



# Essentiel d'informatique

2023/2024

Victor Sarrazin




*Bienvenue dans l'essentiel d'informatique de mes cours de prépa. Ce document a pour objectif de contenir l'intégralité des cours d'informatique afin de les condenser et de les adapter.*

*Bonne lecture...*

# Sommaire


## Introduction aux langages :

 I Introduction au C	3
 II Introduction au OCaml	8

## Structures de données :

 I Piles, files, dictionnaires	21
 II Arbres	30
 III Graphes	36

## Informatique théorique :

 I Bases	38
 II Récursion	42
 III Stratégies algorithmiques	43
 IV SQL	44
 V Algorithmes des textes	49
 VI Formules propositionnelles	51



# Introduction



## I Introduction au C



### I.1 Variables

Pour définir une variable en C on a la syntaxe suivante : type nom

```
1 int mango = 0;
```



Il est possible de définir plusieurs variables en même temps :

```
1 int banana = apple = 12;  
2 int pear, orange = 14; // pear est non initialisée et orange vaut 14  
3 int potato = 12, tomato = 14; // potato vaut 12 et tomato vaut 14
```



### I.2 Opérateurs

On a les opérations arithmétiques suivantes :

Opération	En C
Addition	a + b
Soustraction	a - b
Multiplication	a * b
Division	a / b
Modulo	a % b

On peut utiliser +=, -=, \*=, /= et %= pour faire des opérations arithmétiques et des assignments

De plus on peut utiliser ++ et -- pour incrémenter/décrémenter

Les comparaisons se font avec >, >=, <=, < et ==.

On a des opérateurs binaires && (et logique), || (ou logique) et ! (négation de l'expression suivante)



#### Attention :

Le && est prioritaire sur le ||



### I.3 Structures de contrôle

Pour exécuter de manière conditionnelle, on utilise if (cond) {...} else if (...) {} ...  
{ } else { }

Ainsi le code suivant est valide :

```

1  if (x == 1) {
2      // Do code
3  } else if (x > 12) {
4      // Do code bis
5  } else {
6      // Do code ter
7  }

```



#### Attention :

En C un 0 est considéré comme false et toute autre valeur numérique true

Pour faire une boucle on peut utiliser un while (cond) {} qui exécute le code tant que la condition est valide

On peut utiliser do {} while (cond) qui exécute une fois puis tant que la condition est vérifiée

Il est aussi possible d'utiliser for (...) {}, de la manière suivante :

```

1  // De 0 à n - 1
2  for (int i = 0; i < n; i++) {
3
4  }
5
6  // De 0 à n - 1 tant que cond
7  for (int i = 0; i < n && cond; i++) {
8
9  }

```

A noter qu'en C il est possible de modifier la valeur de i et donc de sortir plus tôt de la boucle

Il est possible de sortir d'une boucle avec break, ou de passer à l'itération suivante avec continue



## I.4 Fonctions

Pour définir une fonction on écrit :

```

1  int my_func(int a, int b) {
2      // Do code
3      return 1;
4  }

```

Si on ne prend pas d'arguments on écrit int my\_func(void) {} et si on ne veut rien renvoyer on utilise void my\_func(...) {}

Ainsi pour appeler une fonction on fait :

```

1  int resp = my_func(12, 14);

```



#### Attention :

Les variables sont copiées lors de l'appel de fonction

On peut déclarer une fonction avant de donner son code mais juste sa signature avec :

```
1 int my_func(int);
```

## 1.5 Tableaux en C

Le type d'un tableau en C est `type[]` ou `* type`

Pour initialiser un tableau on a les manières suivantes :

```
1 int[4] test = {0, 1, 2, 3}; // Initialise un tableau de taille 4 avec 0,1,2,3
2 int[] test = {0, 1, 2, 3}; // Initialise un tableau avec 0,1,2,3 (avec 4
3 éléments)
int[4] test = {0, 1}; // Initialise un tableau de taille 4 avec 0,1,0,0 (les
autres valeurs sont à 0)
```

Il n'est pas obligé de donner la taille d'un tableau elle sera déterminée au moment de l'exécution



### Attention :

Si on dépasse du tableau C ne prévient pas mais s'autorise à faire n'importe quoi

Pour affecter dans une case de tableau on fait :

```
1 test[1] = test[2] // On met dans la case 1 la valeur de la case 2
```

Pour faire des tableaux de tableaux on fait :

```
1 int[4][4] test = {{0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11}, {12, 13, 14,
15}}; // Initialise un tableau de taille 4x4 avec les valeurs
2 int[4][4] test = { {0} }; // Initialise un tableau de taille 4x4 avec des 0
3 int[][4] test = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} }; // Initialise un
tableau de taille 3x4 avec les valeurs
```

Comme pour les tableaux on peut initialiser partiellement un tableau de tableaux

## 1.6 Pointeurs

Toute variable en C est une adresse mémoire, on peut donc récupérer cette adresse avec `&` :

```
1 int a = 12;
2 // b est l'adresse mémoire de a
3 int * b = &a;
```

Il est possible de récupérer la valeur d'une adresse mémoire avec `*` (**déréférencement**) :

```
1 int a = 12;
2 int * b = &a;
3 // c est la valeur de a
4 int c = *b;
```

On remarque donc que un pointeur a un type `type * var`

Il est aussi possible de prendre l'adresse d'un pointeur, ainsi on aura un type `** var` (généralisable...)

Si on ne connaît pas l'adresse d'un pointeur on peut le déclarer avec type `* var = NULL`



**Attention :**

Il ne faut SURTOUT PAS déréférencer un pointeur NULL, ou on aura une erreur de segmentation (segmentation fault)

Il est possible d'allouer de la mémoire avec `malloc` :

```
1  int * a = malloc(sizeof(int));
2
3  int * tab = malloc(4 * sizeof(int)); // Alloue un tableau de 4 éléments
4
5  int * tab2 = malloc(4 * sizeof(*tab2)); // Alloue un tableau de 4 éléments
```

L'appel à `malloc` renvoie un pointeur, et un pointeur NULL si il n'y a pas assez de mémoire (il peut donc être judicieux de vérifier si le pointeur est NULL sur des grosses allocations)

L'appel à `sizeof` renvoie la taille en octets de l'élément passé en argument, on peut passer un type ou une variable.

Après utilisation de `malloc` il est important de libérer la mémoire avec `free` quand on a fini d'utiliser la mémoire :

```
1  int * a = malloc(sizeof(int));
2
3  // Do code
4
5  free(a);
```



**Attention :**

Il est important de libérer la mémoire après utilisation pour éviter les fuites mémoires (memory leaks) ou on finit avec un bluescreen

Il est ainsi possible de créer un tableau avec un `malloc` en modifiant la taille du tableau :

```
1  int * tab = malloc(4 * sizeof(int)); // Alloue un tableau de 4 éléments
```

On pourra donc utiliser `tab[0]`, `tab[1]`, `tab[2]` et `tab[3]`...

Quand on passe un tableau à une fonction on passe un pointeur, ainsi on peut modifier le tableau dans la fonction



**Attention :**

Ainsi une fonction NE PEUT renvoyer un tableau créé normalement, il faut ABSOLUMENT renvoyer un tableau qui a été alloué avec `malloc`

Les tableaux, et notamment les cases des tableaux étant des pointeurs, on peut récupérer l'adresse d'une case de tableau avec & :

```
1  int tab[4] = {0, 1, 2, 3};
2
3  int * a = &tab[2]; // a est l'adresse de la case 2
```

Il est intéressant de noter que `tab[2]` est équivalent à `*(tab + 2)` mais que l'arithmétique des pointeurs n'est pas au programme et qu'elle permet d'avoir des erreurs plus facilement

Il est aussi possible de faire des tableaux de tableaux avec des pointeurs :

```
1  int ** tab = malloc(4 * sizeof(int *)); // Alloue un tableau de 4 pointeurs
```

Enfin on peut aussi passer une fonction en argument d'une autre fonction :

```
1  int my_func(int (*func)(int, int), int a, int b) {
2      return func(a, b);
3  } // my_func prend une fonction en argument qui prend deux entiers et renvoie un entier
```

## 1.7 Types construits

Pour définir un alias on utilise typedef :

```
1  typedef int my_type;
2
3  my_type a = 12;
```

Pour définir une structure on utilise struct :

```
1  struct my_struct {
2      int a;
3      int b;
4  };
5
6  struct my_struct s;
7  s.a = 12;
8  s.b = 14;
```

Mais ce n'est pas pratique d'écrire `struct my_struct` à chaque fois, on peut donc utiliser un alias :

```
1  typedef struct my_struct {
2      int a;
3      int b;
4  } my_struct;
5
6  my_struct s;
7  s.a = 12;
8  s.b = 14;
```

On peut aussi initialiser une structure de la manière suivante :

```
1 my_struct s = {.a = 12, .b = 14};
```

On peut ainsi faire des structures récursives :

```
1 typedef struct my_struct {  
2     int a;  
3     struct my_struct * next;  
4 } my_struct;
```

Comme en OCaml on peut définir des énumérations :

```
1 typedef enum my_enum {  
2     A,  
3     B,  
4     C  
5 } my_enum;
```

Enfin si on veut faire des types plus complexes comme `type t = A of int | B of float` (en OCaml) on peut utiliser des unions :

```
1 typedef struct my_union {  
2     enum {  
3         A,  
4         B  
5     } type,  
6     union {  
7         int a;  
8         float b;  
9     } value;  
10 } my_union;  
11  
12 my_union a = {.type = A, .value.a = 12};  
13 my_union b = {.type = B, .value.b = 12.};
```

Dans un champ union on ne peut accéder qu'à un seul champ à la fois, il faut donc connaître le type pour accéder à la bonne valeur (cela permet d'économiser de la mémoire)

## II Introduction au OCaml

### II.1 Expressions

En OCaml on retrouve les types `int`, `float` (qui correspond au double du C) et `bool`.

