← Enfiler(e, F)
return F
```

D - File à double entrée (dequeue)

Spécifications fonctionnelles

opérations	type	comportement
dequeue_empty	'a dequeue	renvoie une défileuse vide
is_empty	'a dequeue -> bool	vrai si la défileuse est vide
push_left	'a -> 'a dequeue -> 'a dequeue	insérer à gauche un élément
push_right	'a dequeue -> 'a -> 'a dequeue	insérer à droite un élément
pop_left	'a dequeue -> ('a * 'a dequeue)	renvoie un couple élément de gauche et file sans cet élément
pop_right	'a dequeue -> ('a dequeue * 'a)	renvoie un couple élément de droite et file sans cet élément

E - File à priorité

Définition

Une **file de priorité** est une structure de données qui permet de gérer un ensemble d'élément dont chacun a une valeur associée appelée **clé**. La **clé** est un élément d'un **ensemble totalement ordonné**. Un élément est généralement une paire (clé, valeur).

☰ Spécifications fonctionnelles

opérations	type	comportement
creer_file_priorite	('a * 'b) fp	renvoie une file à priorité vide
insérer	('a * 'b) -> ('a * 'b) fp -> ('a * 'b) fp	insère un nouvel élément dans la file à priorité
extraire_max	('a * 'b) fp -> (('a * 'b) * ('a * 'b) fp)	renvoie l'élément de priorité max et l'enlève de la file à priorité
maximum	('a * 'b) fp -> ('a * 'b)	renvoie la valeur de l'élément de priorité max sans modifier la file
augmenter_cle	('a * 'b) fp -> 'a -> 'a -> ('a * 'b) fp	modifie (augmente) la valeur de priorité d'un élément

IV - Implémentations classiques

A - Piles et files dans un tableau

Algorithm Pile dans un tableau

```

function PILEVIDE(P)
| return P.sommet = 0
function CREERPILE(n)
| P.n ← n
| P.sommet ← 0
| Allouer(n)
| return P
function EMPILER(P, e)
| if P.sommet = P.n then
| | Erreur
| | P.sommet ← P.sommet + 1
| | P[P.sommet] ← e
function DEPILER(P)
| if PileVide(P) then
| | Erreur
| | tmp ← P[P.sommet]
| | P.sommet ← P.sommet - 1
| return tmp

```

Algorithm File dans un tableau

```

function ENFILER(F, x)
| if FilePleine(F) then
| | Erreur
| | F.fileVide ← false
| | F.tete ← (F.tete + 1) mod (F.n + 1)
| | F[F.queue] ← x
| if F.queue + 1 = F.tete then
| | F.estPlein ← true
function DEFILER(P)
| if EstVide(F) then
| | Erreur
| | E ← F[F.tete]
| | F.tete ← F.tete + 1
| if F.queue + 1 = tete then
| | F.fileVide ← true
| if F.tete = F.n + 1 then
| | F.tete ← 1
| return E

```

B - File avec deux piles

Algorithm File avec deux piles

```

F ← (P1, P2)
function ENFILER(F, e)
| Empiler(P1, e)
function FILEVIDE(F)
| return EstVide(P1) and EstVide(P2)
function DEFILER(P)
| if EstVide(F) then
| | Erreur
| if PileVide(P2) then
| | while not PileVide(P1) do
| | | Empiler(P2, Depiler(P1))
| return Depiler(P2)

```

Remarque

Le coût total d'une série de n opérations est $O(n)$.

Définition

On définit la **complexité amortie** comme le coût moyen pour une série de plusieurs opérations.

Ici la complexité amortie de l'opération **defiler** est $O(1)$

C - Tableaux redimensionnables

1 - Spécifications

→ voir exemple dans le sujet de DS 4

2 - Implémentation

```
typedef struct tab_s {
    int capacite,
    int num,
    int *donnees,
} tableau;
```

3 - Insérer un élément

```
void push_back(tableau *t, int x)
{
    if (t->num == 0)
    {
        t->donnees = (int *)malloc(sizeof(int));
        if (t->donnees == NULL)
        {
            exit(EXIT_FAILURE);
        }
        t->capacite = 1;
        t->num = 1;
        t->donnees[0] = x;
    }
    else
    {
        if (t->num < t->capacite)
        {
            t->donnees[t->num] = x;
            t->num++;
        }
        else
        {
            int *nouveau = (int *)malloc(sizeof(int) * t->capacite * 2);
            if (nouveau == NULL)
            {
                exit(EXIT_FAILURE);
            }
            for (int i = 0; i < t->capacite; i++)
            {
                nouveau[i] = t->donnees[i];
            }
            free(t->donnees);
            t->donnees = nouveau;
            t->capacite = t->capacite * 2;
            t->donnees[t->num] = x;
            t->num++;
        }
    }
}
```

4 - Calcul de la complexité amortie pour n opérations

Complexité amortie pour n opérations :

$$\begin{cases} \text{si } i = 2^k \rightarrow i \\ \text{sinon } 1 \end{cases}$$

$$a \sum_{i=0}^n i + b \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k = 2n - 1$$

$$T(n) = an + b(2n - 1) \leq an + 2bn = n(a + 2b)$$

$$T(n) = T(\lceil \frac{n}{2} \rceil) + 1 \text{ est } \Theta(\log n)$$

Chapitre 10 - Bases de données relationnelles

- I – Introduction
- II – Bases de données relationnelles
- III- Interrogation avec le langage SQL
 - A – Projection
 - 1) Présentation
 - 2) Syntaxe en SQL
 - 3) Gestion des doublons
 - B – Sélection
 - 1) Syntaxe
 - 2) Exemples
 - C – Tris
 - D – Opérateurs d'agrégation
 - E – Partition
 - 1) Syntaxe
- IV - Retour sur le modèle relationnel
 - Définitions
- V - SQL : Éléments avancés
 - A - Produit cartésien
 - 1) Présentation
 - 2) Syntax en SQL
 - B - Jointures
 - 1) Exemple
 - C - Opérateurs ensemblistes
 - 1) Les opérateurs ensemblistes
- VI - Modèle entité association
 - A - Définitions
 - B - Exemple

I – Introduction

Objet d'étude

On s'intéresse aux BD relationnelles

SGBDR = Système de Gestion de Base de Données Relationnelles

II – Bases de données relationnelles

Définitions

Une base de données relationnelles est un ensemble de tables ou de relations

Le **schéma relationnel** de LesVegetaux s'écrit :

LesVegetaux(idVeg, nomVeg, couleur, type, prix)

Une **valeur** est une donnée d'un attribut pour un enregistrement.

attribut : colonne

enregistrement : ligne

Les attributs dont le nom est souligné constituent l'identifiant de la relation
un identifiant est unique à un enregistrement dans une table

Les valeurs de l'identifiant (clé primaire de la relation) sont uniques

Un attribut a un type : c'est le **domaine de définition**

III- Interrogation avec le langage SQL

A – Projection

1) Présentation

A	X	Y
3	2	T
1	6	G
4	3	Z

On « projette » une relation sur un sous-ensemble d'attributs
projection de R dans (A,X) →

A	X
3	2
1	6
4	3

2) Syntaxe en SQL

```
SELECT A,X --le nom des attributs  
FROM R ; --nom de la relation
```

SQL

```
SELECT nomVeg  
FROM lesVegetaux ;
```

SQL

3) Gestion des doublons

```
SELECT DISTINCT A,X  
FROM R ;
```

SQL

B – Selection

1) Syntaxe

```
SELECT A  
FROM R  
WHERE condition ; --condition pour conserver un enregistrement
```

SQL

2) Exemples

```
SELECT nomVeg  
FROM LesVegetaux  
WHERE type = 'légume' AND couleur = 'orange' ;
```

SQL

```
SELECT nomVeg, prix * 1.05  
FROM lesVegetaux  
WHERE type = 'fruit' ;
```

SQL

C – Tris

```
SELECT A --liste d'attributs  
FROM R --nom de la relation  
[WHERE condition]  
ORDER BY B [ASC/DESC] ;
```

SQL

```
SELECT nomVeg  
FROM lesVegetaux  
ORDER BY prix ASC ;
```

SQL

```
SELECT  
FROM  
[WHERE condition]  
ORDER BY  
LIMIT a --nb d enregistrement renvoyé  
OFFSET num --nombre d enregistrement renvoyé ;
```

SQL

🔗 Syntaxe

LIMIT a : réduit le nombre d'enregistrement renvoyé au nombre de a (en partant du haut)
OFFSET num : décale le début des enregistrement sélectionner à num.

D – Opérateurs d'agrégation

```
SELECT count(nomVeg)  
FROM lesVegetaux  
WHERE type = 'fleur' ;
```

SQL

```
SELECT count(DISTINCT nomVeg)
FROM lesVegetaux
WHERE type = 'fleur' ;
```

Syntaxe

count() : compte le nombre
max() : la valeur maximale
min() : la valeur minimal
avg() : une moyenne
sum() : la somme

E – Partition

1) Syntaxe

```
SELECT
FROM
WHERE
GROUP BY A –nom d un attribut
HAVING
ORDER BY ;
```

Exemple

Type de végétaux qui ont au moins 2 végétaux enregistrés.

```
SELECT type
FROM lesVegetaux
GROUP BY type
HAVING count(idVeg) >= 2;
```

IV - Retour sur le modèle relationnel

Définitions

Définitions

Un **domaine** est une ensemble fini ou non de valeurs possibles.
L'ensemble des **n-uplet** est un produit cartésien de domaines.

On dit qu'on a une **relation** \mathcal{R} entre deux ensembles A et B si \mathcal{R} relation sur $A \times B$

Si on a $a \in A$ et $b \in B$, a et b sont en relation par \mathcal{R} si $(a, b) \in \mathcal{R}$, noté $a \mathcal{R} b$

On donne un nom aux différentes domaines de la relation, ce nom est appelé **attribut**.

La **clé primaire** d'une relation est un ensemble d'attributs qui détermine un n-uplet de manière unique.

Une **clé étrangère** est un ensemble d'attribut qui référence un ensemble d'attribut d'une autre relation.

[Définitions plus clairs \(cours de terminal\)](#)

V - SQL : Éléments avancés

Exemple de schéma relationnel

```
LesVegetaux(idVeg, nom, couleur, type)
LesProducteurs(idProd, nomBoutique, surface, dept)
LesPrix(#idProd, #idVeg, prix)
```

A - Produit cartésien

1) Présentation

R

A	B
1	'C'
2	'D'
3	'C'

S

A	C
3	'A'
4	'C'

$R \times S$

A	B	A	C
1	'C'	3	'A'
1	'C'	4	'C'
2	'D'	3	'A'
2	'D'	4	'C'
3	'C'	3	'A'
3	'C'	4	'C'

2) Syntax en SQL

```
SELECT *
FROM R, S;
```

SQL

B - Jointures

1) Exemple

Nom et prix des végétaux chez le producteur idProd = 1

```
SELECT V.nom, P.prix
FROM LesVegetaux AS V JOIN LesPrix AS P
ON V.idVeg = P.idVeg
WHERE P.idProd = 1;
```

SQL

Prix max par boutique

```
SELECT nomBoutique, max(prix)
FROM LesProducteurs AS P JOIN LesPrix
ON P.idProd = LesPrix.idProd
GROUP BY nomBoutique;
```

SQL

R

A	B
1	Castor
2	Loutre
3	Elan

S

A	C
3	bonjour

SQL

```
SELECT S.A, S.C, R.B
FROM R
JOIN S ON R.A = S.A;
```

S.A	S.C	R.B
3	bonjour	Elan

SQL

```
SELECT S.A, S.C, R.B
FROM R LEFT JOIN S ON S.A = R.A;
```

S.A	S.C	R.B
3	bonjour	Elan
NULL	NULL	Castor
NULL	NULL	Loutre

C - Opérateurs ensemblistes

1) Les opérateurs ensemblistes

UNION, INTERSECT, EXCEPT

SQL

s'appliquent sur des relations qui ont le même schéma relationnel

Nom des légumes de type légume qui ne sont pas de couleur rouge :

```
SELECT nom FROM LesVegetaux
WHERE type='legume'
EXCEPT
SELECT nom FROM LesVegetaux
WHERE couleur='rouge';
```

SQL

IN / NOT IN

SQL

```
SELECT nom AS n
FROM LesVegetaux
WHERE nom NOT IN
(SELECT nom FROM LesVegetaux WHERE couleur='rouge')
AND type='legume';
```

SQL

VI - Modèle entité association

A - Définitions

Définition

Une **entité** est la représentation d'un objet ayant une existence propre.

Exemples

- un végétal
- une boutique
- un département
- un prix
- un producteur

Remarque

Une entité peut posséder des propriétés, par exemple la couleur.

Définitions

Une **association** est un lien entre différentes entités.

La **cardinalité** d'un lien est représentée par un couple de valeurs indiquant le minimum et le maximum de fois qu'une entité apparaît dans une association.
La valeur min est généralement soit 0 soit 1. On écrit * lorsqu'on ne connaît pas la valeur max. La cardinalité est notée a..b avec a la valeur min et b la valeur max.
Notation : la cardinalité 0..* est notée *

Définitions

Le **type d'une association** est déterminé par les bornes maximums des deux côtés de l'association.

Le type de l'association indique comment traduire l'association dans le modèle relationnel.

1-* : association dite *fonctionnelle* ou *hiérarchique*, chaque élément de l'entité du côté * peut être lié à plusieurs éléments de l'entité du côté 1 (ex: livre-auteur)
1-1 : à chaque élément de l'entité A est lié exactement un élément de l'entité B (ex: proviseur-lycée)
- : chaque élément de l'entité A est lié à plusieurs éléments de l'entité B et inversement (ex: animal-maladie)

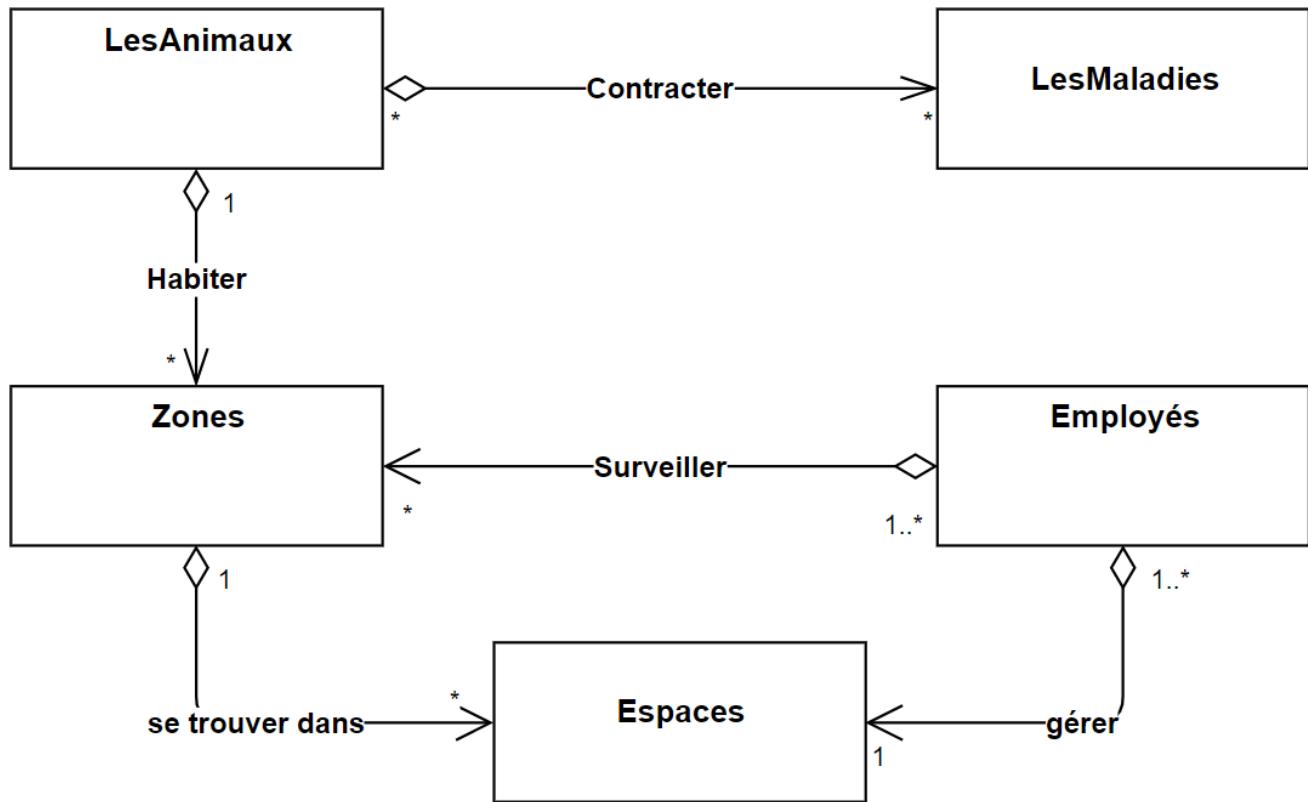
Chaque entité est représentée par une table. Pour chaque association :

1-* : ajouter une clé étrangère du côté 1 qui référence la clé primaire du côté *

- : ajouter une table d'association dont la clé est la paire de clés étrangères

Dans le cas des associations 1-1, on peut fusionner les entités

B - Exemple



Chapitre 11 - Traits impératifs en OCaml

- I - Les basiques
 - A - type unit
 - B - affichage
 - C - l'opérateur ;
 - D - blocs
- II - Boucles
 - A - Boucle for
 - B - Boucle while
- VI - Structures mutables
 - A - Mutabilité
- V - Cours plus structuré

I - Les basiques

A - type unit

Le type `unit` en OCaml est un type produit donc le constructeur est vide.