Pour les opérations arithmétiques **sur les entiers** on a :

Opération	En OCaml
Addition	<code>a + b</code>
Soustraction	<code>a - b</code>



Multiplication	$a * b$
Division	$a / b$
Modulo	$a \bmod b$

Pour les opérations arithmétiques **sur les flottants** on a :

Opération	En OCaml
Addition	$a +. b$
Soustraction	$a -. b$
Multiplication	$a *. b$
Division	$a /. b$
Exponentiation	$a **. b$

On notera que le modulo n'est pas défini pour les flottants et que l'exponentiation est définie pour les flottants

On dispose aussi des fonctions mathématiques classiques, comme  $\sin$ ,  $\cos$ ,  $\tan$ ,  $\exp$ ,  $\log$ ,  $\sqrt{\phantom{x}}$ ,  $\text{abs}$ ,  $\text{acos}$ ,  $\text{asin}$ ,  $\text{atan}...$  avec  $\log$  la fonction logarithme népérien

Comme en C on a les opérateurs binaires  $\&\&$  (et logique),  $\|\|$  (ou logique) et  $\text{not}$  (négation de l'expression suivante)

Pour faire des comparaisons **sur des valeurs** on a  $=$  pour l'égalité,  $<>$  pour la différence, et  $>$ ,  $>=$ ,  $<=$ ,  $<$  pour les comparaisons

#### Attention :



En OCaml un  $==$  est une comparaison de référence (d'étiquette), il ne faut pas l'utiliser pour comparer des valeurs, et de même pour  $!=$

## II.2 Typage fort

On a pu remarquer notamment sur les entiers et float que le typage est fort : aucune conversion implicite n'est faite c'est à l'utilisateur de le faire

Il n'est donc pas possible de faire  $1 +. 2$  mais il faut faire  $1. +. 2.$

Pour passer d'un type à un autre on utilise les fonctions `int_of_float`, `float_of_int`, `int_of_string`, `float_of_string...`

## II.3 Définitions

En OCaml le principe de variable n'existe pas réellement, on a des constantes, on ne peut pas modifier une variable à proprement parler

Pour faire une **définition** on utilise `let` :

```
1  let a = 12;; (* Définit a comme étant 12 *)
2
3  let b = 12 + a;; (* Définit b comme étant 24 *)
```



Il est possible de redéfinir une variable, mais on ne modifie pas la variable mais on en crée une nouvelle :

```

1  let a = 12;;
2
3  let a = a + 1;; (* a est maintenant 13 *)

```



La mémoire est gérée différemment qu'en C, par exemple avec le code suivant en C on a b qui est une copie de a :

```

1  int a = 12;
2  int b = a;

```



Alors qu'en OCaml b est une référence à a, si on modifie a on modifie b et inversement :

```

1  let a = 12;;
2  let b = a;;

```



Il est possible de définir plusieurs variables en même temps :

```

1  let a = 12 and b = 14;; (* a est 12 et b est 14 *)

```



Il est aussi possible de faire des variables locales en utilisant in :

```

1  let a = 12 and b = 14 in
2    a + b;; (* a + b vaut 26, et a et b ne sont pas accessibles en dehors du bloc *)

```



Il est bien sûr possible d'imbriquer les in :

```

1  let a = 12 in
2    let b = 14 in
3      a + b;; (* a + b vaut 26, et a et b ne sont pas accessibles en dehors du bloc *)

```



## II.4 Fonctions

Le OCaml est un langage fonctionnel, il est donc possible de définir des fonctions, de plusieurs manières différentes.

La première manière est de définir une fonction d'une manière semblable à une variable :

```

1  let sum a b = a + b;; (* Définit une fonction sum qui prend deux arguments a et b et renvoie a + b *)

```



Il existe un mot clé function (qui ne peut prendre qu'un argument) pour définir une fonction anonyme, ainsi on peut faire :

```

1  let sum = function a -> function b -> a + b;; (* Définit une fonction sum qui prend deux arguments a et b et renvoie a + b *)

```



On remarque que pour passer plusieurs arguments avec function on utilise plusieurs function, ce qui peut être fastidieux

Ainsi il existe le mot clé fun qui permet de définir une fonction de manière plus simple :

```
1 let sum = fun a b -> a + b;; (* Définit une fonction sum qui prend deux arguments a et b et renvoie a + b *)
```

Pour appeler une fonction on fait :

```
1 sum 12 14;; (* Renvoie 26 *)
```

Il faut faire attention au fait que chaque bloc est considéré comme un argument, ainsi on a les cas de figure suivants :

```
1 sum -12 12;; (* Erreur, on a -, 12 et 12 comme arguments *)  
2 sum (-12) 12;; (* Renvoie 0 *)
```

#### Attention :



Il est important de bien mettre des parenthèses pour les arguments négatifs, ou pour des appels intermédiaires

OCaml va déterminer tout seul la signature de la fonction, ainsi on peut faire :

```
1 let sum a b = a + b;; (* Définit une fonction sum qui prend deux arguments a et b et renvoie a + b *)  
2 (* sum : int -> int -> int *)
```

En analysant la signature de la fonction on peut voir que sum prend deux entiers et renvoie un entier

Mais on peut aussi faire du polymorphisme, ainsi on peut faire :

```
1 let min a b = if a < b then a else b;; (* Définit une fonction min qui prend deux arguments a et b et renvoie le minimum *)  
2 (* min : 'a -> 'a -> 'a *)
```

Ainsi ici min prend deux arguments de même type et renvoie un argument du même type (il est aussi possible de faire du polymorphisme sur plusieurs types et d'avoir des 'b, 'c...)

Il peut arriver qu'une fonction ait des effets de bord, ainsi elle peut renvoyer le type unit :

```
1 let nothing a = ();; (* Définit une fonction nothing qui prend un argument a et ne renvoie rien *)  
2 (* nothing : 'a -> unit *)
```

Il est aussi possible de ne pas prendre d'arguments :

```
1 let nothing = ();; (* Définit une fonction nothing qui ne prend pas d'arguments et ne renvoie rien *)  
2 (* nothing : unit *)
```

**Attention :**

Pour appeler une fonction qui ne prend pas d'arguments il faut mettre des parenthèses, sinon on aura une erreur (ici `nothing ()`)

Mais le mot clé `function` a un avantage : il permet de faire des **match** qui vont être des conditions sur les arguments :

```
1 let my_func a = function
2   | 0 -> 1 * a (* Si a est 0 on renvoie a *)
3   | 1 -> 2 * a (* Si a est 1 on renvoie 2a *)
4   | _ -> 3 * a;; (* Sinon on renvoie 3a *)
5 (* my_func : int -> int -> int *)
```



Ainsi le mot clé `function` avec un **match** permet de prendre un argument mais sans le nommer

Il est aussi possible de faire des motifs **gardés**, pour imposer une condition sur un motif avec `when` :

```
1 let my_func a = function
2   | 0 when a > 0 -> 1 * a (* Si a est 0 et a > 0 on renvoie a *)
3   | 0 -> -1 * a (* Si a est 0 et a <= 0 on renvoie -a *)
4   | 1 -> 2 * a (* Si a est 1 on renvoie 2a *)
5   | _ -> 3 * a;; (* Sinon on renvoie 3a *)
6 (* my_func : int -> int -> int *)
```



Il faut noter que les motifs sont examinés dans l'ordre, ainsi si on a plusieurs motifs qui correspondent on prend le premier qui correspond

Enfin il est possible de faire des fonctions récursives, pour cela on utilise le mot clé `rec` :

```
1 let rec fact = function
2   | 0 -> 1
3   | n -> n * fact (n - 1);;
4 (* fact : int -> int *)
```



## II.5 Expressions plus complexes

Si on veut faire des expressions plus complexes on peut utiliser `if ... else if ... else ...` :

```
1 let my_func a =
2   if a = 0 then
3     1 * a (* Si a est 0 on renvoie a *)
4   else if a = 1 then
5     2 * a (* Si a est 1 on renvoie 2a *)
6   else
7     3 * a;; (* Sinon on renvoie 3a *)
8 (* my_func : int -> int *)
```



Si on veut faire des opérations plus complexes entre les `if ... else` on peut utiliser `begin ... end` ou `(...)` :

```

1  let my_func a =
2      if a = 0 then
3          begin
4              let b = 1 in
5                  b * a (* Si a est 0 on renvoie a *)
6          end
7      else if a = 1 then
8          (let b = 2 in b * a) (* Si a est 1 on renvoie 2a *)
9      else
10         3 * a;; (* Sinon on renvoie 3a *)
11 (* my_func : int -> int *)

```

Il n'est pas obligé de mettre le `else ()` si on ne fait rien

Il est aussi possible de réaliser des filtrages sans le mot clé `function` :

```

1  let my_func a b =
2      match b with
3      | 0 -> 1 * a (* Si b est 0 on renvoie a *)
4      | 1 -> 2 * a (* Si b est 1 on renvoie 2a *)
5      | _ -> 3 * a;; (* Sinon on renvoie 3a *)
6 (* my_func : int -> int *)

```

On peut construire des n-uplets avec `(..., ...)`, et en OCaml on peut les déconstruire avec `let (..., ...) = ...`

A noter que les couples possèdent les fonctions `fst` et `snd` pour récupérer le premier et le second élément

```

1  let a = (12, 14);; (* a est un couple de 12 et 14 *)
2
3  print_int (fst a);; (* Affiche 12 *)

```

Il est donc possible de donner un couple à fonction ou `match` :

```

1  let my_func a =
2      match a with
3      | (0, 0) -> 1 (* Si a est (0, 0) on renvoie 1 *)
4      | (1, 0) -> 2 (* Si a est (1, 0) on renvoie 2 *)
5      | _ -> 3;; (* Sinon on renvoie 3 *)
6 (* my_func : int * int -> int *)

```

On remarque sur la signature qu'un n-uplet est défini par `type1 * type2 * ...` et qu'on peut donc définir des n-uplets de n'importe quel type (même avec des types différents)

## II.6 Exceptions

On peut vouloir lever une exception, pour cela on utilise `raise` :

```

1 let my_func a =
2   if a = 0 then
3     raise (Invalid_argument "a ne peut pas être 0")
4   else
5     1 / a;; (* Renvoie 1/a *)
6 (* my_func : int -> int *)

```

Il est possible de définir ses propres exceptions avec exception :

```

1 exception My_exception of string;; (* Définit une exception My_exception qui
   prend un argument de type string *)
2
3 raise (My_exception "Erreur");; (* Lève l'exception My_exception avec le message
   "Erreur" *)

```

Si on veut attraper une exception on utilise try ... with :

```

1 try
2   let a = 1 / 0 in
3   a (* Renvoie a *)
4 with
5 | Division_by_zero -> 0;; (* Si on a une division par zéro on renvoie 0 *)

```

Ainsi on peut utiliser un match pour attraper une exception dans le with.