```
();;  
- : unit = ()  
let res = ();;  
val res : unit = ()
```

B - affichage

```
print_int;;  
(* int -> unit *)  
  
(* il existe aussi les fonction print_float, print_char, print_endline etc *)
```

Une autre manière de faire :

```
Printf.printf "ceci est un entier %d" 10;;
```

C - l'opérateur ;

L'opérateur `;` permet de séparer plusieurs expressions de type `unit`.

```
let x = e1 in e2; e3  
(* est équivalent à *)  
let x = e1 in (e2; e3)
```

Exemple:

```
print_int 10; print_newline ();;
```

D - blocs

```
if e then begin  
e1; e2;  
e3;  
end  
else f1
```

Sans le `begin ... end`, l'expression est équivalente à

```
(if e then e1); e2; e3; else f1
```

II - Boucles

A - Boucle for

Syntaxe :

```
for i = e1 to e2 do  
  e  
done
```

`i`, `e1` et `e2` sont de type `int`
La borne `e2` du `for` est inclusive.

On peut boucler à l'envers :

```
for i = e1 to e2 downto
  e
done
```

B - Boucle while

Syntaxe :

```
while c do
  e
done
```

`c` est de type `bool` et `e` de type `unit`

VI - Structures mutables

A - Mutabilité

Une structure de donnée est **mutable** si certains de ses "champs" peuvent être modifiés.

V - Cours plus structuré

 Lien

[cours structuré](#)

Chapitre 12 : Un peu d'ordre...

- I - Relation Binaires
 - A - Rappels
 - B - Relation fonctionnelle
 - C - Association fonctionnelle
- II - Relation d'ordre
 - A - Prédécesseurs, successeurs
 - B - Extrêmaux
 - C - Propriétés des extrêmaux
 - D - Ordre bien fondé
- III - Ordres sur des paires
 - A - Ordre lexicographique
 - B - Ordre produit
- IV - Applications

I - Relation Binaires

A - Rappels

⌚ Prérequis

Voir le cours de maths

B - Relation fonctionnelle

✍ Définition

Une **relation fonctionnelle** est une relation binaire décrivant le lien entre les entrées et les sorties d'une fonction. Une telle relation \mathcal{R} pour une fonction $f : A \rightarrow B$ contient donc les paires (a, b) telles que $f(a) = b$. Cet ensemble de paires est aussi appelé le graphe de la fonction f . La caractéristique principale d'une telle relation qui définit le concept de fonction est que la sortie $f(a)$ est uniquement déterminée par l'entrée a . Autrement dit, il ne peut pas y avoir deux images associées au même antécédent.

Une relation $\mathcal{R} \subseteq A \times B$ est fonctionnelle si pour tout $a \in A$, il existe au plus un $b \in B$ tel que $a \mathcal{R} b$. Autrement dit, quelque soient $a \in A$ et $b_1, b_2 \in B$ si $a \mathcal{R} b_1$ et $a \mathcal{R} b_2$ alors $b_1 = b_2$.

C - Association fonctionnelle

✍ Remarque

Une association entre deux relations A et B de type 1- **est une association** fonctionnelle* : un même élément de la relation A apparaît une seule fois dans l'association avec la relation B .

II - Relation d'ordre

A - Prédécesseurs, successeurs

⌚ Prérequis

- définition d'une relation d'ordre
- définition d'un ordre total

✍ Définition

Un **ordre strict** est une relation binaire homogène qui est à la fois transitive, antisymétrique et irréflexive.

Si \preceq est un ordre, l'ordre strict associé est la relation \prec définie par $a \prec b \Leftrightarrow a \preceq b$ et $a \neq b$. Si \prec est un ordre strict, l'ordre associé est la relation \preceq définie par $a \preceq b \Leftrightarrow a \prec b$ ou $a = b$.

✍ Définitions

Etant donné un élément e d'un ensemble ordonné (E, \prec) , on appelle :

- **prédécesseur de e** un élément $a \in E$ tel que $a \prec e$.
- **successeur de e** un élément $a \in E$ tel que $e \prec a$.

Etant donné un élément de e d'un ensemble ordonné (E, \prec) , on appelle :

- **prédécesseur immédiat de e** un élément $a \in E$ tel que $a \prec e$ et qu'il n'existe pas de $b \in E$ vérifiant $a \prec b \prec e$.
- **successeur immédiat de e** un élément $a \in E$ tel que $e \prec a$ et qu'il n'existe pas de $b \in E$ vérifiant $e \prec b \prec a$.

B - Extrémaux

Remarque

Pour "être le plus petit" deux possibilités :

- être le plus petit que tout les autres
- être tel qu'aucun autre élément n'est plus petit

Définitions

Soit (E, \leq) un ensemble ordonné

- e est **le plus petit élément** de E si pour tout $a \in E$, $e \leq a$.
- e est **un élément minimal** de E s'il existe pas de $a \in E$ tel que $a < e$ (ou si $\forall a \in E$ tel que $a \leq e$ alors $a = e$).

Soit (E, \leq) un ensemble ordonné

- e est **le plus grand élément** de E si pour tout $a \in E$, $a \leq e$.
- e est **un élément maximal** de E s'il n'existe pas de $a \in E$ tel que $e < a$ (ou si $\forall a \in E$ tel que $e \leq a$ alors $a = e$).

C - Propriétés des extrémaux

Propriétés de extremaux

Soit (E, \leq) un ensemble ordonné

1. le plus petit élément de E , s'il existe, est unique.
2. le plus petit élément de E est un élément minimal de E .
3. si l'ordre est total et si e est un élément minimal de E , alors e est également le plus petit élément de E .

D - Ordre bien fondé

Prérequis

- Définition du majorant et du minorant.
- Définition d'une borne supérieure et d'une borne inférieure.

Propriétés

Un **minorant** d'un ensemble A qui appartient à A est le plus petit élément de A .

Si e est **le plus petit élément** de l'ensemble ordonné (A, \leq) alors e est la borne inférieure de l'ensemble A .

Un ordre (E, \leq) sur un ensemble E est **bien fondé** si et seulement si tout partie non vide de E admet un élément minimal.

Propriétés d'un ordre bien fondé

Si un ordre \leq sur un ensemble E est **bien fondé** si et seulement s'il n'existe pas de suite infinie strictement décroissante pour \leq . Autrement dit, avec \prec l'ordre strict associé à \leq , il ne peut pas exister de suite (x_k) telle que $\forall k \in \mathbb{N}$ on ait $x_{k+1} \prec x_k$.

Preuve

⇒ Supposons que toute partie A non vide d'un ensemble E admette un élément minimal.

Supposons que (x_k) soit infinie décroissante dans E , on note A l'ensemble des valeurs de cette suite.

L'ensemble A est non vide il contient par exemple x_0 . Il admet donc un élément minimal x_k . Or $x_{k+1} < x_k$ avec $x_{k+1} \in A$ ce qui contredit la minimalité de A.

III - Ordres sur des pairs

A - Ordre lexicographique

Prérequis

- définition de l'ordre lexicographique

Propriété

Si (A, \leq_A) et (B, \leq_B) sont deux ordres bien fondés, alors l'ordre lexicographique \leq sur $A \times B$ est bien fondé.

Intérêt

L'ordre lexicographique permet de justifier la terminaison d'un algorithme dans lequel on a une hiérarchie entre les différentes variables susceptibles de décroître.

B - Ordre produit

Définition

On se donne deux ensembles ordonés (A, \preceq_A) et (B, \preceq_B) . L'**ordre produit** sur $A \times B$ est l'ordre \preceq tel que $(a_1, b_1) \preceq (a_2, b_2) \Leftrightarrow (a_1 \preceq_A a_2 \text{ et } b_1 \preceq_B b_2)$.

⇒ **L'ordre produit n'est pas un ordre total.**

Propriété

Si (A, \preceq_A) et (B, \preceq_B) sont deux ordres bien fondés, alors l'**ordre produit** \preceq sur $A \times B$ est bien fondé.

Intérêt

A utiliser pour prouver la terminaison d'un algorithme dans lequel plusieurs variables décroissent à tour de rôle.

IV - Applications

Suite d'Ackermann

On définit la suite d'Ackermann par

$\text{ack}(0, m) = m + 1$

$\text{ack}(n, 0) = \text{ack}(n-1, 1)$ si $n > 0$

$\text{ack}(n, m) = \text{ack}(n-1, \text{ack}(n, m-1))$ si $n > 0$ et si $m > 0$

utilisation de l'ordre lexicographique sur (n, m)

Application à un tri

Pour trier un tableau t de n entiers, on utilise le principe suivant
tant que le tableau n'est pas trié choisir deux indices i et j tel que

$i < j$

$t[i] > t[j]$

et échanger les deux valeurs.

Cet algorithme termine-t-il ?

Chapitre 13 : Les arbres

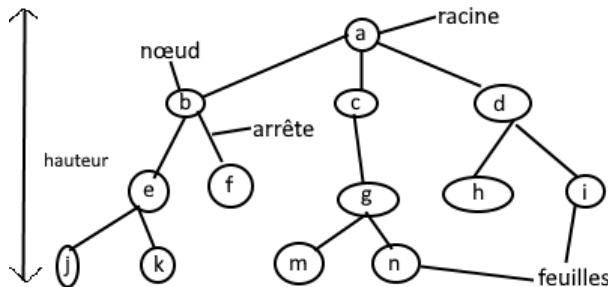
- I - Notion d'arbre
 - A - Introduction
 - B - Première définition
 - C - Représentations
 - 1) Avec des ensembles
 - 2) Indentation
 - 3) Linéaires parenthésées
 - D - Arborescences
 - E - Forêts
 - F - Terminologie des arbres
 - G - Implémentation en OCaml
 - 1) Première définition
 - 2) Deuxième définition
 - H - Implémentation C
- II - Objets inductifs
 - A - Définition informelle
 - B - Exemples
 - C - Principe d'induction
- III - Arbres binaires
 - A - Définition
 - B - Implémentation en OCaml
 - C - Implémentation en C
 - D - Propriété des arbres binaires
 - Définitions supplémentaires
 - E - Conversion d'un arbre en arbre binaire
 - F - Parcours d'arbres binaires
- IV - Arbres binaires de recherche
- V - Tas

I - Notion d'arbre

A - Introduction

Mise en situation

Considérons un ensemble fini A de points disposés de cette manière :



On peut définir une relation d'ordre \prec :

- a est le plus petit élément de l'ensemble A .
 $a \prec b \prec e \prec j, h$
 $a \prec b \prec f$
 $a \prec c \prec g \prec m, n$
- j, k, f, m, n, h des éléments maximaux de A .
- $x \prec y$ si il existe un chemin d'origine x et de destination y .

B - Première définition

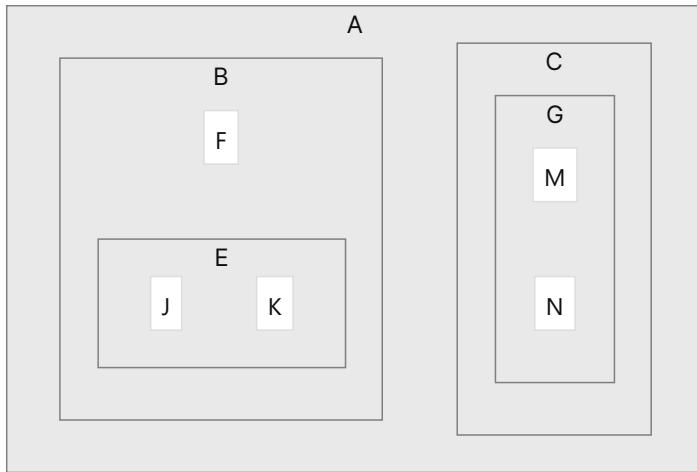
Définition

Soit un ensemble (A, \prec) , tel que A est **arborescent** :

- A admet un élément minimal → **la racine**
- lorsqu'on supprime la racine on obtient une réunion d'arbres qu'on appelle une **forêt**.

C - Représentations

1) Avec des ensembles



2) Indentation

```
a  
  b  
    e  
      j  
    f  
  c  
  ...
```

3) Linéaires parenthésées

Notation préfixée

Racine puis les sous-arbres :
(a (b (e (j) (k)) (f)) ...)

Notation postfixée

Sous-arbres puis racine :
(... ((j) (k) e) b) a)

D - Arborescences

Définitions

Soit (A, \prec) un ensemble ordonné où \prec est l'ordre strict associé à \preceq .

On dit que A est **arborescent** si on a :

- A est un ensemble fini
- A admet un plus petit élément
- Pour tout $a \in A$ l'ensemble des prédécesseurs de a est *totalement ordonné*.

Définition

Les éléments maximaux de A sont appelés les **feuilles**.

Définitions

Soit $a \in A$

- Si a n'est pas la racine de l'ensemble de A , on appelle **père** de a le *prédecesseur immédiat* de a .
- Si a n'est pas une feuille de A , on appelle **fils** de a les *successeurs immédiats* de a .

Proposition

Soit A, \prec un ensemble arborescent et $a \in A$

- Si a n'est pas une feuille de A alors il est le père de ses fils.
- Si a n'est pas la racine alors il est le fils de son père.

Preuve

Supposons que a n'est pas une feuille de A et soit b un fils de a .

Alors b est un successeur immédiat de a donc en particulier il n'existe pas de x tel que $a \prec x \prec b$ et donc a est un prédécesseur immédiat de b .

E - Forêts

Définition

Soit (F, \prec) un ensemble ordonné

F est une **forêt** si on a :

- F est fini
- Pour tout $a \in F$, l'ensemble des prédécesseurs de a est *totalement ordonné*

Théorème

Toute forêt est la réunion d'un nombre fini d'ensembles d'arborescence deux à deux disjoints.

F - Terminologie des arbres

Définitions

Racine : noeud sans prédécesseur.

Feuille : noeud sans successeur.

Nœuds internes : nœuds avec successeur.

Fils : les successeurs d'un nœud.

Père : le prédécesseur d'un nœud.

Arrête (branche) : le lien entre le fils et le père.

Hauteur : plus grande distance depuis la racine (arbre à un seul élément : hauteur = 0).

Profondeur (d'un nœud) : distance du nœud à la racine (on compte le nombre d'arrête).

Degré (arité) d'un nœud : nombre de branches qui partent d'un nœud.

Degré d'un arbre : n si tous les nœuds internes sont d'arité n.

Taille : nombre de nœuds.

On appelle **arbre étiqueté** tout quadruplet $(A, \prec, \varepsilon_N, \varepsilon_F)$ d'un arbre (A, \prec) , d'une application ε_N de l'ensemble des nœuds dans N et de l'application ε_F de l'ensemble des nœuds de F .

G - Implémentation en OCaml

1) Première définition

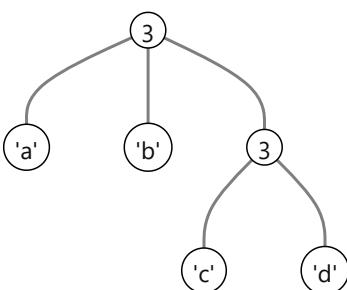
```
type 'a arbre = 'a * 'a arbre list
```

```
type 'a arbre =
| Feuille of 'a
| Nœud of ('a * 'a arbre list)
```

2) Deuxième définition

```
type ('f, 'n) arbre =
| Feuille of 'f | Nœud of 'n * ('f, 'n) foret
and ('f, 'n) foret = Arbres of ('f, 'n) arbre list
```

Exemple



```
let a = Nœud (3, Arbres ([Feuille 'a'; Feuille 'b'; Nœud (3, Arbres ([Feuille 'c'; Feuille 'd']))]));;
let f = Arbres ([a;a])
```

H - Implémentation C

```
typedef struct noeud {
    int element;
    struct noeud * fils;
    struct noeud * frere;
} arbre;
```

C

II - Objets inductifs

A - Définition informelle

✍ Définition inductive d'un objet

- un (ou plusieurs) objets de base
- une (ou plusieurs) règles de combinaison des objets

B - Exemples

☰ Les listes OCaml

{ une liste vide, notée [] -> cas de base
une règle : si e est un élément et l une liste, alors e::l est une liste qui contient l'élément e et dont la queue est l }

☰ Les entiers de Peano

{ le nombre appelé Zéro (0) est un entier naturel
si n est un entier naturel, alors Succ(n) est un entier naturel }

```
type nat =
| Zero
| Succ of nat

let rec addition n m = match n with
| Zero -> m
| Succ p -> addition p (Succ m)

let rec double n = match n with
| Zero -> Zero
| Succ p -> Succ (double p))
```

C - Principe d'induction

✍ Principe

Soit $X \subseteq E$ définie inductivement par (B, R) . Soit P un prédictat sur E .

1. Si $P(x)$ est vrai pour tout $x \in B$.
2. Si pour tout $r \in R$, pour tous $x_1, \dots, x_n \in X$. Si lorsque $P(x_1), \dots, P(x_n)$ sont vrais alors $P(r(x_1, \dots, x_n))$ est vrai de 1 et 2. $P(x)$ est vrai.

☰ Exemple

Tout arbre à n sommets possède $n - 1$ arrêtes.

Initialisation : Si l'arbre est réduit à 1 feuille, alors 0 arrêtes, la propriété est vérifiée.

Conservation : Soit A un arbre de racine a_0 et de sous-arbres A_1, \dots, A_p avec n sommets, chaque branche issue de a_0 ayant n_1, \dots, n_p sommets.

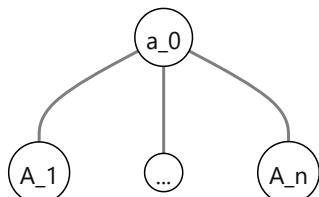
Le nombre de sommets n est donné par $n = 1 + n_1 + \dots + n_p$.

Par hypothèse d'induction, chaque sous-arbres a $n - 1$ arrêtes. Le nombre d'arrêtes est donc $n_1 - 1 + \dots + n_p - 1 + p = n_1 + \dots + n_p = n - 1$.

Conclusion : tous les arbres respectent la propriété.

✍ Définition inductive d'un arbre

- Cas de base : sommets singleton
- Un arbre est construit par le lien entre un nœud racine et d'autres arbres de la manière suivante :



Le a_0 devient le prédécesseur immédiat de tous les arbres A_1, \dots, A_n .

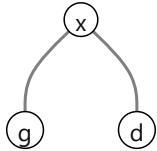
III - Arbres binaires

A - Définition

Définition

Un **arbre binaire** d'éléments de type a est défini de manière inductive par :

- un arbre vide noté `vide`
- un noeud représenté par un triplet (x, g, d) avec x un élément de type a , g un arbre binaire et d un arbre binaire.



Une feuille est représentée par le triplet $(x, Vide, Vide)$.
La hauteur de l'arbre vide est 0.

B - Implémentation en OCaml

```
type 'a arbre =
| Vide
| Noeud of ('a * 'a arbre * 'a arbre)

let rec taille a = match a with
| Vide -> 0
| Noeud (_, gauche, droite) -> 1 + taille gauche + taille droite

let rec feuille a = match a with
| Vide -> 0
| Noeud (_, Vide, Vide) -> 1
| Noeud (_, gauche, droite) -> feuille gauche + feuille droite

let noeuds_interne a = (taille a) - (feuille a)

let rec hauteur a = match a with
| Vide -> -1
| Noeud (_, gauche, droite) -> 1 + max (hauteur gauche) (hauteur droite)
```

C - Implémentation en C

```
typedef struct noeud {
    int element;
    struct noeud *gauche;
    struct noeud *droite;
} arbre_binaire;

// arbre vide
arbre_binaire *a = NULL;

// feuille
arbre_binaire a1 = { .element = 42, .gauche = NULL, .droite=NULL};

int taille(arbre_binaire *a)
{
    if (a == NULL)
    {
        return 0;
    }
    return 1 + taille(a->gauche) + taille(a->droite);
}
```

D - Propriété des arbres binaires

Exercices

Enoncé

Soit A un arbre binaire qui possède n noeuds alors il y a $n + 1$ branches vides.

Initialisation : Un arbre possédant 1 noeud possède 2 branches vides.

Conservation : Soit A un arbre binaire possédant une racine a_0 et 2 sous-arbres : A_g, A_d ayant respectivement n_g et n_d noeuds.

Alors par hypothèse d'induction, A_g possède $n_g + 1$ branches vides et A_d possède $n_d + 1$ racines. Donc A possède $n = 1 + n_g + n_d$ noeuds.

Ainsi, le nombre de branches vides de A est $n_g + 1 + n_d + 1 = n + 1$ branches vides.

Conclusion : Tout arbre possédant n noeuds possède $n + 1$ branches vides.

Enoncé

Soit A un arbre de hauteur h ayant n noeuds, tous de degré inférieur ou égal à 2. Montrer par induction structurelle que $h + 1 \leq n \leq 2^{h+1} - 1$.

Initialisation : Un arbre de hauteur 0 possède 1 noeud, donc $0 + 1 \leq 1 \leq 2^{0+1} - 1 \Leftrightarrow 1 \leq 1 \leq 1$.

Conservation : Soit A un arbre de hauteur h .

A_1 et A_2 sont de hauteur au plus $h - 1$. Donc le nombre de noeuds n_1 de A_1 est entre 0 et $2^h - 1$ de même pour pour A_2 .

$$n = 1 + n_1 + n_2$$

$$\text{Donc } h - 1 \leq n_1 + n_2 \leq 2^h - 1 + 2^h - 1$$

$$\Leftrightarrow h \leq n \leq 2^{h+1} - 1$$

Conclusion : A un arbre de hauteur h ayant n noeuds, tous de degré inférieur ou égal à 2. Montrer par induction structurelle que $h + 1 \leq n \leq 2^{h+1} - 1$.

Définitions supplémentaires

Définitions

Un arbre binaire est **strict** si l'arité est soit 0 soit 2.

Un arbre est **complet** si tous les niveaux sont remplis intégralement \Leftrightarrow toutes les feuilles sont à la même profondeur + l'arbre binaire est strict.

Un arbre **parfaitement équilibré** est un arbre pour lesquels tous les niveaux sont remplis sauf éventuellement le dernier.

Définition

Les nombres $C(n)$ définis par
$$\begin{cases} C(0) = 1 \\ C(n) = \sum_{i=0}^{n-1} C(i)C(n-1-i) \end{cases}$$
 sont appelés les **nombres de Catalan** ($C(n) = \frac{1}{n+1} \binom{2n}{n}$).

Proposition

Il y a $C(n)$ arbre binaires possédant n noeuds.

E - Conversion d'un arbre en arbre binaire

Définition

Conversion : application qui à un arbre de type '**a arbre**' associe un arbre binaire de type '**a arbtrebin**' dans lequel chaque noeud admet pour enfant gauche l'arbre associé à son premier fils et pour le fils droit son premier frère

F - Parcours d'arbres binaires

Définitions

Parmi les parcours d'arbres binaires **en profondeur** on retrouve :

- le parcours **préfixe** NGD
- le parcours **infixe** GND
- le parcours **postfixe** GDN

```
let rec prefixe a acc = match a with
| Vide -> acc
| Noeud (l, x, r) -> x :: (prefixe l (prefixe r acc))

let rec prefixe a = match a with
| Vide -> acc
| Noeud (l, x, r) -> [x] @ (prefixe g) @ (prefixe d)

let rec infixe a = match a with
| Vide -> acc
| Noeud (l, x, r) -> (infixe g) @ [x] @ (infixe d)
```

Définition

Un parcours est effectué en **largeur** lorsque les noeuds sont explorés par profondeur croissante. Pour effectuer ce parcours de manière itérative, on utilise une structure de file.

IV - Arbres binaires de recherche

Définition inductive

Soit (A, \prec) un ensemble totalement ordonné. L'ensemble des arbres binaires de recherche sur A est défini par : $\begin{cases} \text{Vide est un ABR (BST)} \\ \text{Noeud}(g, x, d) \text{ est un ABR} \end{cases}$ ssi :

- g et d sont des ABR
- soit \max_g le maximum de tous les elts de g et \min_d le minimum de tous les elts de d , alors on doit avoir $\max_g < x < \min_d$

Théorème

Un arbre binaire a est un ABR ssi la liste des étiquettes dans l'ordre infixé est triée par ordre croissant.

Preuve

Par induction structurelle :

- pour l'arbre vide ok
- Soit a un arbre de la forme $a = \text{Noeud}(g, x, d)$. Soit l_g, l_d la liste des étiquettes de l'arbre g et de l'arbre d , et l_a la liste des étiquettes de a .

Par définition de l'ordre infixé :

$l_a = l_g . x . l_d$ (avec . un opération de concaténation)

\Rightarrow Si a est un ABR alors l_g et l_d sont croissants par l'hypothèse d'induction et on a $\max_g < x < \min_d$, et donc la liste $l_a = l_g . x . l_d$ est croissant.

\Leftarrow Si l_a est croissante, $l_g . x . l_d$ donc on a forcément $\max_g < x < \min_d$ (1) et puisque l_a est croissant alors l_g et l_d sont croissantes, donc par hypothèse d'induction g est un ABR et d est un ABR (2). De (1) et (2), a est un ABR

V - Tas

A faire

Noter la définition d'un tas (TD + renvoie en TP cette après-midi)

Chapitre 14 - Paradigmes algorithmiques

- I - Recherche exhaustive (brute force)
 - Retour sur trace (backtracking)
- II - Algorithme gloutons
- III - Décomposition en sous-problèmes
 - A - Diviser pour régner
 - B - Programmation dynamique
- IV - Stratégies algorithmique
- V - Cours propre

Situation

Nous avons un problème à résoudre et nous cherchons une solution, ou la meilleure solution (*problème d'optimisation*)

I - Recherche exhaustive (brute force)

Définition

Effectuer une **recherche exhaustive** (*brute force*), c'est explorer l'ensemble des solutions.

Avantages

On trouve la ou la meilleure solutions si elle existe

Inconvénients

La complexité est en général non polynomiale

Exemple : Pilzgal

→ calculer le nombre de solutions : 2^n
→ parcourir les entiers de 0 à $2^n - 1$ et tester en utilisant la représentation binaire

Retour sur trace (backtracking)

Principe

Le principe est de chercher à construire une solution en parcourant l'arbre des possibilités en profondeur et en remontant dès qu'on détecte que ce n'est pas possible.

II - Algorithme gloutons

Définition

Un algorithme est dit **glouton** lorsqu'il prend des décisions petits à petit, sans revenir sur ses pas.

Avantages

Facile à coder, complexité "sympa"

Inconvénients

Ne donne pas forcément la meilleure solution

Exemple : rendu de monnaie

C'est un *problème d'optimisation*.

On a un montant à rendre, une série de pièces de valeurs différentes (disponibles à l'infini), et on cherche à rendre la monnaie avec *le moins de pièces possibles*.

III - Décomposition en sous-problèmes

A - Diviser pour régner

🔗 Principe

La stratégie **diviser pour régner** est utilisée pour traiter des sous-problèmes indépendants.
Il y a 3 étapes :

- *diviser* : décomposer le problème en sous-problèmes de taille strictement plus petite
- *régner* : traiter les sous-problèmes
- *rassembler / combiner* : rassembler les solutions des sous-problèmes

☰ Exemple : tri fusion

- diviser : découper la liste en deux
- régner : traiter de la même manière les sous-problèmes, lorsqu'on a plus qu'un élément c'est trié
- rassembler : fusion des deux sous-listes (maintenant triées)

B - Programmation dynamique

🔗 Définition

La **programmation dynamique** consiste à traiter des sous-problèmes et à mémoriser les résultats pour les combiner (sans avoir à recalculer).

🔗 Principe

La stratégie de **mémoisation** (une forme de programmation dynamique) consiste à se souvenir par exemple avec un dictionnaire des cas déjà traités.

☰ Exemples d'algorithmes en programmation dynamique

Mémoisation :

```
let binome_memo n k =
  let dico = Hashtbl.create n in
  let rec aux n k =
    if k = 0 then 1
    else if k = n then 1
    else if Hashtbl.mem dico (n, k) then Hashtbl.find dico (n, k)
    else let v = aux (n-1) (k-1) + aux (n-1) k
          in Hashtbl.add dico (n, k) v; v
  in aux n k;;
```

Programmation dynamique avec un tableau :