## II.7 Listes

### II.7.a Créer une liste

On peut créer une liste en OCaml avec [] :

```

1 let lst = [];; (* Définit une liste vide *)
2
3 let lst = [1; 2; 3];; (* Définit une liste avec 1, 2 et 3 *)
4 (* int list *)
5
6 let lst = [[1; 2]; [3; 4]]; (* Définit une liste de listes *)
7 (* int list list *)

```



#### Attention :

Attention, on sépare les éléments de la liste avec ; et non ,

Comme en C, on ne peut mélanger les types

### II.7.b Opérations sur les listes

Pour ajouter un élément à une liste on utilise :: :

```

1 let lst = 1 :: [2; 3];; (* Ajoute 1 à la liste [2; 3] *)
2 (* int list *)

```

Il est possible de concaténer deux listes avec @ :

```

1 let lst = [1; 2] @ [3; 4];; (* Concatène [1; 2] et [3; 4] *)
2 (* int list *)

```



Cette opération est coûteuse en temps, il est donc préférable de ne pas l'utiliser pour des listes de grande taille

(On peut aussi utiliser `List.append lst1 lst2` pour concaténer deux listes)

Les listes en OCaml n'étant pas mutables, il est impossible de modifier une liste, il faut donc créer une nouvelle liste, de plus il n'est pas conseillé d'accéder à un élément d'une liste par son indice (avec la fonction `List.nth`)

Pour récupérer le premier élément d'une liste on utilise `List.hd` :

```

1 let a = List.hd [1; 2; 3];; (* a vaut 1 *)
2 (* int *)

```



Pour récupérer le reste de la liste on utilise `List.tl` :

```

1 let a = List.tl [1; 2; 3];; (* a vaut [2; 3] *)
2 (* int list *)

```



### II.7.c Fonctions sur les listes

Il est aussi possible d'utiliser des listes dans des match, ainsi on peut faire :

```

1 let rec sum = function
2   | [] -> 0 (* Si la liste est vide on renvoie 0 *)
3   | [h] -> h (* Si la liste a un seul élément on renvoie cet élément *)
4   | h::t -> h + sum t;; (* Sinon on renvoie le premier élément plus la somme du
5   reste *)
6 (* int list -> int *)

```



Ainsi on peut déconstruire une liste dans les match.

Mais on peut aussi vouloir faire des opérations sur les listes entières.

Si on veut itérer sur une liste on peut utiliser `List.iter` :

```

1 let lst = [1; 2; 3];;
2
3 List.iter (fun x -> print_int x) lst;; (* Affiche 123 *)

```



Si on veut appliquer une fonction à tous les éléments d'une liste on peut utiliser `List.map` :

```

1 let lst = [1; 2; 3];;
2
3 let lst2 = List.map (fun x -> x + 1) lst;; (* lst2 vaut [2; 3; 4] *)

```



Si on veut filtrer une liste on peut utiliser `List.filter` :

```

1 let lst = [1; 2; 3];;
2
3 let lst2 = List.filter (fun x -> x mod 2 = 0) lst;; (* lst2 vaut [2] *)

```

Si on veut vérifier un predicat sur tous les éléments d'une liste on peut utiliser `List.for_all` ( $\forall$ ) (la recherche s'arrête dès qu'un élément ne vérifie pas le prédicat) :

```

1 let lst = [1; 2; 3];;
2
3 let b = List.for_all (fun x -> x mod 2 = 0) lst;; (* b vaut false *)

```

Si on veut savoir si un élément de la liste vérifie un prédicat on peut utiliser `List.exists` ( $\exists$ ) :

```

1 let lst = [1; 2; 3];;
2
3 let b = List.exists (fun x -> x mod 2 = 0) lst;; (* b vaut true *)

```

Si on veut récupérer le premier élément qui vérifie un prédicat on peut utiliser `List.find` (erreur `Not_found` si aucun élément ne vérifie le prédicat) :

```

1 let lst = [1; 2; 3];;
2
3 let a = List.find (fun x -> x mod 2 = 0) lst;; (* a vaut 2 *)

```

On peut aussi vouloir faire des appels récurrents sur une liste, ainsi on a deux possibilités (`('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc`) :

Si on veut appliquer `f (f (f ... (f x)))` on peut utiliser `List.fold_left` :

```

1 let lst = [1; 2; 3];;
2
3 let a = List.fold_left (fun acc x -> acc + x) 0 lst;; (* a vaut 6 *)

```

Si on veut appliquer `f x (f x (f x ... (f init x)))` on peut utiliser `List.fold_right` (`('a -> 'acc -> 'acc) -> 'a list -> 'acc -> 'acc`) :

```

1 let lst = [1; 2; 3];;
2
3 let a = List.fold_right (fun x acc -> acc + x) lst 0;; (* a vaut 6 *)

```



#### Attention :

On remarque que l'ordre des arguments n'est pas le même entre `List.fold_left` et `List.fold_right`

Si on veut savoir si un élément est dans une liste on peut utiliser `List.mem` :

```

1 let lst = [1; 2; 3];;
2
3 let b = List.mem 2 lst;; (* b vaut true *)

```



Si on veut trier une liste on peut utiliser `List.sort` avec une fonction de comparaison :

```
1 let lst = [3; 2; 1];;
2
3 let lst2 = List.sort compare lst;; (* lst2 vaut [1; 2; 3] *)
```

La fonction de comparaison doit renvoyer un entier négatif si le premier élément est plus petit, un entier positif si le premier élément est plus grand et 0 si les deux éléments sont égaux, et `compare` est une fonction prédéfinie qui fait cela en OCaml

## II.8 Types construits

On peut créer des types construits en OCaml, on a 2 différents types de types construits : les **types somme** (unions ou énumérations) et les **types produit** (structures)

Pour définir un type somme on utilise `type` :

```
1 type fruit = Apple | Banana | Pear | Orange;; (* Définit un type fruit qui peut
être Apple, Banana, Pear ou Orange *)
```

Ainsi on a créé un type `fruit` qui peut être soit `Apple`, soit `Banana`, soit `Pear`, soit `Orange`

Mais on peut vouloir ajouter des informations à un type somme, par exemple la quantité de fruits :

```
1 type basket = Fruit of int | Empty;; (* Définit un type fruit qui peut être
Apple, Banana, Pear, Orange ou Fruit avec une quantité *)
2
3 let empty = Empty;; (* Définit un panier vide *)
4 let my_basket = Fruit 12;; (* Définit un panier avec 12 fruits *)
```

Il est possible de définir des types produits, pour cela on utilise `type` :

```
1 type point = { x: int; y: int };; (* Définit un type point qui a deux champs x et
y *)
2
3 let origin = { x = 0; y = 0 };; (* Définit un point d'origine *)
4 print_int origin.x;; (* Affiche 0 *)
5 print_int origin.y;; (* Affiche 0 *)
```

Les champs définis sont immuables, il n'est pas possible de les modifier si ils ne sont pas déclarés comme mutable :

```
1 type point = { mutable x: int; mutable y: int };; (* Définit un type point qui a
deux champs x et y mutables *)
2
3 let origin = { x = 0; y = 0 };; (* Définit un point d'origine *)
4 origin.x <- 12;; (* Modifie la valeur de x *)
```

On note l'utilisation de `<-` pour modifier un champ

Il est possible de définir des types récurifs, par exemple une liste :

```

1  type tree = Leaf | Node of tree * tree;; (* Définit un type arbre qui peut être
2                                     une feuille ou un noeud avec deux sous arbres *)
3
3  let tree = Node (Leaf, Node (Leaf, Leaf));; (* Définit un arbre avec une feuille
                                         et un noeud avec deux feuilles *)

```

## II.9 Programmation impérative

### II.9.a Blocs d'instructions

On peut effectuer plusieurs opérations à la suite en OCaml, pour cela on utilise ; :

```

1  print_int 12; print_string " "; print_int 14;; (* Affiche 12 14 *)

```

Si on définit une fonction avec plusieurs expressions on peut les séparer avec ;, et la valeur de retour sera la dernière expression :

```

1  let my_func a =
2      print_int a;
3      print_string " ";
4      print_int (a + 1);
5      print_newline ();;
6  (* int -> unit *)

```

Il est préférable que les expressions soient de type unit pour éviter des erreurs (on aura un avertissement si ce n'est pas le cas)

Comme vu précédemment si on veut utiliser plusieurs instructions dans un if ... then ... else ... on doit utiliser begin ... end :

```

1  let my_func a =
2      if a = 0 then
3          begin
4              print_int 1;
5              print_newline ()
6          end
7      else
8          begin
9              print_int a;
10             print_newline ()
11         end;;
12  (* int -> unit *)

```

De même, si on veut imbriquer des match on doit utiliser begin ... end :

```

1  let my_func a =
2      match a with
3      | 0 -> begin
4          print_int 1;
5          print_newline ()
6      end
7      | _ -> begin
8          print_int a;
9          print_newline ()
10     end;;
11 (* int -> unit *)

```

Il ne faut pas oublier le caractère local des variables, ainsi si on définit une variable dans un bloc elle ne sera pas accessible en dehors de ce bloc

### II.9.b Références

En OCaml on ne peut modifier les définitions, ainsi on va utiliser des références pour modifier des valeurs.

Pour définir une référence on utilise `ref` :

```

1  let a = ref 12;; (* Définit une référence à 12 *)

```

Pour accéder à la valeur d'une référence on utilise `!` :

```

1  print_int !a;; (* Affiche 12 *)

```

Pour modifier la valeur d'une référence on utilise `:=` :

```

1  a := 14;; (* Modifie la valeur de la référence à 14 *)

```

C'est avec les références que `==` et `!=` sont définis, ils comparent les références et non les valeurs

Le type d'une référence est `'a ref`, ainsi on peut avoir des références de n'importe quel type

### II.9.c Boucles

Si on veut faire une boucle on peut utiliser `for` :

```

1  for i = 0 to 10 do
2      print_int i;
3      print_string " "
4  done;; (* Affiche 0 1 2 3 4 5 6 7 8 9 10 *)

```

Le `for` est inclusif, ainsi `for i = 0 to 10` va de 0 à 10 inclus, et on ne peut pas modifier `i` dans la boucle (il est redéfini à chaque itération)

Si on veut descendre on peut utiliser `downto` :

```

1  for i = 10 downto 0 do
2      print_int i;
3      print_string " ";
4  done;; (* Affiche 10 9 8 7 6 5 4 3 2 1 0 *)

```

Il est aussi possible de faire une boucle avec `while cond do ... done` :

```

1  let i = ref 0 in
2  while !i <= 10 do
3      print_int !i;
4      print_string " ";
5      i := !i + 1
6  done;; (* Affiche 0 1 2 3 4 5 6 7 8 9 10 *)

```

## II.10 Tableaux

Les listes ne sont pas très adaptées avec une utilisation impérative, on va donc utiliser des tableaux.