```
let binome_progdyn_tab n k =
  assert (k <= n);
  let tableau = Array.make (k+1) 0 in
  tableau.(0) <- 1;
  for i = 1 to n do
    for j = k downto 1 do
      tableau.(j) <- tableau.(j-1) + tableau.(j);
    done;
  done;
  tableau.(k)
```

💡 Tip

La technique de programmation dynamique s'applique :

- pour une problème d'optimisation
- si on a des sous-problèmes non-indépendants (on parle de *chevauchement de sous-problèmes*)

🔗 Définition

On parle de **sous-structures optimales** pour faire référence au fait d'utiliser les solutions des sous-problèmes pour construire la solution optimale du problème donné et toute solution d'un sous-problème utilisée est également *optimale* pour le sous-problème.

IV - Stratégies algorithmique

☰ Problème d'organisation d'activités

On a une liste d'activité $[(d_1, f_1), (d_2, f_2), \dots, (d_n, f_n)]$ avec d_i les dates de début et f_i les dates de fin des activités.

Deux activités i et j sont compatibles si $d_i < f_j$ ou $d_j < f_i$

Principe général d'un algo glouton :

- trier les activités selon un critère
- pour chaque activité
 - vérifier si elle est compatible avec toutes les activités déjà insérées
 - si oui l'ajouter au résultat

Optimalité : trier par ordre de leur date de fins

- Il existe une solution optimale contenant (d_1, f_1)
- Si X est une solution optimale contenant (d_1, f_1) , alors $X' = X(d_1, f_1)$ est une solution optimale pour le problème du choix d'activités commençant à la date f_1
⇒ réitérer l'argument pour montrer que l'on peut transformer toute solution optimale en la solution gloutonne

Chapitre 14 - Paradigmes algorithmiques

Situation Nous avons un problème à résoudre et nous cherchons une solution, ou pour les « problèmes d'optimisation », la meilleure solution selon un critère.

Nous avons déjà rencontré différents paradigmes algorithmiques. L'objectif ici est de faire le lien entre toutes ces notions.

I - Recherche exhaustive

Effectuer une **recherche exhaustive**, ou recherche par **force brute**, signifie explorer l'ensemble des solutions. *C'est un terme assez générique.*

Une recherche exhaustive permet d'être sûr de trouver une solution, si elle existe, sur l'instance du problème en question, ou de trouver la meilleure solution.

En revanche, effectuer l'exploration de toutes les solutions conduit souvent à des complexités élevées, c'est-à-dire « non polynomiales » en la taille du problème.

Exemple

Connaissez-vous le « tri stupide » ?

Algorithme : construire toutes les permutations des éléments et pour chaque permutation tester si les éléments sont dans l'ordre. S'arrêter lorsqu'on a trouvé.

Cette définition de l'algorithme n'indique pas comment construire les permutations, cependant le nombre de permutations étant $n!$ avec n le nombre d'éléments du tableau, la complexité de cet algorithme sera également $O(n!)$ en pire cas (la permutation dans l'ordre pouvant être la dernière permutation testée).

REMARQUE 1

Remarque : une recherche exhaustive peut être difficile à écrire !!!

Exemple Pour Pilzegal, une partie des solutions explorées étaient du type « chercher à obtenir la moitié de la somme avec une partie des éléments ».

ASTUCE 1

Lorsque la recherche d'une solution s'exprime de la forme « oui/non » (pour Pilzegal : « liste 1/liste 2 »), explorer toutes les solutions peut se faire en utilisant la représentation binaire des entiers de 0 à $2^n - 1$...

La plupart du temps, il faut explorer l'ensemble des solutions.

Retour sur trace

Une des manières d'effectuer une recherche exhaustive est le **retour sur trace** ou **backtracking**. Le principe est de chercher à construire une solution intégralement en parcourir l'arbre des possibilités et de revenir en arrière lorsqu'il s'avère que la solution ne peut pas être construite.

Exemple L'exemple typique est la recherche des solutions pour le problème des N Reines. (*cf projet des dernières vacances*)
Exemple pour 4 reines...

II - Algorithmes gloutons

On appelle **algorithme glouton** un algorithme qui prend des décisions qualifiées de « locales », c'est-à-dire petit à petit, en utilisant les données disponibles à un moment donné. L'algorithme ne remet pas en cause ses choix lors de l'examen de nouvelles données.

Les algorithmes gloutons permettent de résoudre des problèmes difficiles en donnant une solution de manière généralement très simple et souvent efficace. En revanche, rien n'indique a priori que la vision locale du problème qui est construite par l'algorithme permette de déterminer la meilleure solution.

Cependant, pour certains problèmes ou parfois pour certaines instances d'un problème, les algorithmes donnent effectivement une solution optimale. Ou une solution « suffisamment bonne ».

Exemple fondamental Problème du rendu de monnaie. Il s'agit d'un *problème d'optimisation* dans lequel on cherche à rendre la monnaie en utilisant le moins de pièces possibles.

Exercice Écrire un algorithme permettant de résoudre le problème du rendu de monnaie (en supposant qu'on a suffisamment de pièces de toutes les valeurs dans la caisse !). Quelle est la complexité de cet algorithme ?

Exercice à la maison : écrire l'algorithme en C et en OCaml, éventuellement de manière récursive !

Dans le système de monnaie européen, l'algorithme glouton qui consiste à commencer par rendre la plus grande pièce disponible de valeur non supérieure à la somme à rendre permet de trouver une solution optimale.

Exercice Démouler l'algorithme du rendu de monnaie si on demande de rendre 6 centimes dans un système monétaire dans lequel seules les pièces de 1, 3 et 4 centimes existe. Que constate-t-on ?

Quelques autres problèmes classiquement résolus par des algorithmes gloutons (liste non exhaustive !)

- on connaît la date de début et de fin d'un ensemble de tâches, on souhaite maximiser le nombre de tâches pouvant se dérouler sur une ressource donnée sans intersection entre deux tâches,
- maximiser la valeur des objets que l'on peut mettre dans un sac à dos tout en minimisant son poids,
- effectuer la partition de deux ensembles,
- minimiser le nombre de caractères utilisé pour l'encodage d'un texte (algorithme de Huffman),
- déterminer le plus court chemin dans un graphe pondéré (algorithme de Dijkstra)

III - Décomposition en sous-problèmes

Dans certaines situations, les stratégies privilégiées sont celles qui découpent le problème en sous-problèmes, c'est-à-dire en problèmes plus simples à résoudre.

A. Diviser pour régner

La stratégie **diviser pour régner** (ou *divide and conquer*) est particulièrement utilisée lorsque les sous-problèmes sont indépendants. Chaque sous-problème est à son tour découpé en plusieurs sous-problème(s) jusqu'à atteindre un *cas de base*, un problème simple à résoudre. Cette stratégie est souvent adaptée à une écriture récursive.

On distingue en général trois étapes :

- diviser** décomposer le problème en un ou plusieurs sous-problèmes de tailles strictement plus petites,
- régner** résoudre les sous-problèmes,
- rassembler** rassembler les solutions des sous-problèmes pour résoudre le problème initial.

Exemple fondamental Algorithme du tri fusion. L'algorithme fonctionne de la manière suivante :

- découpe d'un tableau en deux sous-tableaux de même taille (**diviser**),
- utilisation du tri fusion sur chacun des deux sous-tableaux (**régner**),
- fusion des deux sous-tableaux (maintenant triés) (**rassembler**)

Le cas de base est rencontré lorsque la taille du tableau vaut 1 ce qui correspond à un tableau trié.

La complexité vérifie généralement $T(n) = aT(n/b) + f(n)$ avec a le nombre de sous-problèmes, n/b leurs tailles respectives et $f(n)$ la complexité de l'étape de division et de l'étape de rassemblement.

Pour le tri fusion, on avait $T(n) = 2T(n/2) + O(n)$, ce qui donne une complexité en $O(n \log n)$.

Autres exemples parmi les algorithmes déjà rencontrés exponentiation rapide, recherche dichotomique, tri rapide...

B. Programmation dynamique

La **programmation dynamique** est une stratégie de programmation qui consiste à résoudre les sous-problèmes, mémoriser leurs résultats et les combiner pour traiter des problèmes plus grands. L'objectif étant donc d'éviter de calculer plusieurs fois la même chose.

Attention le nom donné à cette stratégie n'a pas de rapport avec la stratégie !

Le gain en complexité temporelle implique généralement une augmentation de la complexité en espace nécessaire.

La résolution d'un problème en utilisant la programmation dynamique nécessite souvent une réécriture du problème. Les problèmes les plus petits sont traités en premier, le calcul est dit mené « de bas en haut ».

Exemple Calcul des coefficients binomiaux.

Exercice à la maison : calcul des coefficients binomiaux en OCaml via une stratégie de programmation dynamique.

1) Mémoïsation

La stratégie de **mémoïsation** consiste à se souvenir du résultat des sous-problèmes déjà traités pour éviter de les recalculer. Il est donc là encore nécessaire de prévoir l'espace mémoire nécessaire pour stocker tous les sous-problèmes.

Ce stockage peut être réalisé à l'aide d'un tableau (liste Python) ou en utilisant une structure de **dictionnaire**.

La résolution d'un problème en utilisant la technique de mémoïsation ramène à un algorithme proche de la version naïve.

Exemple Calcul des coefficients binomiaux : utiliser une table de hachage pour se souvenir des résultats déjà calculés.

Exercice : À écrire également à la maison.

2) Ingrédients programmation dynamique

La technique de programmation dynamique (au sens large) s'applique :

- pour chercher une solution optimale à un problème
- lorsqu'il est possible de décomposer une instance du problème en instances d'un ou plusieurs sous-problèmes

La stratégie consiste donc à chercher à décomposer le problème en sous-problèmes en assurant :

- les solutions des sous problèmes sont utilisées pour résoudre le problème d'optimisation
- si une solution d'un sous-problème est utilisée dans une solution optimale pour le problème d'origine, alors la solution du sous problème utilisée doit être optimale pour le sous-problème

On parle de **sous-structure optimale**. Lorsque les sous-problèmes ne sont pas indépendants, on dit qu'il y a **chevauchement des sous-problèmes** (ou des sous-structures optimales). *Sinon on cherche en général un algorithme du type diviser pour régner.*

Exemple du rendu de monnaie

Reprenons l'exemple du rendu de monnaie.

Comment décomposer le problème en sous-problèmes et construire les sous-structures optimales associées à ces sous-problèmes ?

Idée : déterminer le nombre de pièces nécessaire pour rendre tous les montants inférieurs possibles et en déduire le plus petit nombre de pièces nécessaires pour rendre le montant souhaité...

Objectif : Déterminer le nombre minimal de pièces.

Entrées : `montant` : le montant à rendre
`pieces` : valeurs des pièces disponibles

```

pieces ← tri de tab ← tableau de taille montant +1, toutes les cases initialisées à 0
pour m = 1 à montant (inclus) faire
    pour les pièces de montant inférieur à m faire
        choisir la pièce de valeur p qui minimise la quantité : 1 + tab[m-p]
    fin
    affecter 1 + tab[m-p]
fin
return tab[montant]
```

Chapitre 15 - Graphes et algorithmes associés

- I - Graphes
 - A - Graphe orienté
 - B - Graphe non-orienté
 - C - Arbres et graphes
- II - Représentation des graphes
- III - Algorithmes
 - A - Introduction
 - B - Parcours en profondeur
- IV - Graphes pondérés

I - Graphes

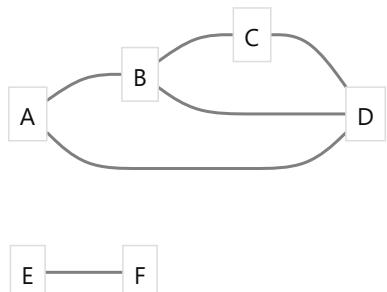
A - Graphe orienté

Définition

Un **graphe orienté** est défini par un ensemble de **sommets / noeuds** S et un ensemble $A \subseteq S \times S$ de couple de sommets, appelés **arcs**.

Exemple

$$\begin{aligned}S &= \{a, b, c, d, e, f\} \\A &= \{(a, b), (a, d), (b, c), (b, d), (c, d), (d, b), (e, f)\}\end{aligned}$$



Définitions

Si $(x, y) \in A$ alors on appelle:

- y le **successeur** de x
- x le **prédecesseur** de y
- y est un voisin de x

Définition

Un arc de la forme (x, x) est appelé une **boucle**.

Définitions

Pour un sommet $x \in S$:

- le nombre d'arcs de la forme $(x, y \in S)$ est appelé **degré sortant** du sommet x , on le note $d_+(x)$
- le nombre d'arcs de la forme $(y \in S, x)$ est appelé **degré entrant** du sommet x , on le note $d_-(x)$

Définitions

Un **chemin** du sommet u au sommet v dans un graphe $G = (S, A)$ est une séquence de sommets $x_0, \dots, x_n \in S$ tels que $u = x_0$, $v = x_n$ et $\forall i \in [[0; n - 1]], (x_i, x_{i+1}) \in A$. La **longueur** de ce chemin est n , c'est le nombre d'arcs à utiliser.
Un chemin est **simple** s'il n'y a pas de répétition d'arêtes.
Un chemin est **élémentaire** s'il n'y a pas de répétition de sommets.
Un **circuit** est un chemin de x à x de longueur $n > 0$.

Définition

Un **DAG** : **Directed Acyclic Graph** est un graphe orienté acyclique.

Définitions

On dit qu'un sommet v est **accessible** depuis un sommet u s'il existe un chemin de u à v .
Une composante **fortement connexe** d'un graphe est un ensemble de sommets tous accessibles deux à deux.
Un graphe est **complet** si tous ses sommets sont voisins deux à deux.

Définitions

Dans un graphe orienté, on note :

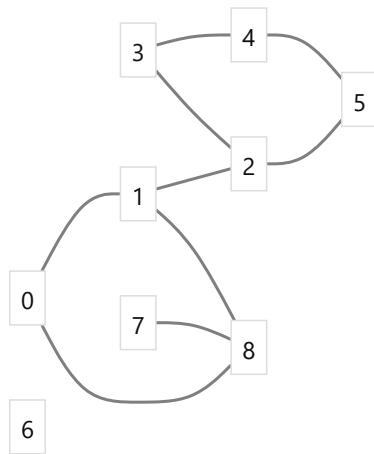
- **racine** (ou **source**) tout sommet de degré entrant nul
- **feuille** (ou **puits**) tout sommet de degré sortant nul

Définition

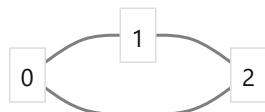
Un **ordre topologique** sur un graphe orienté G est un ordre total sur les sommets de G tel que s'il y a un arc de u vers v alors $u \prec v$.

Exemple

Le graphe suivant possède un ordre topologique :



Le graphe suivant ne possède pas d'ordre topologique :



Théorème

Un graphe orienté $G = (S, A)$ admet un ordre topologique \Leftrightarrow il est **acyclique**.

Preuve

⇒ Soit G un graphe qui admet un ordre topologique.

Si G admet un cycle, choisissons deux sommets u et v de ce cycle, alors d'une part il existe un chemin de u à v et d'autre part il existe un chemin de v à u donc $u \prec v$ et $v \prec u$ absurde.
Donc si G admet un ordre topologique, alors il est acyclique.

⇐ Soit $P(n)$ la propriété "si $G = (S, A)$ avec $n = |S|$ est un graphe orienté acyclique alors il admet un ordre topologique"

Conservation :

Soit $n \geq 1$ tel que $P(n)$ soit vraie. Soit $G = (S, A)$ un DAG tel que $|S| = n + 1$. Puisque G est acyclique, il existe un sommet s de degré sortant nul et soit G' le graphe obtenu en retirant ce sommet.

On a $G' = (S \setminus \{s\}, A \setminus \{\dots, s\})$

Par HR, puisque le nombre de sommet de G' est $|A \setminus \{s\}| = n$ alors il admet un ordre topologique.

(*Preuve incomplète*)

B - Graphe non-orienté

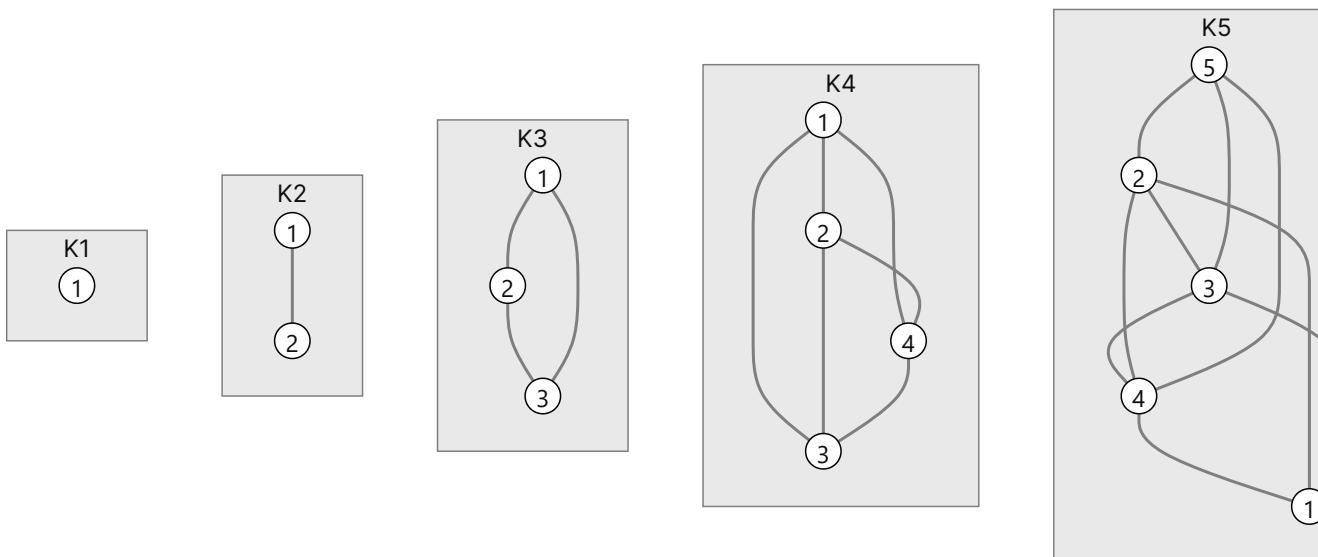
Il existe plusieurs familles de graphes, voici quelques exemples :

Exemples

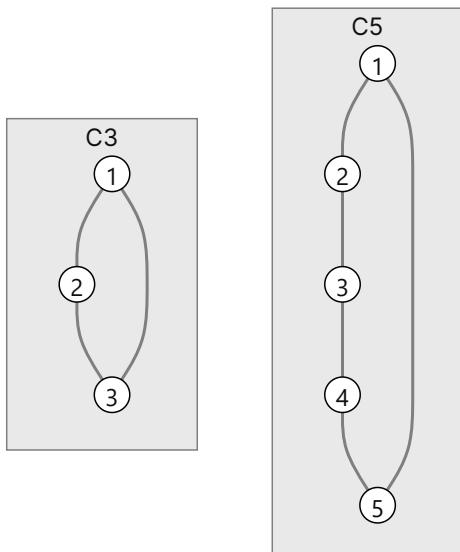
Graphe entièrement déconnecté :



Graphe complet, noté K_n avec n le nombre de sommets :

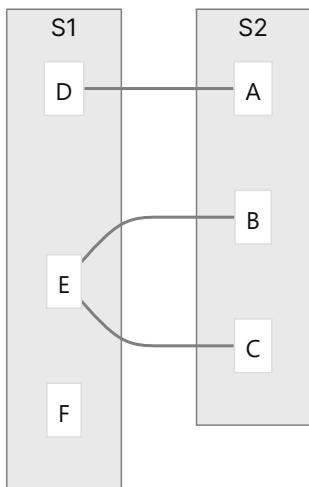


Graphe cycle C_n :



☰ Exemple

Graphe **bipartis**, pour lesquels on peut partitionner l'ensemble des sommets S et S_1 et S_2 de telle sorte que toutes les arêtes du graphe relient un sommet de S_1 à S_2 :



✍ Définitions

Un **graphe non orienté** est défini par un ensemble S de sommets et un ensemble A de paires non orientées de sommets appelées **arêtes**. Le **degré** d'un sommet est son nombre de voisins.

⌚ Proposition

$$\sum_{i \in S} \deg i = 2|A|$$

⌚ Lemme des poignées de main

Dans un graphe, il y a un nombre pair de sommets de degré impair.

✍ Définitions

Un graphe non orienté $G = (S, A)$ est **connexe** si, pour toute paire de sommets x et y de S , il existe un chemin de x à y .

Une **composante connexe** de G est un sous-ensemble de sommets deux à deux reliés par des chemins, maximal pour l'inclusion.

⌚ Définitions

Un **isomorphisme** entre deux graphes orientés $G_1 = (S_1, A_1)$ et $G_2 = (S_2, A_2)$ est une bijection telle que $f : S_1 \rightarrow S_2$ telle que :

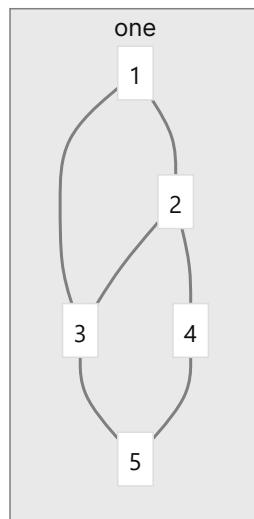
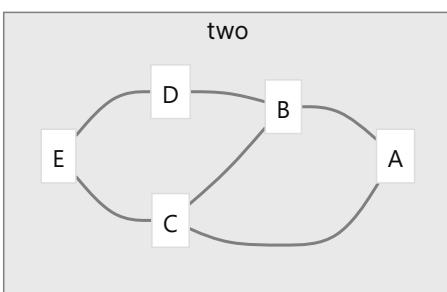
$\forall x, y \in S_1, (f(x), f(y)) \in A_2 \Leftrightarrow (x, y) \in A_1$

Un **isomorphisme** entre deux graphes non orientés $G_1 = (S_1, A_1)$ et $G_2 = (S_2, A_2)$ est une bijection telle que $f : S_1 \rightarrow S_2$ telle que :

$\forall x, y \in S_1, \{f(x), f(y)\} \in A_2 \Leftrightarrow \{x, y\} \in A_1$

Deux graphes sont dits **isomorphes** s'il existe un isomorphisme entre eux.

Exemple



C - Arbres et graphes

Remarques

Un arbre est un graphe :

- non orienté
- connexe
- acyclique

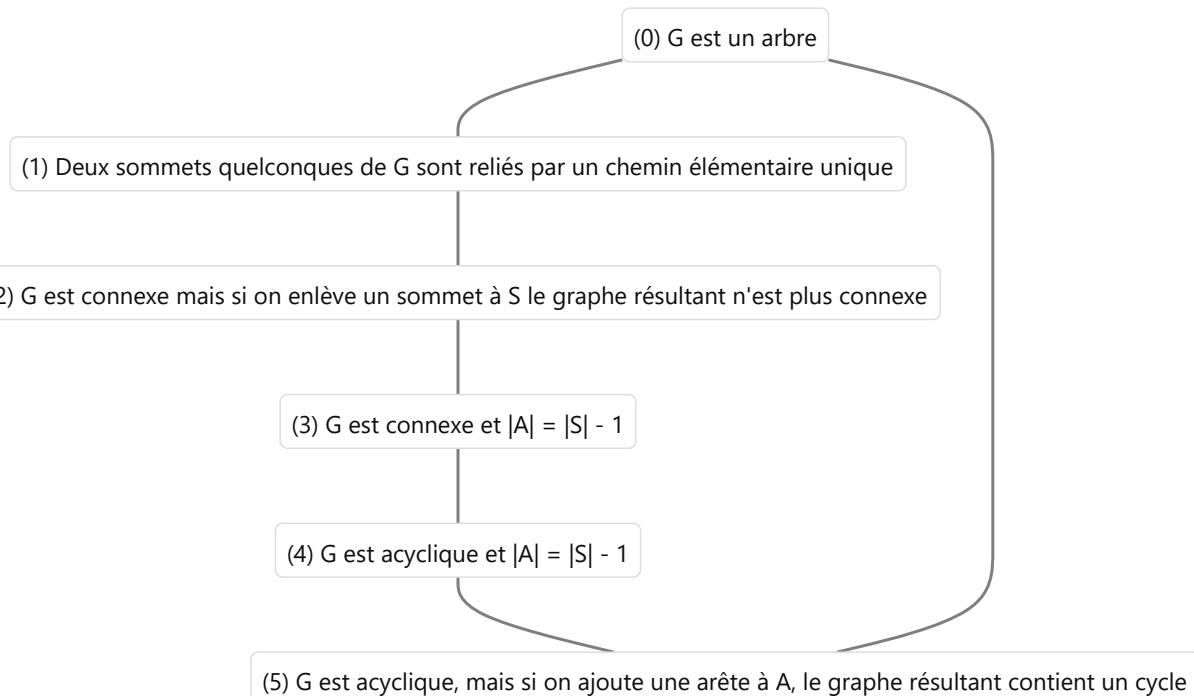
De plus :

- un graphe de ce type est parfois appelé arbre **libre**, ou arbre **non enraciné**, il n'a en effet pas de racine
- si un graphe est non orienté et acyclique : il s'agit d'une forêt

Graphes des arbres

Soit $G = (S, A)$ un graphe non orienté.

Il y a équivalence entre :



II - Représentation des graphes

Représentations

Deux représentations classiques des graphes :

- matrice d'adjacence
- listes d'adjacence

Complexités temporelles

Soit $G = (S, A)$ avec $n = |S|$, $m = |A|$, on obtient les complexités suivantes sur les opérations :

Matrice d'adjacence :

Opération	Complexité
create_graph	$O(n^2)$
has_edge	$O(1)$
add_edge	$O(1)$
neighbours	$O(n)$
all_edges	$O(n^2)$

Listes d'adjacence :

Opération	Complexité
create_graph	$O(n + m)$
has_edge	$O(n)$
add_edge	$O(1)$
neighbours	$O(1)$
all_edges	$O(n + m)$

Complexité spatiale

- matrices d'adjacence : $O(n^2)$
- listes d'adjacences : $O(n + m)$

III - Algorithmes

A - Introduction

Exemple : ordre topologique

```
let rec topo ?(ordre : int list = []) (g : graph) =
  if List.length ordre = nb_vertices g then ordre
  else
    let rec aux i =
      if g.(i) = [] && (not @@ List.mem i ordre) then
        topo ~ordre:(i :: ordre) (Array.map (List.filter (( < ) i)) g)
      else aux (i + 1)
    in
    aux 0
```

B - Parcours en profondeur

⚠ Référence

Voir algorithme sur fiche

```
Initialisations :  
début  
    visités <- ensemble vide  
fin  
Procédure Visiter(s) :  
début  
    si s pas dans visités alors  
        visités <- visités U s  
        pré-traitement(s)  
        pour chaque successeur v de s faire  
            Visiter(v)  
        fin  
        post-traitement(s)  
    fin  
fin  
Visiter(s)
```

📋 Terminaison

Chaque appel de `Visiter()` termine immédiatement ou fait diminuer le nombre de sommets qui ne sont pas dans `visités`, le graphe étant fini l'algorithme termine donc.

📋 Correction

- Après un appel à `Visiter(s)` pour tout sommet $y \in \text{visités}$ il existe un chemin entre s et y . La pile des appels donne le chemin de s à y .
- Après l'appel à `Visiter(s)`, s'il existe un chemin entre s et y alors `Visiter(y)` a été appelé.
 - longueur nulle : $y = s$
 - si la longueur est $n > 0$, on appelle w le dernier sommet atteint avant y , le chemin entre s et w est de longueur $n - 1$ donc le sommet w est visité et `Visiter(w)` a été appelé donc d'après le code `Visiter(y)` a été appellé puisqu'il existe un arc entre w et y .

📋 Conclusion

La parcours en profondeur de s permet de visiter exactement les sommets accessibles depuis s .

✍ Remarque

Si y est accessible depuis s alors les traitements sont effectués dans l'ordre $\text{pré}(s) \prec \text{pre}(y) \prec \text{post}(y) \prec \text{post}(s)$

⌚ Parcours complet

Pour effectuer un parcours complet du graphe $G = (S, A)$, remplacer la ligne 16 par une boucle qui visite tous les sommets de S

📋 Complexité

On impose l'ensemble `visités` à être représenté par un tableau de booléens où `visités[s]` est vrai \iff `Visiter(s)` a été appelé

Création de `visités` et parcours de la ligne 16 : $O(|S|)$

Appel sur un sommet déjà visité : $O(1)$

Sinon, dépend du nb de successeurs du sommet, mais chaque arc n'est visité qu'une seule fois donc $O(|A|)$

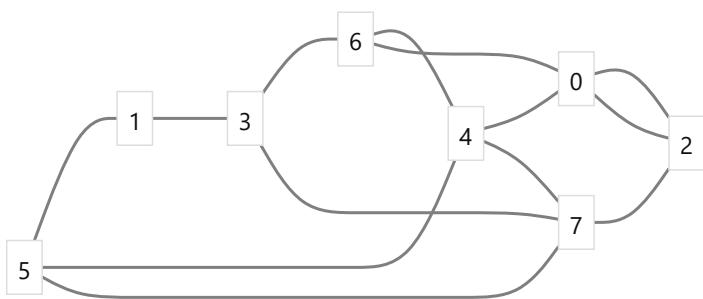
La complexité finale est $O(|S| + |A|)$

☰ Ordre post-fixe

```
Initialisations :  
début  
    liste <- pile vide  
fin  
    pré-traitement(s):  
        // ne rien faire  
fin  
    post-traitement(s):  
        empiler(liste, s)  
fin
```

☰ Ordre post-fixe

Graphe :



Parcours infixé :

[5; 1; 3; 6; 4; 7; 0; 2] où le sommet de la pile est à gauche

⌚ Propriété

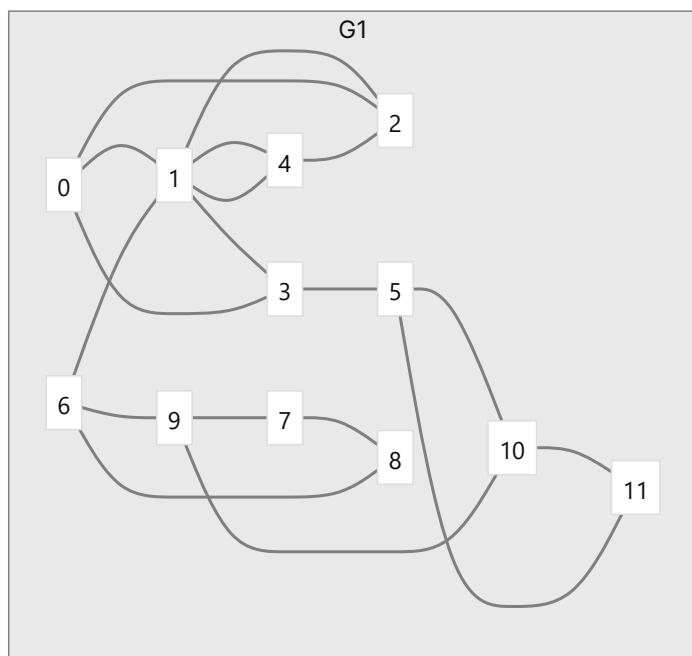
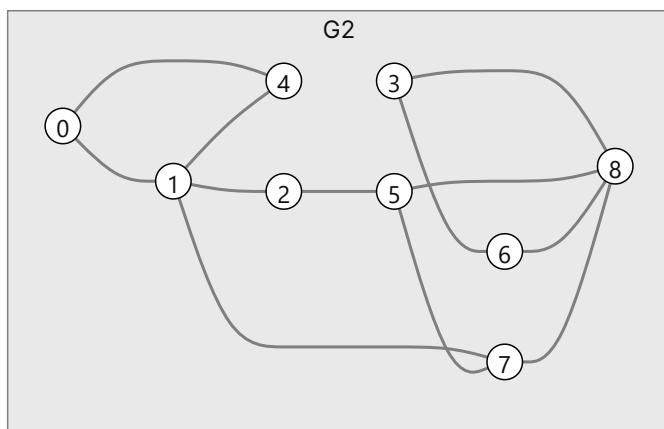
Sur un DAG, l'ordre postfixe renvoyé par le parcours complet est un tri topologique.

📋 Preuve

Soit $u \rightarrow v$ un arc de G , lorsque **Visiter(u)** est appelé pour la première fois :

- soit $v \in \text{visités}$ et dans ce cas v est déjà dans la pile donc u sera ajouté au dessus
- sinon **Visiter(u)** entraîne l'appel **Visiter(v)** donc **post-traitement(v)** effectué avant **post-traitement(u)** et donc u est ajouté après v dans la pile

Graphes d'exemples



III - Algorithmes

B. Parcours en profondeur - suite

ATTENTION 1

La complexité obtenue pour le parcours en profondeur est valable uniquement si le graphe est représenté par un tableau de listes d'adjacence.

Un graphe est-il acyclique ?

Considérons un parcours en profondeur d'un graphe $G = (S, A)$ dans lequel le pré-traitement consisterait à ajouter le sommet dans un ensemble O et le post-traitement consisterait à enlever le sommet de l'ensemble O et à l'ajouter dans l'ensemble F . L'ensemble O représente l'ensemble des sommets *ouverts*, c'est-à-dire pour lesquels une visite a été entamée. Un sommet est dans l'ensemble O tant que l'ensemble de ses successeurs n'a pas été entièrement visités. L'ensemble F représente l'ensemble des sommets *fermés*, c'est-à-dire dont tous les enfants ont été explorés.

Remarque Un sommet est soit dans l'ensemble O , soit dans l'ensemble F , soit dans aucun de ces deux ensembles : $\forall s \in S, (s \in O \vee s \in F) \wedge \neg(s \in O \wedge s \in F)$.

Algorithme : Détermine si un graphe orienté est acyclique (*a dag !*).

Entrées : $G = (S, A)$: un graphe orienté

Résultat : Renvoie False si le graphe contient un cycle, True sinon.

1 Initialisations :

2 **début**

3 | $O \leftarrow \emptyset$ // ensemble des sommets ouverts
4 | $F \leftarrow \emptyset$ // ensemble des sommets fermés

5 **fin**

6 Procédure Visiter(s) :

7 **début**

8 | **si** $s \in O$ **alors**

9 | **renvoyer** False

10 | **fin**

11 | **si** $s \notin F$ **alors**

12 | $O \leftarrow O \cup \{s\}$ /* pré-traitement de s */

13 | **pour chaque** successeur v de s **faire**

14 | | Visiter(v)

15 | | **fin**

16 | | $O \leftarrow O \setminus \{s\}$ /* post-traitement de s */

17 | | $F \leftarrow F \cup \{s\}$

18 | **fin**

19 **fin**

20 **pour chaque** $s \in S$ **faire**

21 | Visiter(s)

22 **fin**

23 **renvoyer** True

Proposition Si un graphe G contient un cycle alors il existe un sommet s qui sera dans l'ensemble O lors d'un des appels à visiter(s).

Démonstration Choisissons deux sommets x et y d'un cycle avec $(x, y) \in A$. Supposons que l'exploration du sommet x commence avant celle du sommet y . D'après le code, le sommet x est ajouté dans l'ensemble O puis la visite du sommet y est exécutée. Le sommet y n'était ni dans O ni dans F . Donc le sommet y est ajouté dans O également et, puisqu'il existe un chemin entre y et x , et qu'aucun des sommets de ce chemin ne peut être dans F , alors une nouvelle visite du sommet x est effectuée avant que le sommet x puisse être marqué fermé.

Proposition si au cours d'une telle exploration, un sommet visité se trouve déjà dans l'ensemble O , alors il existe un cycle dans le graphe G .

Démonstration Supposons que lors de l'évaluation de la ligne 8, un sommet x se trouve (déjà) dans l'ensemble O . L'appel à Visiter(x) ne peut pas provenir de la boucle principale de la ligne 20. En effet, lorsque Visiter(s) est exécuté dans cette boucle, tous les appels précédents sont terminés ce qui implique que les sommets explorés avant x sont dans l'ensemble F . Alors cela impose que ce Visiter(x) a été appelé lors de l'exploration d'un autre sommet, appelons-le y . Il existe donc un arc de y à x . Par ailleurs, puisqu'à la ligne 8 le sommet x est dans l'ensemble O , alors le pré-traitement du sommet x a déjà été exécuté. Ce pré-traitement s'est produit avant le pré-traitement du sommet y qui vient de générer l'appel Visiter(x) considéré et, puisque x n'est pas dans F , avant le post-traitement de x . Le sommet y est donc accessible depuis le sommet x . Puisque y est accessible depuis x et qu'il existe un arc de y à x alors il existe un cycle $x \rightarrow \dots \rightarrow y \rightarrow x$.

Implémentation pour représenter l'appartenance d'un sommet aux ensembles O ou F , définir un **tableau** conservant l'état de chaque sommet, initialisé à une valeur neutre (NonVu), indiquant pour un sommet le fait de ne pas avoir été découvert.

EXERCICE 1

Traduire l'algorithme de détection d'un cycle dans un graphe en OCaml à l'aide d'un tableau d'états et en utilisant le mécanisme de lancement et de récupération d'une exception.

Remarque pour un graphe non orienté, chaque arête $\{u,v\} \in A$ est représentée par un arc (u,v) et un arc (v,u) . L'algorithme précédent détecterait un cycle pour chaque arête !

EXERCICE 2

Écrire un algorithme permettant de déterminer si un graphe non orienté est acyclique.

Variante impérative

Pour décrire un parcours en profondeur de manière itérative, une structure de pile est utilisée. L'algorithme ci-dessous parcourt l'ensemble des sommets accessibles depuis un sommet de départ s .

Effectuer le parcours **complet** du graphe nécessite d'empiler tous les sommets du graphe dans p avant la boucle **tant que**.

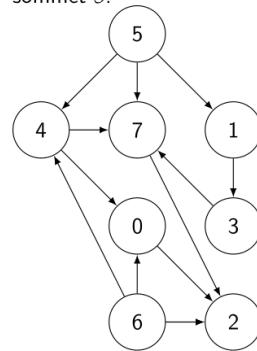
Algorithme : Parcours en profondeur itératif depuis un sommet