Pour définir un tableau on utilise `[| |]` :

```

1  let tab = [|1; 2; 3|];; (* Définit un tableau avec 1, 2 et 3 *)

```

Il est aussi possible de définir un tableau avec `Array.make` (le premier argument est la taille du tableau, le deuxième est la valeur par défaut) :

```

1  let tab = Array.make 3 0;; (* Définit un tableau de 3 éléments initialisés à 0 *)

```

De même il est possible d'utiliser `Array.init` pour initialiser un tableau :

```

1  let tab = Array.init 3 (fun i -> i);; (* Définit un tableau de 3 éléments
    initialisés à 0, 1 et 2 *)

```

On peut obtenir la taille d'un tableau avec `Array.length` (en  $O(1)$ ) :

```

1  let a = Array.length tab;; (* a vaut 3 *)

```

Pour accéder à un élément d'un tableau on utilise `arr.(idx)` :

```

1  let a = tab.(0);; (* a vaut 1 *)

```

Pour créer un tableau bidimensionnel on utilise `Array.make_matrix` :

```

1  let tab = Array.make_matrix 3 3 0;; (* Définit un tableau de 3x3 initialisé à 0 *)
2  *
3
1  let a = tab.(0).(0);; (* a vaut 0 *)

```

# Structures de données

## I Piles, files, dictionnaires

### I.1 Listes chaînées

En OCaml on a vu les listes, qui sont des listes chaînées, c'est à dire que chaque élément pointe vers le suivant.

On pourrait vouloir les réaliser en C :

```
1 typedef struct list {  
2     int value;  
3     struct list * next;  
4 } list;
```

Pour ajouter un élément à une liste chaînée on fait :

```
1 void add(list * lst, int value) {  
2     list * new = malloc(sizeof(*new));  
3     new->value = value;  
4     new->next = lst->next;  
5     return new;  
6 }
```

Pour récupérer le premier élément d'une liste chaînée on fait :

```
1 int get(list * lst) {  
2     if (lst == NULL) { // On empêche une segmentation fault  
3         return -1; // On renvoie une valeur par défaut  
4     }  
5  
6     return lst->value;  
7 }
```

Pour récupérer le reste de la liste chaînée on fait :

```
1 list * next(list * lst) {  
2     if (lst == NULL) { // On empêche une segmentation fault  
3         return NULL;  
4     }  
5  
6     return lst->next;  
7 }
```

Pour parcourir une liste chaînée on fait :

```

1 void print(list * lst) {
2     while (lst != NULL) {
3         printf("%d ", lst->value);
4         lst = lst->next;
5     }
6 }

```

On pourra bien sûr définir une fonction pour avoir la longueur de la liste chaînée, pour insérer un élément à un indice donné, pour supprimer un élément, pour concaténer deux listes chaînées...

Une fonction intéressante est la fonction pour supprimer totalement une liste chaînée :

```

1 void free_list(list * lst) {
2     while (lst != NULL) {
3         list * tmp = lst;
4         lst = lst->next;
5         free(tmp);
6     }
7 }

```

## I.2 Piles

Les piles sont des conteneurs de données qui suivent le principe LIFO (Last In First Out), c'est à dire que le dernier élément ajouté est le premier élément sorti, comme sur une pile d'assiettes.

### I.2.a En OCaml

En OCaml on peut utiliser le module Stack pour réaliser une pile.

Pour créer une pile on fait `Stack.create` :

```

1 let stack = Stack.create ();; (* Crée une pile *)

```

Pour ajouter un élément à une pile on fait `Stack.push` :

```

1 Stack.push 12 stack;; (* Ajoute 12 à la pile *)

```

Pour récupérer le prochain élément d'une pile on fait `Stack.top` :

```

1 let a = Stack.top stack;; (* a vaut 12 *)

```

Pour récupérer et supprimer le prochain élément d'une pile on fait `Stack.pop` :

```

1 let a = Stack.pop stack;; (* a vaut 12 *)

```



#### Attention :

Il faut faire attention à ne pas utiliser `Stack.top` ou `Stack.pop` sur une pile vide, sinon on aura l'erreur `Stack.Empty`

On peut aussi vérifier si une pile est vide avec `Stack.is_empty` :

```
1 let b = Stack.is_empty stack;; (* b vaut true *)
```

## I.2.b En C

Pour créer une pile en C on va voir 2 méthodes.

Premièrement on peut utiliser une liste chaînée :

```
1 typedef struct cell {  
2     int value;  
3     struct cell * next;  
4 } cell;  
5 typedef struct stack {  
6     cell * top;  
7 } stack;
```

On définit alors les fonctions suivantes :

```
1 stack * create_stack() {  
2     stack * s = malloc(sizeof(*s));  
3     s->top = NULL;  
4     return s;  
5 }
```

```
1 bool is_empty(stack * s) {  
2     return s->top == NULL;  
3 }
```

```
1 void push(stack * s, int value) {  
2     cell * new = malloc(sizeof(*new));  
3     new->value = value;  
4     new->next = s->top;  
5     s->top = new;  
6 }
```

```
1 int top(stack * s) {  
2     if (is_empty(s)) { // Très important pour éviter une segmentation fault  
3         return -1; // On renvoie une valeur par défaut  
4     }  
5  
6     return s->top->value;  
7 }
```

```

1  int pop(stack * s) {
2      if (is_empty(s)) { // Très important pour éviter une segmentation fault
3          return -1; // On renvoie une valeur par défaut
4      }
5
6      int value = s->top->value;
7      cell * tmp = s->top;
8      s->top = s->top->next;
9      free(tmp); // /\ Il faut libérer la mémoire
10     return value;
11 }

```

On peut faire la fonction pour supprimer la pile comme pour les listes chaînées.

Mais il est aussi possible de réaliser une pile avec un tableau dynamique : pour cela on va doubler la taille du tableau à chaque fois que la taille est atteinte.

```

1  typedef struct stack {
2      int * values;
3      int size;
4      int capacity;
5  } stack;

```

Ainsi les fonctions s'adaptent :

```

1  stack * create_stack() {
2      stack * s = malloc(sizeof(*s));
3      s->values = malloc(4 * sizeof(int));
4      s->size = 0;
5      s->capacity = 4;
6      return s;
7  }

```

```

1  bool is_empty(stack * s) {
2      return s->size == 0;
3  }

```

La fonction push est un peu plus complexe, car il faut doubler la taille du tableau si la capacité est atteinte :



```

1 void push(stack * s, int value) {
2     if (s->size == s->capacity) {
3         s->capacity *= 2;
4         int * new_values = malloc(s->capacity * sizeof(int));
5
6         for (int i = 0; i < s->size; i++) {
7             new_values[i] = s->values[i];
8         }
9
10        free(s->values);
11        s->values = new_values;
12    }
13
14    s->values[s->size] = value;
15    s->size++;
16 }

```

```

1 int top(stack * s) {
2     if (is_empty(s)) { // Très important pour éviter une segmentation fault
3         return -1; // On renvoie une valeur par défaut
4     }
5
6     return s->values[s->size - 1];
7 }

```

```

1 int pop(stack * s) {
2     if (is_empty(s)) { // Très important pour éviter une segmentation fault
3         return -1; // On renvoie une valeur par défaut
4     }
5
6     int value = s->values[s->size - 1];
7     s->size--;
8     return value;
9 }

```

Ainsi on a 2 approches différentes pour créer une pile en C, une avec une liste chaînée et une avec un tableau dynamique.

## I.3 Files

Les files sont des conteneurs de données qui suivent le principe FIFO (First In First Out), c'est à dire que le premier élément ajouté est le premier élément sorti, comme dans une file d'attente.

### I.3.a En OCaml

En OCaml on peut utiliser le module `Queue` pour réaliser une file.

Pour créer une file on fait `Queue.create` :

```

1 let queue = Queue.create ();; (* Crée une file *)

```

Pour ajouter un élément à une file on fait `Queue.push` :

```
1 Queue.push 12 queue;; (* Ajoute 12 à la file *)
```



Pour récupérer le prochain élément d'une file on fait `Queue.top` :

```
1 let a = Queue.top queue;; (* a vaut 12 *)
```



Pour récupérer et supprimer le prochain élément d'une file on fait `Queue.pop` :

```
1 let a = Queue.pop queue;; (* a vaut 12 *)
```



#### Attention :



Il faut faire attention à ne pas utiliser `Queue.top` ou `Queue.pop` sur une file vide, sinon on aura l'erreur `Queue.Empty`

Pour vérifier si une file est vide on fait `Queue.is_empty` :

```
1 let b = Queue.is_empty queue;; (* b vaut true *)
```



### I.3.b En C

Pour réaliser une file en C, on pourrait faire un tableau dynamique, mais on peut aussi faire une liste doublement chaînée.

```
1 typedef struct cell {  
2     int value;  
3     struct cell * next;  
4     struct cell * prev;  
5 } cell;  
6 typedef struct queue {  
7     cell * front;  
8     cell * back;  
9 } queue;
```



On définit alors les fonctions suivantes :

```
1 queue * create_queue() {  
2     queue * q = malloc(sizeof(*q));  
3     q->front = NULL;  
4     q->back = NULL;  
5     return q;  
6 }
```



```
1 bool is_empty(queue * q) {  
2     return q->front == NULL;  
3 }
```



Pour l'opération d'ajout on ajoute un nouvel élément à la fin de la file, ainsi on le met à la fin de la liste chaînée :

```

1 void push(queue * q, int value) {
2     cell * new = malloc(sizeof(*new));
3     new->value = value;
4     new->next = NULL;
5     new->prev = q->back;
6
7     if (q->back != NULL) { // Si la file n'est pas vide
8         q->back->next = new;
9     }
10
11     q->back = new;
12
13     if (q->front == NULL) { // Si la file est vide
14         q->front = new;
15     }
16 }

```

Pour récupérer le prochain élément de la file on prend le premier élément de la liste chaînée :

```

1 int top(queue * q) {
2     if (is_empty(q)) { // Très important pour éviter une segmentation fault
3         return -1; // On renvoie une valeur par défaut
4     }
5
6     return q->front->value;
7 }

```

Pour récupérer et supprimer le prochain élément de la file on prend le premier élément de la liste chaînée et on le supprime :

```

1 int pop(queue * q) {
2     if (is_empty(q)) { // Très important pour éviter une segmentation fault
3         return -1; // On renvoie une valeur par défaut
4     }
5
6     int value = q->front->value;
7     cell * tmp = q->front;
8     q->front = q->front->next;
9
10    if (q->front == NULL) { // Si la file est vide
11        q->back = NULL;
12    } else {
13        q->front->prev = NULL;
14    }
15
16    free(tmp); // /\ Il faut libérer la mémoire
17    return value;
18 }

```

On peut faire la fonction pour supprimer la file comme pour les listes chaînées.

## 1.4 Dictionnaires

Les dictionnaires sont des conteneurs de données qui associent une clé à une valeur pour permettre une recherche rapide.

On parle de **table de hachage** pour réaliser un dictionnaire, on va donc utiliser une fonction de hachage pour associer une clé à un indice.

En OCaml on utilise la fonction `Hashtbl.hash` pour obtenir le hachage d'une clé.

En C le hachage est souvent réalisé avec une fonction de hachage codée à la main, par exemple :

```
1  uint32_t hash(char* s) {  
2      uint32_t r = 0;  
3      for (int i=0; s[i] != '\0'; ++i) {  
4          r = (r+(r<<5)) ^ s[i]; // 33 = 1 + 2^5  
5      }  
6      return r;  
7  }
```

### 1.4.a En OCaml

En OCaml on peut utiliser le module `Hashtbl` pour réaliser un dictionnaire.

Pour créer un dictionnaire on fait `Hashtbl.create` :

```
1  let dict = Hashtbl.create 97;; (* Crée un dictionnaire *)
```



#### Attention :

Il est important de donner une taille première au dictionnaire

Pour ajouter un élément à un dictionnaire on fait `Hashtbl.add` :

```
1  Hashtbl.add dict "key" 12;; (* Ajoute 12 à la clé "key" *)
```

Pour récupérer un élément d'un dictionnaire on fait `Hashtbl.find` :

```
1  let a = Hashtbl.find dict "key";; (* a vaut 12 *)
```



#### Attention :

Il faut faire attention à ne pas utiliser `Hashtbl.find` sur une clé qui n'existe pas, sinon on aura l'erreur `Not_found`

Pour vérifier si une clé est dans un dictionnaire on fait `Hashtbl.mem` :

```
1  let b = Hashtbl.mem dict "key";; (* b vaut true *)
```

Pour supprimer un élément d'un dictionnaire on fait `Hashtbl.remove` :

```
1 Hashtbl.remove dict "key";; (* Supprime la clé "key" *)
```



### I.4.b En C

Pour implémenter un dictionnaire en C on va utiliser une liste chaînée, et une fonction de hashage hash préalablement définie.

On va considérer un tableau de listes chaînées, pour éviter que les recherches soient trop longues.

Ainsi dans la case  $i$  du tableau on aura une liste chaînée de tous les éléments ayant le hachage  $i \bmod n$ .

```
1 typedef struct cell {
2     char * key;
3     int value;
4     struct cell * next;
5 } cell;
6 typedef struct dict {
7     cell ** values;
8     int size;
9     int nb_keys;
10 } dict;
```



On ne s'attardera pas ici sur l'augmentation de la taille du tableau, mais on peut imaginer que quand beaucoup d'éléments sont dans le dictionnaire on perd en efficacité, donc on va doubler la taille du tableau et réinsérer tous les éléments à leur nouvelle place.

On définit alors les fonctions suivantes :

```
1 dict * create_dict(int size) {
2     dict * d = malloc(sizeof(*d));
3     d->values = malloc(size * sizeof(cell*));
4     d->size = size;
5     d->nb_keys = 0;
6
7     for (int i = 0; i < size; i++) {
8         d->values[i] = NULL;
9     }
10
11     return d;
12 }
```



```

1 char * find(dict * d, char * key) {
2     uint32_t h = hash(key) % d->size; // On calcule le hachage modulo la taille du
    tableau
3     cell * c = d->values[h];
4
5     while (c != NULL) { // On regarde toute la liste chaînée
6         if (strcmp(c->key, key) == 0) {
7             return c->value;
8         }
9
10        c = c->next;
11    }
12
13    return NULL;
14 }

```

```

1 void add(dict * d, char * key, int value) {
2     uint32_t h = hash(key) % d->size; // On calcule le hachage modulo la taille du
    tableau
3     cell * c = d->values[h];
4
5     while (c != NULL) { // On regarde toute la liste chaînée
6         if (strcmp(c->key, key) == 0) {
7             // On a trouvé la clé, on modifie la valeur
8             c->value = value;
9             return;
10        }
11
12        c = c->next;
13    }
14
15    // On ajoute un élément en tête de liste
16    cell * new = malloc(sizeof(*new));
17    new->key = key;
18    new->value = value;
19    new->next = d->values[h];
20    d->values[h] = new;
21    d->nb_keys++;
22 }

```

Ces fonctions sont un peu complexes, mais elles permettent de réaliser un dictionnaire en C.

## II Arbres

 A faire À venir

### II.1 Définitions

Un **arbre** est une collection d'éléments appelés **noeuds** reliés par des **liens**. Les noeuds peuvent porter des étiquettes.

Si un lien va d'un noeud  $\alpha$  vers un noeud  $\beta$ , on dit que  $\alpha$  est le **parent** de  $\beta$  et que  $\beta$  est un **enfant** de  $\alpha$ .

Le noeud qui n'a pas de parent est appelé **racine** de l'arbre, tandis que les noeuds qui n'ont pas d'enfants sont appelés **feuilles**. Les autres noeuds sont appelés **noeuds internes**.

On parle de **descendant** d'un noeud  $\alpha$  pour tout noeud  $\beta$  tel qu'il existe un chemin de  $\alpha$  à  $\beta$ .

On parle d'**antécédent** d'un noeud  $\beta$  pour tout noeud  $\alpha$  tel qu'il existe un chemin de  $\alpha$  à  $\beta$ .

On appelle **branche** une suite de noeuds reliés par des liens.

On appelle **sous-arbre** d'un noeud  $\alpha$  l'arbre formé par  $\alpha$  et tous ses descendants.

On appelle **arité** d'un noeud le nombre de ses enfants.

On appelle **squelette** d'un arbre l'arbre obtenu en supprimant les étiquettes.

On note  $|A|$  la **taille** d'un arbre  $A$ , c'est à dire le nombre de noeuds.

La **profondeur** d'un noeud est la longueur du chemin de la racine à ce noeud.

On note  $h(A)$  la **hauteur** d'un arbre  $A$ , c'est à dire la longueur du plus long chemin de la racine à une feuille, ou la profondeur du noeud le plus profond.

Un arbre est dit **parfait** si tous les noeuds internes ont le même nombre d'enfants.

#### Majorations de la hauteur/taille :

Pour un arbre binaire d'arité maximale  $a$ , on a :

$$h(A) + 1 \leq |A| \leq \frac{a^{h(A)+1} - 1}{a - 1}$$

On en déduit que  $\lfloor \log_a((a-1)|A|) \rfloor \leq h(A) \leq |A| - 1$

#### Preuve :

Il suffit d'encadrer le nombre de noeuds par le nombre de noeuds de profondeur  $p$  par :

$$\sum_{p=0}^{h(A)} 1 \leq |A| \leq \sum_{p=0}^{h(A)} a^p$$

Pour la deuxième inégalité, on reprend :

$$|A| \leq \frac{a^{h(A)+1} - 1}{a - 1}$$
$$|A|(a - 1) + 1 \leq a^{h(A)+1}$$

D'où l'inégalité en passant au logarithme en base  $a$ .

## II.2 Représentation en OCaml

Une première représentation d'un arbre en OCaml est faite avec une liste d'enfants :

```
1 type 'a tree = { element: 'a; children: 'a tree list };;
```



Mais on préfère représenter avec un couple :

```
1 type 'a tree = Node of 'a * 'a tree list;;
```



On en déduit des fonctions pour récupérer la racine, les enfants, les feuilles, la taille, la hauteur, la profondeur, le nombre de feuilles, le nombre de noeuds internes...

Les algorithmes importants sur les arbres sont le parcours en profondeur (préfixe, infixe, postfixe) et le parcours en largeur.

Pour faire un parcours en profondeur on peut faire :

```
1 let rec dfs f = function
2   | Node (a, children) -> print_string a; List.iter (dfs f) children
3   (* ('a -> unit) -> 'a tree -> unit *)
```



Ainsi on réalise un parcours en profondeur (*Depth First Search*) en appliquant la fonction  $f$  à chaque noeud, en allant le plus profondément possible puis en remontant.

Pour faire un parcours en largeur on peut faire :

```
1 let bfs f tree =
2   let queue = Queue.create () in
3   Queue.push tree queue;
4   while not (Queue.is_empty queue) do
5     let Node (a, children) = Queue.pop queue in
6     f a;
7     List.iter (fun x -> Queue.push x queue) children
8   done;;
9   (* ('a -> unit) -> 'a tree -> unit *)
```



## II.3 Arbres binaires stricts

On appelle **arbre binaire** un arbre dont chaque noeud a au plus 2 enfants (ie l'arité est au plus 2).

On peut alors définir un arbre binaire en OCaml :

```
1 type 'a binary_tree =
2   | Leaf of 'a
3   | Node of 'a * 'a binary_tree * 'a binary_tree;;
```



On peut alors définir des fonctions pour récupérer la racine, les enfants, les feuilles, la taille, la hauteur, la profondeur, le nombre de feuilles, le nombre de noeuds internes...

Il est possible de définir de manière formelle un arbre binaire :

### Arbres binaires stricts :

Par induction structurelle on définit :

- Toute feuille  $y$  est un arbre binaire strict
- Si  $\mathcal{A}_g$  et  $\mathcal{A}_d$  sont des arbres binaires stricts, alors  $(x, \mathcal{A}_g, \mathcal{A}_d)$  est un arbre binaire strict



Ainsi on peut démontrer facilement des propriétés sur les arbres binaires stricts avec l'induction structurale, car si l'assertion est vraie pour les sous-arbres, elle est vraie pour l'arbre.

#### Nombre de feuilles :

Pour un arbre binaire strict  $\mathcal{A}$  non vide,  $f$  le nombre de feuilles,  $n$  le nombre de noeuds internes, on a  $f = n + 1$

#### Preuve :

Pour un arbre à une feuille, on a  $f = 1$  et  $n = 0$ , donc  $f = n + 1$

Soit  $\mathcal{A}$  un arbre binaire strict, avec  $\mathcal{A}_g$  et  $\mathcal{A}_d$  les sous-arbres gauches et droits, on a  $f = f_g + f_d$  et  $n = n_g + n_d + 1$

D'où  $f = f_g + f_d = (n_g + 1) + (n_d + 1) = n + 1$

Il est aussi possible de démontrer cette propriété par double comptage : on a  $n$  noeuds internes, avec chacun 2 enfants et une racine et le nombre de noeuds sans la racine est  $n + f - 1$ , d'où  $n + f - 1 = 2n$  d'où  $f = n + 1$

## II.4 Arbres binaires unaires

Il est aussi possible de définir des arbres binaires unaires, c'est à dire des arbres dont chaque noeud a 0, 1 ou 2 enfants.

En OCaml on peut définir un arbre binaire unaire :

```
1 type 'a tree =  
2   | Nil  
3   | Node of 'a * 'a tree * 'a tree;;
```



Ainsi une feuille sera un noeud avec 0 enfant.