```

Entrées :  $G = (S, A)$  : un graphe ;  $s_0$  un sommet de départ
Résultat : Applique un traitement à tous les sommets du graphe
 $G$  accessibles depuis  $s_0$ .
1 Initialisations :
2 début
3    $V \leftarrow \emptyset$            // ensemble des sommets visités
4    $p \leftarrow \text{pile\_vide}()$  // pile des sommets à visiter
5   empiler( $p, s_0$ )
6 fin
7 tant que  $\neg \text{est\_vide}(p)$  faire
8    $s \leftarrow \text{dépiler}(p)$ 
9   si  $s \notin V$  alors
10     $V \leftarrow V \cup \{s\}$ 
11    traitement( $s$ )
12    pour chaque successeur  $v$  de  $s$  faire
13      empiler( $p, v$ );
14    fin
15 fin
16 fin
```

EXERCICE 3

Soit le traitement d'un sommet s : « afficher(s), afficher("—") ». Démouler l'algorithme lors du parcours depuis le sommet 5.



C. Parcours en largeur

Définition Dans un graphe $G = (S, A)$, la **distance** $d(x, y)$ d'un sommet x à un sommet y est la longueur minimale d'un chemin reliant x à y .

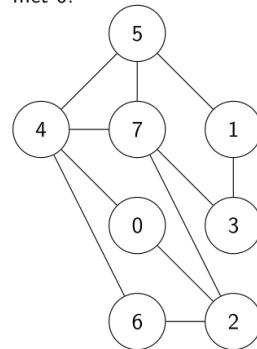
Algorithme : Distances depuis un sommet

```

Entrées :  $G = (S, A)$  : un graphe ;  $s_0$  un sommet de départ
Résultat : Distances des sommets de  $G$  accessibles depuis  $s_0$ .
1 début
2    $f \leftarrow \text{file\_vide}()$            // file des sommets à visiter
3    $\forall s \in S, \text{dist}_s \leftarrow +\infty$  // tableau des distances
4 fin
5 enfiler( $f, s_0$ )
6  $\text{dist}_{s_0} \leftarrow 0$ 
7 tant que  $\neg \text{est\_vide}(f)$  faire
8    $s \leftarrow \text{défiler}(f)$ 
9    $d \leftarrow \text{dist}_s$ 
10  pour chaque successeur  $v$  de  $s$  faire
11    si  $\text{dist}_v = +\infty$  alors
12       $\text{dist}_v \leftarrow d + 1$ 
13      enfiler( $f, v$ )
14    fin
15 fin
16 fin
17 envoyer dist
```

EXERCICE 4

Démouler l'algorithme à partir du sommet 0.



L'algorithme précédent effectue un parcours en largeur du graphe (BFS pour *Breadth-First Search*) réalisé à l'aide d'une file. Ce parcours permet de traiter les nœuds par distance croissante à s_0 . L'idée est de parcourir le graphe en cercles concentriques de distance croissante.

Proposition Un appel à l'algorithme avec G un graphe fini et un sommet source¹ s parcourt l'ensemble des sommets accessibles depuis s et détermine pour chacun la longueur d'un plus court chemin depuis s .

Terminaison Chaque sommet ne peut être inséré qu'une seule fois dans la file et chaque tour de boucle retire un sommet de la file.

Correction Une distance affectée dans le tableau dist n'est pas modifiée par la suite (ligne 11). Notons d_v la distance de la source au sommet v , avec $d_v = +\infty$ si v n'est pas atteignable depuis s .

Soit la propriété $\mathcal{P}(d)$ suivante : « au tout début de l'étape d , pour tout sommet v , $d_v \leq d \iff \text{dist}_v = d_v \neq +\infty$ (1) ; $d_v > d \iff \text{dist}_v = +\infty$ (2), et un sommet v est dans la file ssi $d_v = d \gg$ (3).

initialisation $\mathcal{P}(0) : \forall v \in S \setminus \{s\}, \text{dist}_v = +\infty$ et $\text{dist}_s = 0$ or $d_s = 0$.

conservation Supposons $\mathcal{P}(d')$ pour tout $d' \leq d$ et montrons $\mathcal{P}(d+1)$. Au tout début de l'étape d , tous les sommets de la file sont donc à distance d (HR 3). Pour chaque sommet v à distance d , l'algorithme considère chaque voisin w . D'après le code, si $\text{dist}_w \neq +\infty$ alors il ne se passe rien et $d_w \leq d$ (HR 1). Sinon $\text{dist}_w = +\infty$, donc par HR 2, $d_w > d$ or d'après le code, $\text{dist}_w \leftarrow d+1$. C'est correct car le sommet v est à distance d du point de départ (HR 3) et il existe un arc/une arête $v \rightarrow w$, donc $d_w = d+1$. w est alors ajouté à la file.

Une fois tous les sommets à distance d sortis de la file, seuls des sommets à distance $d+1$ ont été ajoutés dans la file. Montrons qu'ils sont tous ajoutés. Soit w un sommet tel que $d_w = d+1$. Il existe alors un chemin $s \rightarrow \dots \rightarrow v \rightarrow w$ avec $d_v = d$. Le sommet v a été considéré (puisque dans la file d'après HR 3) et on avait $\text{dist}_w = +\infty$ (minimalité de d_w), donc w a été ajouté à la fin.

Enfin, le tableau dist renseigne bien la distance de tout sommet v avec $d_v \leq d+1$. Donc $\mathcal{P}(d+1)$ est vraie.

conclusion L'algorithme s'arrête lorsque la file est vide. Soit d la valeur de cette étape. La propriété implique alors qu'il n'y a aucun sommet à distance d et donc aucun sommet à distance $> d$. La distance de tout sommet à distance $< d$ est renseignée dans dist .

Complexité Chaque sommet est mis dans la file au plus une fois et donc examiné au plus une fois. Chaque arc est considéré au plus une fois, lorsque son origine est examinée. La complexité est donc $O(|S| + |A|)$. Le tableau dist occupe un espace $|A|$ et la file peut contenir jusqu'à $|S| - 1$ sommets dans le pire cas.

D. Nombre de chemins de longueur k

Soit un graphe $G = (S, A)$ représenté par une matrice M d'entiers telle que $M_{i,j} = \begin{cases} 1 & \text{si } (i,j) \in A \\ 0 & \text{si } (i,j) \notin A \end{cases}$

Proposition Pour tout $k \geq 1$, la matrice M^k détermine exactement le nombre de chemins de longueur k entre deux sommets donnés.

Démonstration Soit $\mathcal{P}(k)$ la propriété « $(M^k)_{i,j}$ est le nombre de chemins de i à j de longueur k ».

initialisation Lorsque $k = 1$, $M^1 = M$ donc pour tout i et j , $M_{i,j}^1 = M_{i,j}$ et il existe 1 chemin (de longueur 1) de i à j si et seulement si $M_{i,j} = 1$.

conservation Supposons $\mathcal{P}(k)$ vraie pour $k \geq 1$. Il existe un chemin de longueur $k+1$ entre les sommets i et j si et seulement si il existe un sommet l tel que

- il existe un chemin de longueur k entre i et l ,
- il existe un arc entre l et j .

Par hypothèse de récurrence, le nombre de chemins de longueur k entre i et l vaut $(M^k)_{i,l}$. Ainsi le nombre de chemins de longueur $k+1$ entre i et j et dont l'avant dernière étape est l est égal à $(M^k)_{i,l} M_{l,j}$ (il existe un arc entre l et j donc $M_{l,j} = 1$).

Alors le nombre de chemins de longueur $k+1$ qui relient i à j vaut $\sum_{l=0}^{|S|-1} M_{i,l}^k M_{l,j} = (M^{k+1})_{i,j}$.

1. point de départ, pas forcément une source

IV - Graphes pondérés

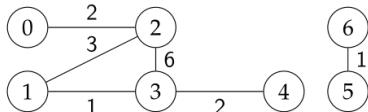
Lorsqu'une information est associée aux arcs d'un graphe, ce graphe est appelé un **graphe pondéré**.

Formellement Un **graphe pondéré** est un triplet $G = (S, A, p)$ où S est l'ensemble des sommets, A l'ensemble des arcs et p une application de A dans \mathbb{R} qui à un arc associe une grandeur généralement appelée **poids**.

Le valeur $+\infty$ est souvent utilisée comme poids pour représenter l'absence de chemin entre deux sommets.

Le **poids** d'un chemin est la somme des poids des arcs/arêtes constituant le chemin.

La **distance** entre deux sommets d'un graphe pondérée est le minimum des poids des chemins entre les deux sommets considérés.



EXERCICE 5

Déterminer la matrice des distances associées entre chaque paire de sommets du graphe ci-contre.

Notion de plus court chemin Dans un graphe pondéré, un **plus court chemin** entre deux sommets x et y est le chemin de x à y dont le poids est le plus petit.

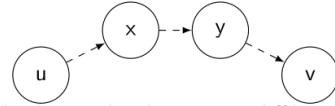
Remarque En général, seuls les graphes ne possédant pas de cycle de poids strictement négatif sont considérés.

A. Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall est un algorithme utilisant une stratégie de programmation dynamique pour déterminer la distance entre toute paire de sommets d'un graphe pondéré sans cycle de poids négatif.

Tout sous chemin d'un plus court chemin est lui-même un plus court chemin.

Illustration soit un plus court chemin de u à v , passant par deux sommets x et y . Alors la portion du plus court chemin qui se trouve entre x et y est elle-même un plus court chemin entre x et y .



Le principe de l'algorithme de Floyd-Warshall est de chercher des chemins passant par de plus en plus de sommets différents. Initialement seuls les chemins réduits à un seul arc, donc sans sommet intermédiaire sont considérés. Puis les chemins qui empruntent uniquement le sommet 0 comme sommet intermédiaire, donc de la forme $u \rightarrow 0 \rightarrow v$. Puis les chemins peuvent emprunter jusqu'au sommet 1, etc...

Soit une numérotation x_0, \dots, x_{n-1} des sommets d'un graphe $G = (S, A)$ avec $n = |S|$. Pour $0 \leq k \leq n$, $pcc_k(u, v)$ est le poids d'un plus court chemin du sommet u au sommet v dans lequel chaque sommet intermédiaire (un sommet du chemin autre que u et v) est numéroté de 0 à $k - 1$.

$$\begin{cases} pcc_0(u, v) &= p(u, v) \\ pcc_{k+1}(u, v) &= \min(pcc_k(u, v), pcc_k(u, x_k) + pcc_k(x_k, v)) \quad \text{si } 0 \leq k < n \end{cases}$$

Alors pour chaque couple (u, v) de sommets, le poids d'un plus court chemin de u à v est donné par $pcc_n(u, v)$.

Algorithme : Algorithme de Floyd-Warshall

Entrées : A la matrice d'adjacence d'un graphe pondéré à n sommets.

Résultat : Une matrice carrée D de taille $n \times n$ avec $D_{i,j} = pcc(x_i, x_j)$

```

1  $D \leftarrow \text{copie}(A)$ 
2 pour  $k = 0$  à  $n - 1$  faire
3   pour  $i = 0$  à  $n-1$  faire
4     pour  $j = 0$  à  $n-1$  faire
5        $D_{i,j} \leftarrow \min(D_{i,j}, D_{i,k} + D_{k,j})$ 
6     fin
7   fin
8 fin
9 renvoyer  $D$ 

```

V - Graphes pondérés

A. Algorithme de Floyd-Warshall

Complexité Soit $n = |S|$ le nombre de sommets du graphe. La complexité temporelle de l'algorithme de Floyd-Warshall est $\mathcal{O}(n^3)$: la création et l'initialisation de la matrice D est en $\mathcal{O}(n^2)$. La mise à jour de la matrice est en $\mathcal{O}(n^3)$ (trois boucles `for` de taille n imbriquées).

La complexité spatiale de l'algorithme est $\mathcal{O}(n^2)$ ce qui correspond à l'espace occupé par la matrice D de taille $n \times n$ utilisée pour mémoriser les poids des plus courts chemins entre chaque paire de sommets du graphe. Pour reconstruire les chemins, une deuxième matrice de taille $n \times n$ est nécessaire pour mémoriser l'indice k qui a permis d'obtenir une distance plus courte pour chaque paire (i, j) .

Terminaison L'algorithme de Floyd-Warshall termine (boucles `for` dont la taille est connue à l'avance).

Correction Proposer un invariant pour la boucle `for` sur l'indice k et le justifier rapidement.

EXERCICE 1

Soit un graphe pondéré représentant un réseau de communications de telle sorte que le poids associé à chaque arc corresponde au temps qu'il faut à un message pour traverser un lien de communication. Décrire un algorithme qui permet de déterminer la plus longue durée de transmission d'un message au sein du réseau.

B. Algorithme de Dijkstra

L'algorithme de Dijkstra permet de déterminer les plus courts chemins dans un graphe pondéré G à partir d'une source s_0 . L'algorithme de Dijkstra consiste à parcourir le graphe en traitant à chaque étape le sommet qui se trouve à la plus petite distance de la source parmi les sommets non encore traités.

Voici une variante avec une **file de priorité min** qui accepte l'opération `DIMINUER-CLÉ`.

Algorithme : Algorithme de Dijkstra

```

Entrées :  $G$  : un graphe pondéré ;  $s_0$  : un sommet de départ
Résultat : Pour chaque sommet  $v \in S \setminus \{s_0\}$  : pluscourt(v) contient le poids d'un plus court chemin de  $s_0$  à  $v$ ,  

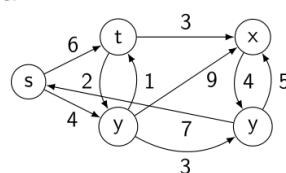
pred(v) est le sommet qui précède  $v$  sur un plus court chemin de  $s_0$  à  $v$ 