Il est possible de définir de manière formelle un arbre binaire unaire :

#### Arbres binaires unaires :

Par induction structurale on définit :

- Nil est un arbre binaire unaire (ie l'arbre vide)
- Si  $\mathcal{A}_g$  et  $\mathcal{A}_d$  sont des arbres binaires unaires, alors  $(x, \mathcal{A}_g, \mathcal{A}_d)$  est un arbre binaire unaire

On retrouve l'encadrement de la taille d'un arbre binaire unaire :

#### Majorations de la hauteur/taille (Arbres binaires unaires) :

Pour un arbre binaire unaire on a :

$$h(A) + 1 \leq |A| \leq 2^{h(A)+1} - 1$$

### Preuve :

On peut reprendre la preuve de l'encadrement précédent ou en le faisant par induction structurelle

On retrouve aussi un résultat analogue pour le nombre de feuilles :

#### Nombre de feuilles (Arbres binaires unaires) :

Pour un arbre binaire unaire  $\mathcal{A}$  non vide,  $f$  le nombre de feuilles,  $n$  le nombre de noeuds internes, on a  $f \leq n + 1$

### Preuve :

La preuve est la même que dans le cas strict à la différence près de l'inégalité

Ainsi on peut définir les fonctions pour récupérer la racine, les enfants, les feuilles, la taille, la hauteur, la profondeur, le nombre de feuilles, le nombre de noeuds internes...

On revient rapidement sur les algorithmes de parcours en profondeur et en largeur.

On va distinguer les parcours en profondeur préfixe, infixe et suffixe :

- Le parcours en profondeur préfixe est le parcours en profondeur où on applique la fonction à la racine avant les enfants
- Le parcours en profondeur infixe est le parcours en profondeur où on applique la fonction à la racine entre les enfants
- Le parcours en profondeur suffixe est le parcours en profondeur où on applique la fonction à la racine après les enfants

Pour faire un parcours en profondeur préfixe on peut faire :

```
1 let rec dfs f = function
2   | Nil -> ()
3   | Node (a, left, right) -> f a; dfs f left; dfs f right
4   (* ('a -> unit) -> 'a tree -> unit *)
```



Pour faire un parcours en profondeur infixe ou suffixe on déplacera l'appel à la fonction  $f$ , respectivement  $\text{dfs } f \text{ left}; f \text{ a}; \text{dfs } f \text{ right}$  ou  $\text{dfs } f \text{ left}; \text{dfs } f \text{ right}; f \text{ a}$ .

L'algorithme de parcours en largeur est le même que pour les arbres sans contrainte.

Il peut être intéressant de numéroté les noeuds d'un arbre.

#### Numérotation de Sosa-Stradonitz :

On numérote les noeuds d'un arbre binaire de la manière suivante :

- La racine est numérotée 1
- Si un noeud est numéroté  $i$ , alors son fils gauche est numéroté  $2i$  et son fils droit  $2i + 1$

Cette numérotation est d'autant plus intéressante qu'elle permet de retrouver le chemin vers un noeud depuis la racine grâce à sa représentation binaire : si le bit  $i$  est à 0 alors on va à gauche, sinon à droite.

## II.5 Complexité

On va être amené à faire des opérations sur les arbres, il est donc important de connaître la complexité de ces opérations.

Pour toute fonction qui visite tous les noeuds d'un arbre un nombre constant de fois, la complexité est en  $O(|A|)$ .

Il existe des cas où les fonctions seront plus complexes, notamment si on commence à travailler avec des listes : on se rend compte qu'utiliser `::` pour ajouter un élément en tête est en  $O(1)$ , mais @ pour concaténer deux listes est en  $O(n)$ .

Prenons l'exemple d'une fonction qui transforme un arbre binaire en liste :

```
1 let rec to_list = function
2   | Nil -> []
3   | Node (a, left, right) -> to_list left @ [a] @ to_list right
```

On voit bien que si l'arbre est une branche qui descend vers la gauche la complexité va exploser car on va concaténer des listes de plus en plus grandes.

Ainsi on peut réécrire la fonction pour être en  $O(|A|)$  :

```
1 let to_lst t =
2   let rec aux lst = function
3     | Nil -> lst
4     | Node (x, lchild, rchild) -> aux (x::aux lst rchild) lchild
5   in aux [] t;;
```

On a donc une complexité en  $O(|A|)$  car on ne fait que des ajouts en tête de liste.

## II.6 En C

En C on utilisera souvent la représentation d'un arbre binaire avec une structure :

```
1 typedef struct node {
2   int value;
3   struct node * left;
4   struct node * right;
5 } node;
```

Les fonctions pour récupérer la racine, les enfants, les feuilles, la taille, la hauteur, la profondeur, le nombre de feuilles, le nombre de noeuds internes seront similaires à celles en OCaml.



### Attention :

Il est important de faire des null-checks pour éviter les segmentation faults

## II.7 Arbres binaires de recherche

Beaucoup de problèmes en informatique peuvent être résolus avec des arbres binaires de recherche.

### Arbres binaires de recherche :

On définit un arbre binaire de recherche par :

- Si l'arbre est vide, c'est un arbre binaire de recherche
- Sinon il est de la forme  $(x, \mathcal{A}_g, \mathcal{A}_d)$  avec  $\mathcal{A}_g$  et  $\mathcal{A}_d$  des arbres binaires de recherche et  $x$  une valeur telle que  $\forall y \in \mathcal{A}_g, y \leq x$  et  $\forall y \in \mathcal{A}_d, y \geq x$  (pour une certaine relation d'ordre)

On adoptera la représentation en OCaml des arbres unaires pour les arbres binaires de recherche.

Pour rechercher un élément dans un arbre binaire de recherche on peut faire :

```
1 let rec mem y = function
2   | Nil -> false
3   | Node (x, left, right) when x = y -> true (* On a trouvé l'élément *)
4   | Node (x, left, right) when x > y -> mem y left (* On va à gauche *)
5   | Node (x, left, right) -> mem y right (* On va à droite *)
6 (* 'a -> 'a tree -> bool *)
```


Cet algorithme est en  $O(h(A))$  où  $h(A)$  est la hauteur de l'arbre, donc en  $O(\log(|A|))$

On dit que  $y$  est le **successeur** de  $x$  si  $y = \sup_{\{z \in A \mid z > x\}} z$  et  $y$  est le **prédécesseur** de  $x$  si  $y = \inf_{\{z \in A \mid z < x\}} z$

## II.8 Arbres binaires de recherche équilibrés

## II.9 Tas binaires

## III Graphes

 A faire À venir

### III.1 Définitions

### III.2 En OCaml

### III.3 En C

### III.4 Étiquetage

### III.5 Parcours de graphes

### III.6 Cycles

### III.7 Recherche de plus courts chemins

#### III.7.a Graphes avec poids négatifs

Dans un graphe on dit que l'arc  $u \triangleright v$  est en **tension** si  $\delta(v) > \delta(u) + w(u, v)$

L'approche de Ford est donc d'éliminer les arcs en tension

Tant qu'il existe des arcs en tension, on traite tous les arcs de  $E$  et on traite ceux en tension, on a donc une complexité  $O(n \times p)$



# Informatique théorique



## I Bases



### I.1 Fonctions

On dit qu'une fonction a des **effets de bord** si son exécution a des conséquences sur d'autres choses que ses variables locales

Une fonction est **déterministe** si le résultat est toujours le même avec les mêmes arguments

Une fonction est dite **pure** lorsqu'elle est déterministe et sans effets de bord



### I.2 Complexité

On dit qu'un algorithme est en  $O(f(n))$  **pire cas** si il existe une constante  $k$  telle que pour tout  $n$  assez grand, le nombre d'opérations est inférieur à  $kf(n)$

On dit qu'un algorithme est en  $\Omega(f(n))$  **meilleur cas** si il existe une constante  $k$  telle que pour tout  $n$  assez grand, le nombre d'opérations est supérieur à  $kf(n)$

On dit qu'un algorithme est en  $\Theta(f(n))$  **cas moyen** si il est en  $O(f(n))$  et en  $\Omega(f(n))$

On parle alors :

- $O(1)$  pour une complexité **constante**
- $O(\log(n))$  pour une complexité **logarithmique**
- $O(n)$  pour une complexité **linéaire**
- $O(n \log(n))$  pour une complexité **quasi-linéaire**
- $O(n^2)$  pour une complexité **quadratique**
- $O(k^n)$  pour une complexité **exponentielle**



### I.3 Algorithmes de tri

En informatique on a souvent besoin de trier des listes, on a plusieurs algorithmes pour cela

#### Tri stable :

Un tri est dit **stable** si l'ordre des éléments égaux est conservé



#### I.3.a Tri par sélection

Le tri par sélection est l'algorithme le plus simple de tri, on prend le minimum et on le met en tête de liste.

Ainsi on a un invariant de boucle : la liste est triée jusqu'à l'indice  $i$

Pour l'implémenter en C on fait :

```

1 void selection_sort(int arr[], int n) {
2     for (int i = 0; i < n; i++) {
3         // Les i premiers éléments sont bien triés
4         int min_i = i;
5
6         for (int j = i+1; j<n; j++) {
7             if (arr[j] < arr(min_i)) {
8                 min_i = j;
9             }
10        }
11
12        // On échange les éléments en i et min_i
13        int tmp = arr[i];
14        arr[i] = arr[min_i];
15        arr[min_i] = tmp;
16    }
17 }

```

Le tri par sélection est en  $O(n^2)$ , on a  $n$  comparaisons pour le premier élément,  $n - 1$  pour le second, etc.