1 Initialisations :
2 début
3    $\forall s \in S$ , pluscourt(s) ← +∞                                // tableau des poids
4    $\forall s \in S$ , pred(s) ← None                               // tableau des prédecesseurs
5   pluscourt( $s_0$ ) ← 0
6    $FP \leftarrow \text{FILE\_PRIORITY\_VIDE}()$                       // file de priorité min
7    $\forall s \in S$ , INSÉRER(FP, (pluscourt(s), s))      // insertion des sommets - priorité : distance à  $s_0$ 
8 fin
9 tant que  $\neg \text{EST\_VIDE}(FP)$  faire
10   $u \leftarrow \text{EXTRAIRE-MIN}(FP)$ 
11    pour chaque successeur  $v$  de  $u$  faire
12      si pluscourt(v) > pluscourt(u) + d_{u,v} alors
13         $pluscourt(v) \leftarrow pluscourt(u) + d_{u,v}$ 
14        pred(v) ← u
15        DIMINUER-CLÉ(FP, v, pluscourt(v))
16      fin
17    fin
18 fin
19 renvoyer pluscourt, pred
```

Cet algorithme utilise une stratégie gloutonne.

EXERCICE 2

Dérouler l'algorithme sur le graphe ci-contre.



REMARQUE 1

L'implémentation concrète et l'analyse de l'algorithme seront travaillées en TP.

☰ Correction de l'algorithme de Floyd-Warshall

Invariant :

Après la boucle k , la matrice D contient le poids du plus petit chemin dont les indices des sommets intermédiaires sont compris entre 0 et k (inclus).

Initialisation :

D'après le code, $D \leftarrow A$ donc $\forall i, j, \begin{cases} D_{i,j} = +\infty & \text{si pas d'arc de } i \text{ à } j \\ D_{i,j} = p(i,j) & \text{sinon} \end{cases}$ ce qui correspond bien au chemin sans sommet intermédiaire.

Conservation :

Supposons ok pour k .

- soit passer par $k+1$ permet de construire un chemin plus court, $D_{i,j} = D_{i,k} + D_{k,j}$
- sinon on conserve $D_{i,j}$
⇒ la propriété est conservée pour $k+1$

Conclusion :

Après la dernière itération, on a traité $k = n - 1$ donc la matrice D contient le poids du plus petit chemin dont les indices des sommets intermédiaires sont compris entre 0 et $n - 1$.

Chapitre 16 - Logique

- I - Définitions
 - A - Formules en tant qu'arbre
 - B - Définition inductive
- II - Sémantique du calcul propositionnel
 - A - Notion de valuation
 - B - Satisfabilité
 - C - Équivalence entre deux formules
 - D - Substitution
- III - Formes normales
 - A - FNC et FND
 - 1) Forme Normale Conjonctive
 - 2) Forme Normale Disjonctive
 - B - Mise sous forme normale disjonctive
 - C - Mise sous forme normale conjonctive
- IV - Problème SAT
 - D - Application : coloration de graphe
- V - Logique du premier ordre

I - Définitions

A - Formules en tant qu'arbre

Définition

Une **formule de logique propositionnelle** est un arbre ayant des noeuds binaires, étiquetés par \wedge , \vee , \Rightarrow , \Leftrightarrow ou des noeuds unitaires étiquetés par \neg dont les feuilles sont étiquetées par des éléments de \mathcal{V} , l'ensemble des variables propositionnelles.

Définition

Pour une formule propositionnelle P :

- la **hauteur** de P est la hauteur de l'arbre correspondant
- la **taille** de P est le nombre des noeuds (internes ou non) de l'arbre
- une **sous-formule** de P est sous-arbre de P , soit un arbre enraciné en l'un des noeuds de P

B - Définition inductive

Définition

L'ensemble F des formules de la logique propositionnelle est défini inductivement :

- toute variable propositionnelle p est une formule
- si φ est une formule alors $\neg\varphi$ est une formule
- si φ, ψ sont des formules, alors $(\varphi \vee \psi)$, $(\varphi \wedge \psi)$, $(\varphi \Rightarrow \psi)$ et $(\varphi \Leftrightarrow \psi)$ sont des formules

Définition

L'ensemble $SF(\varphi)$ des **sous-formules** d'une formule φ est défini inductivement :

- $SF(p) = \{p\}$
- $SF(\neg\varphi) = \{\neg\varphi\} \cup SF(\varphi)$
- $SF(\varphi \vee \psi) = \{\varphi \vee \psi\} \cup SF(\varphi) \cup SF(\psi)$
- $SF(\varphi \wedge \psi) = \{\varphi \wedge \psi\} \cup SF(\varphi) \cup SF(\psi)$
- $SF(\varphi \Rightarrow \psi) = \{\varphi \Rightarrow \psi\} \cup SF(\varphi) \cup SF(\psi)$

Exemple

$$SF(p \Rightarrow (q \vee r)) = \{p \Rightarrow (q \vee r), p, q \wedge r, q, r\}$$

II - Sémantique du calcul propositionnel

A - Notion de valuation

Définition

Soit un ensemble à deux éléments $\mathbb{B} = \{V, F\}$. Une **valuation** est une application $v : \mathcal{V} \rightarrow \mathbb{B}$

Exercice

Ecrire la table de vérité de la formule $\varphi = (p \wedge q) \vee \neg p$. Evaluer la formule avec la valuation $v(p) = F, v(q) = F$

p	q	$p \wedge q$	$\neg p$	φ
V	V	V	F	F
V	F	F	F	F
F	F	F	V	V
F	V	F	V	V

Définition

Si une valuation v **satisfait** une formule φ on note $v \models \varphi$

Si $v \models \varphi$ alors on dit que v est un **modèle** de φ .

$Mod(\varphi) = \{v \mid v \models \varphi\}$ est l'ensemble des modèles de φ .

Example

$\varphi = (p \wedge q) \vee \neg p, Mod(\varphi) = \{(p \mapsto V, q \mapsto V), (p \mapsto F, q \mapsto V), (p \mapsto F, q \mapsto F)\}$

B - Satisfabilité

Définition

Une formule est :

- **satisfiable** si elle admet un modèle ($Mod(\varphi) \neq \emptyset$)
- **tautologique** si toute valuation est un modèle
- **antilogique** si aucune valuation n'est pas un modèle ($Mod(\varphi) = \emptyset$)

Notation

Une tautologie est représentée par \top

Une antilogie est représentée par \perp

C - Equivalence entre deux formules

Définition

Les formules φ et ψ sont **équivalentes** lorsque $Mod(\varphi) = Mod(\psi)$. On le note $\varphi \equiv \psi$

Définition

Si φ et ψ sont deux formules propositionnelles :

ψ est une **conséquence logique** de φ si $Mod(\varphi) \subseteq Mod(\psi)$, autrement dit si toute valuation satisfaisant φ satisfait également ψ . On le note $\varphi \models \psi$

Equivalence

$\varphi \equiv \psi \Leftrightarrow \varphi \models \psi \text{ et } \psi \models \varphi$
 $\varphi \equiv \psi \Leftrightarrow \models (\varphi \Leftrightarrow \psi)$

Transitivité

Si $P \models Q$ et $Q \models R$ alors $P \models R$

Elements neutres

$\varphi \wedge \top \equiv \varphi$
 $\varphi \vee \perp \equiv \varphi$

Associativité et commutativité

$(\varphi \vee \psi) \vee \theta \equiv \varphi \vee (\psi \vee \theta)$
 $(\varphi \wedge \psi) \wedge \theta \equiv \varphi \wedge (\psi \wedge \theta)$
 $\psi \vee \varphi \equiv \varphi \vee \psi$
 $\varphi \wedge \psi \equiv \psi \wedge \varphi$

Distributivité

$P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$
 $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$

Lois de De Morgan

$\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$
 $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$

Négation

$\neg\neg\varphi \equiv \varphi$

Tiers exclus

Tautologie : $P \vee \neg P \equiv \top$
Antilogie : $\neg(P \wedge \neg P) \equiv \top$, $P \wedge \neg P \equiv \perp$
 $\models P \vee \neg P$
 $\models \neg(P \wedge \neg P)$

Contraposée

$\models [(\varphi \rightarrow \psi) \leftrightarrow (\neg\psi \rightarrow \neg\varphi)]$

Exportation

$(\varphi \wedge \psi \rightarrow \theta) \leftrightarrow (\varphi(\psi \rightarrow \theta)) \equiv \top$

D - Substitution

Définition

Soient deux formules φ, ψ et une variable propositionnelle $p \in \mathcal{V}$. La **substitution**, notée $\varphi\{p \mapsto \psi\}$ (ou $\varphi[\psi \setminus p]$) est définie inductivement par :

- Si $\varphi = p$, alors $\varphi\{p \mapsto \psi\} = \psi$
- Si $\varphi = q$ et $q \neq p$, alors $\varphi\{p \mapsto \psi\} = \varphi$
- Si $\varphi = \neg\varphi_1$, alors $\varphi\{p \mapsto \psi\} = \neg(\varphi_1\{p \mapsto \psi\})$
- Si $\varphi = \varphi_1 \wedge \varphi_2$, alors $\varphi\{p \mapsto \psi\} = \varphi_1\{p \mapsto \psi\} \wedge \varphi_2\{p \mapsto \psi\}$
- Si $\varphi = \varphi_1 \vee \varphi_2$, alors $\varphi\{p \mapsto \psi\} = \varphi_1\{p \mapsto \psi\} \vee \varphi_2\{p \mapsto \psi\}$
- Si $\varphi = \varphi_1 \Rightarrow \varphi_2$, alors $\varphi\{p \mapsto \psi\} = \varphi_1\{p \mapsto \psi\} \Rightarrow \varphi_2\{p \mapsto \psi\}$

III - Formes normales

A - FNC et FND

1) Forme Normale Conjonctive

Définition

Un **littéral** est une formule de la forme p ou $\neg p$ avec $p \in \mathcal{V}$.

Une **clause** est une formule de la forme $\varphi_1 \vee \dots \vee \varphi_m$ où les φ_i sont des littéraux.

Une **forme normale conjonctive** est une formule de la forme $C_1 \wedge \dots \wedge C_n$ où les C_i sont des clauses.

2) Forme Normale Disjonctive

Définition

C'est une disjonction de clauses conjonctives.

B - Mise sous forme normale disjonctive

Exemple

Déterminer la forme normale disjonctive de la formule $\varphi = (p \wedge q) \vee \neg p$

$\varphi = (p \wedge q) \vee (\neg p \wedge \neg q) \vee (\neg q \wedge q)$ (obtenu avec la table de vérité de la formule)

C - Mise sous forme normale conjonctive

Méthode

En partant d'une formule quelconque :

- transformer les implications
- faire "descendre" les négations
- faire "remonter" les connecteurs \wedge : remplacer $P \vee (Q_1 \wedge Q_2)$ par $(P \vee Q_1) \wedge (P \vee Q_2)$

Exemple

Déterminer la forme normale conjonctive de la formule $\varphi = (p \wedge q) \vee (z \wedge r)$

$\varphi \equiv (p \vee z) \wedge (q \vee z) \wedge (p \vee r) \wedge (q \vee r)$

IV - Problème SAT

A. Présentation

LE **problème SAT**, qui consiste à déterminer si une formule est satisfiable, est un problème fondamental en informatique¹. Ce problème est un **problème de décision** : un algorithme qui résout ce type de problème détermine en un temps fini sur toute **instance** du problème si la réponse est vraie ou fausse.

Exemple : un graphe donné admet-il un circuit Eulérien ?

Remarque : il est en général possible de transformer un problème d'optimisation (cf chapitre 14) en un problème de décision.

Exemple : est-il possible de rendre la monnaie en utilisant au plus n pièces ?

Lorsque le problème SAT est appliquée à une formule en Forme Normale Conjonctive dont les clauses comportent au plus k littéraux, le problème est nommé **k -SAT**, abréviation de k -FNC-SAT.

B. Quelques variantes de k -SAT

1-SAT Les formules à résoudre sont donc des conjonctions de littéraux. Soit deux littéraux opposés apparaissent dans la formule et dans ce cas la formule n'est pas satisfiable, soit elle est satisfiable en choisissant une valuation qui rend tous les littéraux positifs.

2-SAT Dans cette variante du problème, chaque clause comporte au plus deux littéraux par clause, comme dans la formule $(x \vee \neg y) \wedge (\neg x \vee y) \wedge (\neg x \vee z) \wedge (y \vee \neg z)$.

Pour cette forme, il est possible de résoudre le problème avec une complexité temporelle polynomiale.

3-SAT Le problème phare ! En effet, il est possible de montrer que toute formule propositionnelle peut s'écrire sous forme normale conjonctive avec des clauses de 3 littéraux au maximum. Ainsi savoir décider pour 3-SAT permet de savoir décider pour SAT.

C. Algorithme de Quine

L'algorithme de Quine consiste à construire un arbre de décision associé à une formule F .

- si F est une valeur booléenne alors $f \equiv \top$ ou \perp : créer une feuille avec cette valeur
- sinon soit x une variable quelconque dans F , construire un arbre binaire dont la racine est étiquetée par x et dont le fils gauche représente la formule F dans laquelle la variable x est substituée x par \top , le fils droit représente la formule F dans laquelle la variable x est substituée par \perp .

Ensuite, pour décider que F est satisfiable il « suffit » de trouver une feuille de valeur \top dans cet arbre.

D. Application : coloration de graphe

Soit $G = (S, A)$ un graphe non orienté comportant n sommets. Le problème ici est de déterminer s'il existe une **coloration** de G à k couleurs.

Représentons chaque sommet de S par un entier i de 1 à n et chaque couleur par un entier j entre 0 et $k - 1$ avec k le nombre de couleurs nécessaires.

Posons :

$$x_{ij} = \begin{cases} V & \text{si couleur}(i) = j \\ F & \text{sinon} \end{cases}$$

Écrire chaque contrainte ci-dessous à l'aide d'une formule de la logique propositionnelle :

1. chaque sommet a au moins une couleur
2. chaque sommet a au plus une couleur
3. deux sommets adjacents n'ont pas la même couleur

En déduire une formule en forme normale conjonctive représentant ce problème.

V - Logique du premier ordre

Jusqu'à présent, nous nous sommes intéressés à des formules constituées de variables propositionnelles et de connecteurs logiques. Nous introduisons ici la **logique des prédictats** qui permet d'exprimer des structures mathématiques.

Un **terme** est obtenu à partir de **symboles de variables** et de **symboles de fonctions**.

1. vous en saurez plus l'année prochaine !

Soit un ensemble infini de variables $X = \{x, y, x_1, x_2, \dots\}$ et un ensemble $\mathcal{F} = \{c, f, g, \dots\}$ de **symboles de fonctions**, étant donnée une application arite qui donne le nombre d'arguments de chaque symbole de fonctions. L'ensemble des éléments d'arité n sont notés \mathcal{F}_n . Les éléments de \mathcal{F}_0 sont appelées **constantes**.

Définition Soit une un ensemble \mathcal{F} et un ensemble X (de variables). Les **termes** sur \mathcal{F} et X sont définis par induction :

- toute variable $x \in X$ est un terme
- tout symbole d'arité 0 est un terme
- $f(t_1, \dots, t_n)$ est un terme dès lors que f est un symbole d'arité n et que t_1, \dots, t_n sont des termes

Soit maintenant \mathcal{R} un ensemble de **symboles de relations**. Ces symboles permettent de décrire des propriété entre des éléments. L'ensemble des symboles de relations d'arité n est noté \mathcal{R}_n . Le symbole $=$ est un symbole de relation d'arité 2, interprété comme l'égalité. Ce symbole appartient toujours à \mathcal{R} .

Signature : paire $S = (\mathcal{F}, \mathcal{R})$ avec \mathcal{F} un ensemble (dénombrable) de symboles de fonctions avec leur arité et \mathcal{R} un ensemble (dénombrable) de symboles de relations avec leur arité.

Définition Étant donnée une signature $S = (\mathcal{F}, \mathcal{R})$, une **formule atomique** sur S est de la forme $R(t_1, \dots, t_n)$ où t_1, t_2, \dots, t_n sont des termes et $R \in \mathcal{R}_n$ un symbole de relations d'arité n .

Définition Soit une signature $S = (\mathcal{F}, \mathcal{R})$ et un ensemble X de variables. Les **formules du premier ordre** sur S sont définies inductivement :

- toute formule atomique sur S est une formule sur S
- si F est une formule alors $\neg F$ est une formule (du premier ordre)
- si F_1, F_2 sont des formules alors $F_1 \wedge F_2$, $F_1 \vee F_2$ et $F_1 \rightarrow F_2$ sont des formules
- si F est une formule et x est une variable, alors $\forall x F$ et $\exists x F$ sont des formules

Construire l'arbre associé à la formule $\exists x P(x) \vee \exists x (P(x) \rightarrow \forall y R(x, y))$.

Définition Une occurrence d'une variable x est **liée** lorsqu'elle appartient à une sous-formule précédée d'un **quantificateur** $\forall x$ (quantificateur universel) ou $\exists x$ (quantificateur existentiel). Sinon elle est **libre**.

Une formule est **close** si elle ne possède aucune variable libre. Une formule close F est **satisfiable** si elle possède un modèle, c'est-à-dire une S -structure \mathcal{M} tel que $\mathcal{M} \models F$.

Définition La **portée** d'une variable associée quantificateur est le sous-arbre enraciné en ce quantificateur.

Substitution d'une formule Substituer une variable dans une formule consiste à remplacer les occurrences libres de variables par des termes.

- il est interdit de substituer des occurrences liées
- la substitution ne doit pas faire apparaître de nouvelles occurrences de variables liées : il faut d'abord **renommer** avant de substituer.

Exemple : soit la formule $\exists y. p(x, y)$.

- Substituer x par $f(z)$.
- Substituer x par $f(y)$.

Quelques lois des quantificateurs

$$\begin{aligned}\neg \forall x. F &\equiv \exists x. \neg F \\ \neg \exists x. F &\equiv \forall x. \neg F \\ \forall x. (F_1 \wedge F_2) &\equiv \forall x. F_1 \wedge \forall x. F_2 \\ \exists x. (F_1 \vee F_2) &\equiv \exists x. F_1 \vee \exists x. F_2\end{aligned}$$

Sources

- Informatique MP2I/MPI, Balabonski et al., Ellipses, 2022,
- Introduction à l'algorithme, Cormen et al., Dunod, 2004,
- Cours MP2I. 2021-2022, Bianquis, poly de cours, 2022,
- Logique : fondements et applications, Barbenchon et al., Dunod, 2022

D - Application : coloration de graphe

Exercice

chaque sommet possède au moins une couleur $\iff \bigwedge_{i=1}^n \bigvee_{j=0}^{k-1} x_{ij}$

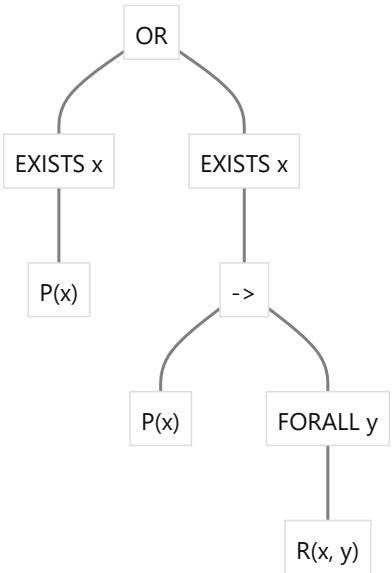
chaque sommet possède au plus une couleur $\iff \bigwedge_{i=1}^n \bigwedge_{j_1, j_2 \in [|0; k-1|], j_1 \neq j_2} (\neg x_{ij_1} \vee \neg x_{ij_2})$

deux sommets adjacents n'ont pas la même couleur $\iff \bigwedge_{i=1}^{n-1} \bigwedge_{j=j+1}^n \bigvee_{q=0}^{k-1} ((i, j) \in A \Rightarrow x_{iq} \wedge \neg x_{jq})$

V - Logique du premier ordre

Exemple 1

Arbre associé à la formule $\exists x P(x) \vee \exists x (P(x) \Rightarrow \forall y R(x, y))$



Chapitre 17 - Fichiers et Entrées-Sorties

- I - Rappels
- II - Système de fichiers
 - A - Structure du contenu d'un fichier
 - B - Nommage de fichiers
 - C - Liens
- III - Entrées / Sorties
 - A - Accès séquentiel
 - B - Flots d'entrée / sortie
 - C - API C et OCaml

I - Rappels

⚠ Voir feuille

II - Système de fichiers

✍ Définition

Système de fichiers : structure de fichiers contenus sur un média

Système de gestion de fichiers : logiciel interne du SE qui gère le SF

Monde Unix et norme POSIX sont associés.

ⓘ Stockage d'informations

- média de stockage (SSD, HDD etc...)
- mécanisme de nommage / implantation

Il y a une distinction entre les données dans la mémoire vide de l'ordinateur et les données sur un disque physique.

📋 Ordres de grandeur

Temps d'accès aux données :

- de $70 \text{ à } 300 \mu\text{s}$ pour un SSD
- quelques ms pour un disque traditionnel

A - Structure du contenu d'un fichier

✍ Définition

Un **fichier** contient une suite d'octets découpés en blocs numérotés.

⚡ Fichiers de texte

Les octets contiennent les codes des caractères.

Structuration en ligne : suite de caractères + caractère fin

ⓘ Sortes de blocs

- des blocs de contenus dispersés ou non sur le disque
- des blocs décrivant l'implantation des blocs de contenu
- des blocs permettant d'associer des informations à ces octets

Note

Cette manière de représentation a un impact sur la lecture des fichiers.

Fragmentation : regrouper les données pour avoir des meilleures performances

B - Nommage de fichiers

Dans le monde Unix

- donner un numéro aux fichiers, le numéro d'**inode** (noeud d'index)
- maintenir une table de correspondance <nom, inode>
- créer une arborescence

Définition

Inode : contient les caractéristiques d'un fichier :

- numéro d'inode
- taille du fichier
- identifiant du périphérique de stockage
- date de dernière modification
- identifiants et groupe propriétaire
- droits accordés
- nombre de liens physiques

Tip

Pour obtenir l'inode d'un fichier sur Unix :

SHELL