Le tri par sélection a donc comme inconvénient d'avoir une complexité quadratique et de ne pas être stable

### 1.3.b Tri bulle

Le tri bulle est un algorithme de tri simple, on compare les éléments deux à deux et on les échange si ils ne sont pas dans le bon ordre, comme des bulles qui remontent à la surface

On peut réaliser un tri pierre en descendant les éléments au lieu de les monter

Pour l'implémenter en C on fait :

```

1  let bubble_sort(int arr[], int n) {
2      for (int i = 0; i < n; i++) {
3          // Les i premiers éléments sont bien placés
4          int k_last_perm = n-1;
5          int smallest = arr[n-1];
6
7          for (int j = n-1; j > i; j--) {
8              if (arr[j-1] <= smallest) {
9                  // On change de bulle
10                 arr[j] = smallest;
11                 smallest = arr[j-1];
12             } else {
13                 // On fait descendre la bulle
14                 arr[j] = arr[j-1];
15                 k_last_perm = j - 1;
16             }
17         }
18
19         arr[i] = smallest;
20         // On n'a pas besoin de regarder les éléments entre i+1 et k_last_perm car on
21         // n'a fait aucune modification
22         i = k_last_perm + 1;
23     }
24 }

```

Le tri bulle a une complexité en  $O(n^2)$ , on a  $n$  comparaisons pour le premier élément,  $n - 1$  pour le second, etc, mais cette complexité est rarement atteinte. De plus le tri bulle est stable

### I.3.c Tri par insertion

Le tri par insertion est un algorithme de tri qui consiste à insérer un élément à sa place dans une liste triée (les éléments précédents sont déjà triés mais pas forcément à leur place définitive)

Pour l'implémenter en C on fait :

```

1  int insertion_sort(int arr[], int n) {
2      for (int i = 0; i < n; i++) {
3          // Les i premiers éléments sont bien triés
4          int j = i;
5          int elem = arr[i];
6
7          for (; j>0 && elem < arr[j-1]; j--) {
8              arr[j] = arr[j-1];
9          }
10
11         arr[j] = elem;
12     }
13 }

```

Le tri par insertion a une complexité en  $O(n^2)$ , on a  $n$  comparaisons pour le premier élément,  $n - 1$  pour le second, etc, mais cette complexité est rarement atteinte. De plus le tri par insertion est stable



### I.3.d Tri rapide

Le tri rapide est un algorithme de tri qui consiste à choisir un pivot et à partitionner la liste en deux parties, les éléments plus petits que le pivot et les éléments plus grands que le pivot, on réitère sur les deux listes

Pour l'implémenter en C on fait :

```
1 void quick_sort(int * arr, int n) {  
2     if (n <= 1) { // Déjà trié  
3         return;  
4     }  
5  
6     int pivot = partition(arr, n);  
7     quick_sort(arr, pivot);  
8     quick_sort(&arr[pivot+1], n-pivot-1);  
9 }
```

Tout l'intérêt du tri rapide est dans la fonction partition qui permet de partitionner la liste en deux parties

On utilise la partition de Lomuto, qui consiste à garder le pivot en première position, puis les éléments plus petits que le pivot, puis les éléments plus grands que le pivot et enfin ceux qui ne sont pas encore triés

```
1 int partition(int arr[], int n) {  
2     int pivot = arr[0];  
3     int p = 1;  
4  
5     for (int i = 1; i<n; i++) {  
6         if (arr[i] < pivot) {  
7             arr_swap(arr, i, p); // On échange les éléments i et p  
8             p++;  
9         }  
10    }  
11  
12    arr_swap(arr, 0, p-1); // On échange le pivot et le dernier élément plus petit  
    que le pivot  
13    return p-1;  
14 }
```

## I.4 Algorithmes classiques

### I.4.a Dichotomie

La dichotomie est un algorithme de recherche efficace : on prend le milieu de la liste et on regarde si l'élément est plus grand ou plus petit, on réitère sur la moitié de la liste etc...

Pour l'implémenter en C on fait de manière récursive :

```

1  let index(int * arr, int n, int elem) {
2      if (n == 0) {
3          return -1; // On ne peut pas trouver
4      }
5
6      int m = n/2;
7
8      if (arr[m] == elem) { // On a trouvé!
9          return m;
10     } else if (arr[m] > elem) { // L'élément se situe peut être dans la partie gauche
11         return index(arr, m, elem);
12     } else { // L'élément se situe peut être dans la partie droite
13         int idx = index(&arr[m+1], n-m-1, elem);
14
15         if (idx != -1) {
16             idx += m+1;
17         }
18
19         return idx;
20     }
21 }

```

On peut aussi faire de manière itérative :

```

1  let index(int * arr, int n, int elem) {
2      int l = 0, r = n;
3
4      while (l < r) { // On recherche dans le tableau avec deux compteurs
5          int m = (l+r)/2;
6
7          if (arr[m] == elem) { // On a trouvé!
8              return m;
9          } else if (arr[m] > elem) { // L'élément se situe peut être dans la partie
10 gauche
11     r = m;
12     } else { // L'élément se situe peut être dans la partie droite
13         l = m + 1;
14     }
15 }
16
17 return -1; // Pas trouvé!
18 }

```

L'avantage de la dichotomie est qu'elle a une complexité en  $O(\log(n))$  : elle permet donc une recherche efficace

## II Récursion

### II.1 Terminaison

### II.2 Récursion terminale

## II.3 Retour sur trace

## II.4 Programmation dynamique

# III Stratégies algorithmiques

## III.1 Algorithmes gloutons

## III.2 Diviser pour régner

Le **tri fusion** est un tri en  $\Theta(n \log(n))$ , on sépare les listes puis on les trie en interne et on fusionne les deux listes triées

Pour l'implémenter en OCaml on fait :

```
1  let rec partition = function
2    | h1::h2::t -> let l,r = partition t in h1::l, h2::r
3    | lst -> lst, [];;
4
5  let rec merge l1 l2 = match l1,l2 with
6    | (h1::t1), (h2::t2) when h1 <= h2 -> h1::(merge t1 l2)
7    | (h1::t1), (h2::t2) -> h2::(merge l1 t2)
8    | l1, [] -> l1;;
9
10 let rec fusion_sort lst = match split lst with
11   | lst, [] -> lst
12   | l1, l2 -> merge (fusion_sort l1) (fusion_sort l2)
```



Analysons l'algorithme du tri fusion, en regardant le nombre de comparaisons on retrouve une complexité en  $\Theta(n \log(n))$  pour ces étapes

Plus mathématiquement on a pour  $n \geq 2$ ,  $u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil} + \frac{n}{2} \leq u_n \leq u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil} + n$  d'où on a  $u_n = u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil} + \Theta(n)$

### A faire (Suites récurrentes d'ordre 1)

#### Suites "diviser pour régner" :

Soit  $a_1, a_2$  deux réels positifs vérifiant  $a_1 + a_2 \geq 1$  et  $(b_n)_{n \in \mathbb{N}}$  une suite positive et croissante et  $(u_n)_{n \in \mathbb{N}}$  une suite vérifiant :

$$u_n = a_1 u_{\lfloor \frac{n}{2} \rfloor} + a_2 u_{\lceil \frac{n}{2} \rceil} + b_n$$

Ainsi en posant  $\alpha = \log_2(a_1 + a_2)$ , on a :

- Si  $(b_n) = \Theta(n^\alpha)$ , alors  $(u_n) = \Theta(n^\alpha \log(n))$
- Si  $(b_n) = \Theta(n^\beta)$  avec  $\beta < \alpha$ , alors  $(u_n) = \Theta(n^\alpha)$
- Si  $(b_n) = \Theta(n^\beta)$  avec  $\beta > \alpha$ , alors  $(u_n) = \Theta(n^\beta)$

**Attention :**

A savoir que si on retombe sur une relation de récurrence connue on peut donner directement la complexité

Pour l'implémenter en C on fait de la manière suivante :



A faire (Réécrire)

1



## IV SQL



### IV.1 Généralités

En SQL on stocke des entités avec des attributs et à chaque attribut on lui associe un type

On peut définir des relations entre les différentes entités

On stocke ces entités dans des tables : dans chaque table on stocke une entité

Il est possible de garder une case vide en plaçant un NULL dans la case



### IV.2 Requêtes

Pour récupérer des données (projections) dans une table on a :

```
1  # Seulement les colonnes spécifiées
2  SELECT col1, ..., coln FROM table
3
4  # Toutes les colonnes
5  SELECT * FROM table
6
7  # Toutes les colonnes mais sans doublon
8  SELECT DISTINCT * FROM table
```



Ainsi on récupère toutes les lignes de la table avec ces projections

On peut aussi faire une sélection sur un critère :

```
1  SELECT * FROM table WHERE bool
```



Les opérations booléennes sont les suivantes :

- `col > a / col < a / col = a` pour faire des comparaisons
- `col IN (a, b, c)` pour savoir si la cellule est dans un ensemble de valeur
- `col IS NULL / IS NOT NULL` pour savoir si la cellule est nulle ou non
- `col LIKE '% Text %'` pour regarder si Text est dans la chaîne de caractère de la cellule

On peut combiner les critères avec AND/OR/NOT

Il est possible de sélectionner un attribut non projeté

Pour ordonner les résultats on ordonne en utilisant

```
1 # Triés par valeur croissante
2 SELECT * FROM table ORDER BY col
3
4 # Triés par valeur décroissante
5 SELECT * FROM table ORDER BY col DESC
```

Pour limiter le nombre de valeurs on utilise

```
1 # On prend au maximum 3 éléments
2 SELECT * FROM table LIMIT 3
3
4 # On prend au maximum 3 éléments mais sans les 2 premiers
5 SELECT * FROM table LIMIT 3 OFFSET 2
```

### IV.3 Fonctions

On peut compter le nombre d'entités qui vont être renvoyées

```
1 # Nombre d'éléments dans la table
2 SELECT COUNT(*) FROM table
```

On peut compter sur une colonne spécifique avec `COUNT(col1, ..., col2)`, les cases ne sont pas comptées si `NULL`,

Il est aussi possible de compter le nombre de valeur distinctes pour une colonne :

```
1 SELECT COUNT(DISTINCT col) FROM table
```

On peut utiliser `MAX`, `MIN`, `SUM` et `AVG` pour avoir du préprocessing, il est aussi possible d'avoir la moyenne en faisant `SUM(col)/COUNT(*)`



#### Attention :

On ne peut mélanger une colonne et une fonction dans la projection

Il est possible de grouper les valeurs

```
1 # Renvoie des groupes des valeurs de col
2 SELECT col FROM table GROUP BY col
```



#### Attention :

Il n'est pas possible d'utiliser `GROUP BY` sur des colonnes non groupées

Par contre les fonctions agissent sur chaque groupe, ainsi il est possible d'écrire

```
1 # Renvoie des groupes des valeurs de col avec le nombre d'occurrence de cette
  valeur dans la table
2 SELECT col, COUNT(*) FROM table GROUP BY col
```

Pour sélectionner des groupes on peut utiliser :

```
1 # Renvoie des groupes des valeurs de col si la valeur minimale du groupe dans la
  colonne col2 est supérieure à x avec la valeur minimale de col2 de ce groupe dans
  la table
2 SELECT col1, MIN(col2) FROM table GROUP BY col1 HAVING MIN(col2) > x
```

Les opérations sont exécutées dans cet ordre :

- WHERE
- GROUP BY
- HAVING
- ORDER BY
- LIMIT/OFFSET
- SELECT à la fin bien qu'on le mette en tête de la requête

Ainsi une clause valide est

```
1 SELECT * WHERE cond GROUP BY col HAVING cond2 ORDER BY col2 LIMIT 3 OFFSET 2
```

#### IV.4 Sous requêtes

Il est possible d'écrire une sous requête :

```
1 # Ici on sélectionne seulement les éléments donc la valeur col est supérieure à
  la valeur moyenne de col
2 SELECT * FROM table WHERE col > (SELECT AVG(col) FROM table)
```

Il est donc aussi possible d'utiliser cette syntaxe avec des IN

```
1 # Ici on va sélectionner seulement les lignes dont la valeur de col correspond à
  la condition cond
2 SELECT * FROM table WHERE col IN (SELECT DISTINCT col FROM * WHERE cond)
```