```
stat nom_fichier  
ls -i nom_fichier
```

C - Liens

Définitions

Un **lien** est une association <nom, inode>

Un **répertoire** est un fichier qui contient une liste de liens

Remarque

Copier / coller recréer un nouveau inode, différent

Créer un lien

SHELL

```
# création d'un lien symbolique  
ln -s fichier lien_vers_fichier  
  
# création d'un lien physique  
ln fichier lien_vers_fichier
```

L'extension du lien doit être la même que celle du fichier.

Un lien physique possède la même inode que le fichier de base.

⌚ Supprimer un lien

Suppression du lien physique :

SHELL

```
rm lien_vers_fichier
```

⇒ diminution du compteur de lien de l'inode.
Si le compteur arrive à 0, la mémoire peut être libérée.

III - Entrées / Sorties

A - Accès séquentiel

ⓘ Accès séquentiel

L'**accès séquentiel** est l'accès le plus courant. Tout accès suppose l'**ouverture** du fichier, et donc ensuite la **fermeture** du fichier.

📎 Note

Ouvrir un fichier → entrée dans une table, flux d'entrée permettant de modifier / lire

ⓘ Info

En mémoire centrale : table des fichiers ouverts.

Par processus : table des fichiers ouverts.

B - Flots d'entrée / sortie

ⓘ Info

Tout est considéré comme un pseudo fichier. Les **flots** (stream) préouverts par défaut :

- entrée standard
- sortie standard
- sortie standard d'erreur

C - API C et OCaml

📋 En C :

action	fonction
ouverture	<code>fopen</code>
fermeture	<code>fclose</code>
lecture	<code>fscanf</code>
écriture	<code>printf</code>

En OCaml :

action	fonction
ouverture en lecture	<code>open_in</code>
fermeture en lecture	<code>close_in</code>
ouverture en écriture	<code>open_out</code>
fermeture en écriture	<code>close_out</code>
lecture	<code>input_line</code>
écriture	<code>output_string</code>

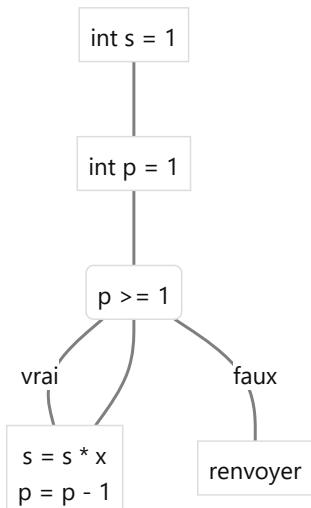
Exemple

```
int main(int argc, char** argv)
{
    assert (argc == 2);
    errno = 0;
    FILE* fd = fopen(argv[1], "r");
    if (fd == NULL)
    {
        printf("Errno %s : %d (%s)\n", argv[1], errno, strerror(errno));
        return 1;
    }
    fclose(fd);
    return 0;
}
```

Chapitre 18 - Graphes de flots de contrôle

- I - Exemple introductif
- II - Converture d'un GFC par des tests

I - Exemple introductif



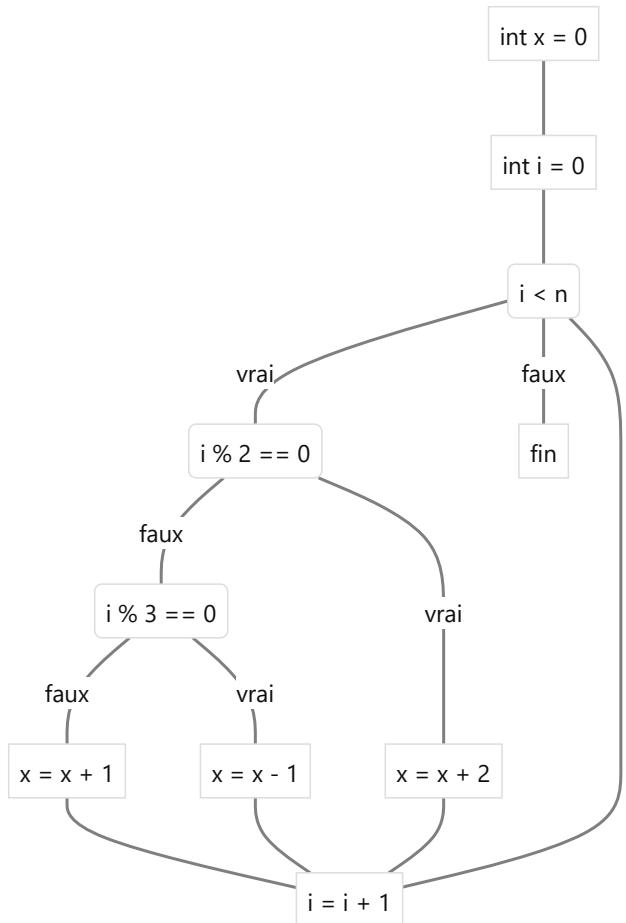
C

```
int mystere(int x)
{
    int s = 1;
    int p = 1;
    while (p >= 1)
    {
        s = s * x;
        p = p - 1;
    }
    return s;
}
```

II - Converture d'un GFC par des tests

C

```
int x = 0;
int i = 0;
while (i < n)
{
    if (i % 2 == 0)
    {
        x = x + 2;
    }
    else
    {
        if (i % 3 == 0)
        {
            x = x - 1;
        }
        else
        {
            x = x + 1;
        }
    }
    i = i + 1;
}
```



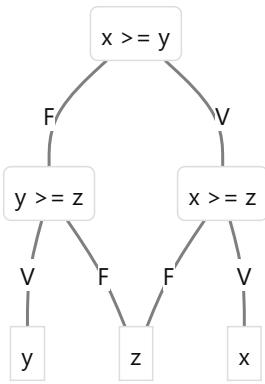
```

C

int max3(int a, int b, int c)
{
    int tmp;
    if (a > b)
    {
        tmp = a;
    } else
    {
        tmp = b;
    }
    if (tmp < c)
    {
        tmp = c;
    }
    return tmp;
}

int max3(int a, int b, int c)
{
    return (a > b ? a : b) < c ? c : (a > b ? a : b);
}

```



C

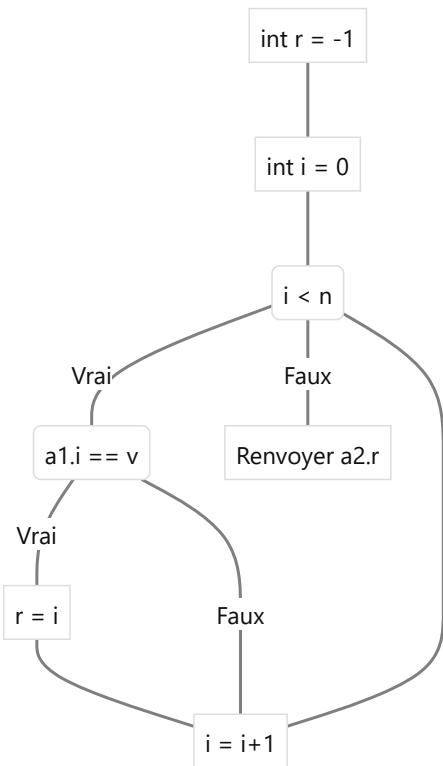
```

int corresponding_v(int v, int* a1, int* a2, int n)
{
    int r = -1;
    int i = 0;
    while (i < n)
    {
        if (a1[i] == v)
        {
            r = i;
        }
        i = i + 1;
    }
    return a2[r];
}
  
```

1. Que fait le programme ?

Le programme renvoie la valeur du tableau a2 à l'indice de la dernière occurrence de la valeur v dans a1.

2. Construire le GFC



3. Indiquer le chemin obtenu avec v=1, a1 = {3,5,1}, a2 = {5,1,3}, n = 3

r=-1, i=0, a[i] ≠ 1, i=1, a[i] ≠ 1, i=2, a[i] = 1, r=i, i=3, i=n donc renvoie a2[i]=3

4. Est ce que cette entrée est suffisant ?

OUI

5. Donner un jeu de tests courrants, le GFC si possible

n=0, v=0 avec les mêmes a1 et a2

Chapitre 19 - Recherche textuelle

- I - Introduction
- II - Algorithme naïf
- III - Algorithme de Boyer-Moore
 - A - Variante de Horspool
 - B - Version de Boyer-Moore
 - C - Version simplifiée
- IV - Algorithme de Rabin-Karp
- V - Algorithme de LZW
 - A - Compression
 - B - Décompression

I - Introduction

Définition

Un **alphabet** est un ensemble fini et non vide d'éléments, appelés **symboles**, **lettres** ou **caractères** noté Σ

Exemple

$\Sigma = \{0, 1\}$
 $\Sigma = \{a, b, c\}$
 $\Sigma = \{a\}$

Définition

Un **mot** sur un alphabet Σ est une suite finie d'éléments de Σ . La concaténation de deux mots x et y s'écrit xy ou $x \circ y$. Le mot vide est représenté ε .

Définition

Un mot x est **préfixe** d'un mot y s'il existe un mot z tel que $y = xz$.
Un mot x est **suffixe** d'un mot y s'il existe un mot z tel que $y = zx$.

Définition

Un **texte** est un tableau d'éléments de caractères appartenant à un alphabet.

Recherche d'une chaîne de caractères

Soit T un texte de longueur n et P un **motif**, un texte de longueur $m \leq n$.
Le problème est de trouver toutes les **occurrences** de P dans T .

Définition

Le motif P apparaît avec un **décalage** s si :

$$\begin{cases} s \in [[0, n - m]] \\ \forall i \in [[0, m - 1]], T[s + i] = P[i] \end{cases}$$

II - Algorithme naïf

Consigne

Ecrire un algorithme naïf qui permet de rechercher un **motif** dans un **texte**. On renverra une liste chaînée de décalages.

Algorithm Algorithme naïf

```
function RECHERCHENAIVE( $P, T$ )
     $l \leftarrow$  ListeVide()
     $n \leftarrow$  Longueur( $T$ )
     $m \leftarrow$  Longueur( $P$ )
    for  $s \leftarrow 0$  to  $n - m$  do
        if  $s$  est un décalage then
            |  $l \leftarrow$  AjoutEnTete( $l, s$ )
    return  $l$ 
```

Exercice

Déterminer la complexité de l'algorithme naïf en pire cas.

Exhiber une situation dans laquelle la complexité est atteinte.

$O(m \times (n - m + 1))$

Exemple

Déterminer le nombre de comparaison de caractères effectuées en pratique dans les cas suivants :

1. $P = aab$ et $T = acaabc$
2. $P = bra$ et $T = abracadabra$
3. $P = kayakaky$ et $T = kaakokkaaokaykyakaya\dots$

III - Algorithme de Boyer-Moore

Lien

[Algorithme de Boyer-Moore](#) sur Wikipédia

Explication ?

[Cours NSI](#)

A - Variante de Horspool

Définition

Une **table de décalage** est une association entre une lettre et sa distance à la dernière lettre du motif.

III - Algorithme de Boyer-Moore

A. Version de Horspool

Transcription de l'algorithme écrit ensemble mardi 20 :

Algorithme : Algorithme de Boyer-Moore, version de Horspool

```

Entrées : P : un motif; T : un texte
Résultat : Renvoie la liste de toutes les occurrences du motif P dans le texte T. Une occurrence est représentée
           par l'indice du début du motif dans le texte.

1 Initialisations :
2 début
3   | d ← TABLEDECALAGES(P)
4   | m ← TAILLE(P); n ← TAILLE(T); l ← LISTEVIDE()
5 fin
6 s ← 0
7 tant que s < n - m + 1 faire
8   | j ← m - 1
9   | tant que j ≥ 0 et T[s + j] = T[j] faire
10  |   | j ← j - 1
11  | fin
12  | si j = -1 alors
13  |   | l ← AJOUTERLISTE(l, s)
14  |   | s ← s + 1                                // Test au décalage suivant
15  | sinon
16  |   | s ← DÉCALERMOTIF(j, m, T[j + s], d)
17  | fin
18 fin
19 Renvoyer (l)
```

L'objectif de l'opération DÉCALERMOTIF est de décaler le plus possible le motif, en déplaçant le motif de manière à faire correspondre la lettre qui était différente avec la lettre correspondant dans le motif.

En pratique, nous avions expliqué les points suivants :

- si la lettre $T[s + j]$ qui pose problème à l'indice j n'apparaît pas du tout dans le motif : $s \leftarrow s + j + 1$
- si la lettre différente apparaît dans le motif et s'est trouvée à $j = m - 1$ alors $s \leftarrow s + d[T[s + j]]$

Dans le cas général, il faut donc appliquer $s \leftarrow s + d[T[s + j]] - (m - 1 - j)$.

Si cette valeur est inférieure à 1, alors il faut décaler de 1.

B - Version de Boyer-Moore

Principe

- table de décalage qui indique la position de chaque caractère de l'alphabet par rapport à la fin
- table de décalage de la taille du motif qui indique en fonction d'un j dans le motif la position k à laquelle on peut retrouver le sous-motif

Exemple

```
motif ABYXCDEYX
j 123456789
k -8 -7 -6 -5 -4 -3 2 -1 8
décalage 9 9 9 9 9 5 9 1
```

C - Version simplifiée

Bug

Oups... je n'ai pas noté.

IV - Algorithme de Rabin-Karp

Principe

Calculer une **empreinte** (un *hash*) du motif.
Comparer l'empreinte du motif à l'empreinte de la sous-chaîne considérée

Remarque

$$h(c_0 c_1 \dots c_{m-1}) = \sum_{0 \leq j < m} B^{m-1-j} \times c_j$$

Exercice

Soit c_i le caractère i du texte t . Montrer que :

$$h(c_{i+1} c_{i+2} \dots c_{i+m}) = h(c_i c_{i+1} \dots c_{i+m-1}) - B^{m-1} c_i + c_{i+m}$$

Exercice

En déduire que le hachage des m caractères à la position $i+1$ peut se calculer en temps constant à partir du hachage des m caractères à la position i .

V - Algorithme de LZW

A - Compression

```
t[x] = x Pour tout x appartenant à [0,255]
m <- "" // séquence vide
quand on lit un octet x :
    si m x est dans la table :
        m <- mx
    sinon
        emettre le code associé à m ajouter l'entier mx dans la table
        m <- x
    emettre t[m]
```

B - Décompression

```
initialisation le dictionnaire inverse t code -> octet
maintenir une variable c qui retient le dernier code lu
pour changer chaque code n lu :
    si n < |t|, alors on a t[n] = xm
        écrire xm
    si n = |t| alors écrire t[c]x sachant que x = t\[c]\[0]
A partir du 2 ème code lu -> ajouter t[c]x à la table
c -> n
```