Le col AS nameBis permet de renommer une colonne

Si on reçoit un tableau, on peut sélectionner dans les réponses

```
1 # Ainsi on renvoie la moyenne d'une colonne col2 telle que ses éléments vérifient
  la condition
2 SELECT AVG(resp.colName) FROM (SELECT col1, col2 AS colName FROM table WHERE
  cond) AS resp
```

#### IV.5 Combiner les tables

Il est possible de combiner des tables

```
1 # Sélectionne dans le produit cartésien des deux tables
2 SELECT * FROM table1, table2
```

Mais en faisant ça on va avoir plein de lignes qui n'ont pas de sens, ainsi si on veut garder seulement les lignes qui nous intéressent

```

1  # Sélectionne dans le produit cartésien des deux tables seulement les éléments
2  # donc la col1 de la table 1 est le même que celui de la col 2 de la table 2
   SELECT * FROM table1, table2 WHERE table1.col1 = table2.col2

```

Mais pour éviter ça on peut aussi de manière équivalente écrire :

```

1  # On sélectionne les éléments de la table1 en ajoutant la table2 si la condition
2  # est vérifiée, le ON est donc un WHERE
   SELECT * FROM table1 JOIN table2 ON table1.col1 = table2.col2

```

Le produit cartésien n'est donc qu'une manière de jointure

On peut aussi utiliser le LEFT JOIN qui permet de garder un élément de la première table même si il n'a pas d'équivalent dans la seconde table

```

1  # On sélectionne les éléments de la table1 en concaténant les éléments dont la
2  # condition est vérifiée, et rien si il n'y a pas d'équivalent
   SELECT * FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2

```

On peut faire l'union de deux requêtes

```

1  # On a les éléments qui vérifient la cond1 ou cond2
2  SELECT * FROM table WHERE cond1 UNION SELECT * FROM table WHERE cond2

```



#### Attention :

Pour utiliser l'union il faut juste que les types sont compatibles mais pas les noms de colonne

On peut aussi faire l'intersection de deux requêtes

```

1  # On a les éléments qui vérifient la cond1 et cond2
2  SELECT * FROM table WHERE cond1 INTERSECT SELECT * FROM table WHERE cond2

```

On peut faire des différences ensemblistes avec MINUS ou EXCEPT



## IV.6 Créer une BDD

Pour créer une base de données on utilisera

```

1  CREATE TABLE IF NOT EXISTS table (
2      col1 TYPE1,
3      col2 TYPE2,
4      col3 TYPE3
5  )

```

Si on veut limiter le nombre de caractères, on peut le préciser entre parenthèses, par exemple VARCHAR(6) pour avoir des chaînes d'au plus 6 caractères

On peut définir une **clé primaire** qui ne peut avoir 2 fois la même valeur, on indiquera PRIMARY KEY après le type :

```

1 CREATE TABLE IF NOT EXISTS table (
2     col1 TYPE1 PRIMARY KEY,
3     ...
4 )

```

Les autres attributs seront dépendant de la clé primaire : si on connaît la clé primaire on peut connaître les autres valeurs associées à la liste

Si on a une clé primaire dans un GROUP BY autorisée à projeter sur tous les éléments (pas comme précédemment)

Il y a au plus une clé primaire par table, et une valeur NULL ne peut être une valeur pour cette case

On peut définir une clé étrangère qui vont être des liens entre les différentes tables

```

1 CREATE TABLE IF NOT EXISTS table (
2     ...,
3     FOREIGN KEY (col) REFERENCES table(col)
4 )

```

Il est aussi possible de modifier une table en utilisant ALTER TABLE

Pour insérer dans une table on utilise :

```

1 INSERT INTO table (col1, col2, col3) VALUES (value1, value2, value3)

```

On peut modifier un élément :

```

1 UPDATE table SET col1 = value WHERE cond

```

On peut aussi supprimer un élément :


```

1 DELETE FROM table WHERE cond

```

## IV.7 Type entités

Les types entités sont liées par des types associations

 A faire (Cardinalité)

On précise les cardinalités :

- 1, 1 en liaison avec une et une seule entité
- 1,  $n$  en liaison avec au moins une autre entité
- 0, 1 en liaison avec au plus une autre entité
- 0,  $n$  en liaison avec un nombre quelconque d'entités



## V Algorithmes des textes

### V.1 Bases

En C on représente les chaînes de caractère par des `char *` avec un `\0` à la fin de la chaîne (donc un `0` dans la dernière case)

On peut utiliser `strlen` pour connaître la longueur d'une chaîne

En OCaml on a le module `String` qui permet de manipuler les chaînes de caractères et les chaînes de caractères sont immuables

On peut concaténer des chaînes avec `^` et on peut accéder à un caractère avec `[i]`

On peut aussi utiliser `String.length` pour connaître la longueur d'une chaîne (en  $O(1)$ )

Pour lire tous les éléments d'une chaîne en C on fera :

```
1  for (int i = 0; str[i] != '\0'; i++) {  
2      // Do code  
3  }
```



En C un `char` correspond à un entier entre  $-128$  et  $127$ , ainsi on peut écrire `int a = (int) 'a'` (le cast n'est pas obligatoire) pour avoir 97

A noter que `'` est un caractère et `"` est une chaîne de caractère



#### Attention :

On ne fera pas une boucle `for` avec `strlen` car on va recalculer la longueur de la chaîne à chaque itération

### V.2 Algorithmes

Imaginons que l'on veuille trouver si une chaîne de caractères n'est constituée que de mots valides (en supposant que la fonction `is_word` existe) :



```
1 void is_sentence(char * s) {
2     if (s[0] == '\0') {
3         return;
4     }
5
6     int n = strlen(s);
7     int * arr = malloc((n+1) * sizeof(*arr));
8     arr[0] = 0;
9
10    for (int i = 1; i <= n; i++) {
11        arr[i] = -1; // On initialise à false car le malloc ne le fait pas
12        char tmp = s[i];
13        s[i] = '\0';
14        for (int j = i-1; arr[i] != -1 && j >= 0; --j) {
15            if (arr[j] != -1 && is_word(&s[j])) {
16                arr[i] = j;
17            }
18        }
19        s[i] = tmp;
20    }
21    // Le tableau arr contient l'indice du début du mot précédent (ou -1 si il n'y
    en a pas)
22    free(arr);
23 }
```

Il est intéressant de mémoriser cette fonction pour éviter de recalculer plusieurs fois la même chose

Pour déterminer si une chaîne de caractères est un mot, on a plusieurs approches, en considérant  $N$  mots et  $p$  la longueur de la chaîne :

- Approche naïve : On compare pour chaque mot  $O(N \times p)$
- Approche dichotomique : On trie les mots et on fait une recherche dichotomique  $O(p \times \log(N))$
- On utilise un *TRIE*, c'est à dire un arbre où chaque noeud est une lettre et chaque branche est un mot, on a une complexité en  $O(p)$  (selon l'implémentation de chaque noeud et de son stockage), on privilégiera de stocker dans un dictionnaire les mots. Une autre solution est de stocker tous les mots dans un dictionnaire et de regarder si le mot est dedans

### V.3 Recherche de motifs

Une recherche de motif est une recherche d'une chaîne de caractères dans une autre chaîne de caractères

On considère un motif de longueur  $p$  et un texte de longueur  $n$

Une première approche naïve est de regarder pour chaque sous-chaîne de longueur  $p$  si elle est égale au motif, on a une complexité en  $O(n \times p)$  (généralement  $O(n)$  en pratique)

## VI Formules propositionnelles

### VI.1 Bases

#### Formule propositionnelle :

On a deux constantes,  $\top$  qui est toujours vraie et  $\perp$  qui est toujours fausse

On peut les combiner avec des opérateurs logiques :

- $\wedge : a \wedge b$  est vrai si  $a$  et  $b$  sont vrais
- $\vee : a \vee b$  est vrai si  $a$  ou  $b$  est vrai
- $\neg : \neg a$  est vrai si  $a$  est faux

On définit la hauteur et la taille d'une formule propositionnelle comme la hauteur et la taille de l'arbre de la formule

Ainsi  $(\mathcal{A} \wedge \top) \vee ((\mathcal{B} \wedge \perp) \wedge \mathcal{C})$  est de hauteur 2 et de taille 5.

On a les opérations binaires suivantes :

#### Opérations binaires :

- OR  $\equiv a \vee b$  (noté  $|$ )
- AND  $\equiv a \wedge b$  (noté  $\&$ )
- XOR  $\equiv a \oplus b \equiv (a \vee b) \wedge \neg(a \wedge b)$  (noté  $\wedge$ )
- NAND  $\equiv \neg(a \wedge b)$
- NOR  $\equiv \neg(a \vee b)$

Une formule propositionnelle est dite **satisfiable** si il existe une valuation des variables qui rend la formule vraie.

On dit que  $f$  est une **conséquence logique** de  $e$  si pour toute valuation de  $e$ ,  $f$  est vraie et on note  $e \models f$

De même si  $f$  est une conséquence logique de  $e_1, \dots, e_n$ , on note  $e_1, \dots, e_n \models f$  qui est équivalent à  $(e_1 \wedge \dots \wedge e_n) \models f$

On parle de **système complet** si on peut exprimer toutes les fonctions logiques avec un nombre fini de connecteurs

Ainsi  $\{\wedge, \neg\}$ ,  $\{\vee, \neg\}$ ,  $\{\text{NAND}\}$  et  $\{\text{NOR}\}$  sont des systèmes complets

### VI.2 Table de vérité

Construire la table de vérité d'une formule propositionnelle est fastidieux car on a  $2^n$  lignes pour  $n$  variables

Ainsi on utilise l'algorithme de **Quine** pour simplifier les formules propositionnelles, pour chaque formule propositionnelle qui n'est pas  $\perp$  ou  $\top$  on choisit une variable qui apparaît dans la formule et on la simplifie, en créant deux sous enfants, un avec la variable à vrai et un avec la variable à faux.

On continue jusqu'à ce que tous les noeuds soient soit  $\perp$  soit  $\top$ , et si on a au moins un feuille  $\top$  alors la formule est satisfiable.

### VI.3 Formes normales et canoniques

On appelle **littéral** une variable ou sa négation

On appelle **conjonction** une suite de littéraux connectés par des  $\wedge$  (par exemple  $f_1 \wedge \dots \wedge f_n$ ) et on appelle **disjonction** une suite de littéraux connectés par des  $\vee$  (par exemple  $f_1 \vee \dots \vee f_n$ )

On parle de **forme conjonctive** si on a une conjonction de disjonctions et de **forme disjonctive** si on a une disjonction de conjonctions

Un **minterme** est une conjonction de littéraux où chaque variable apparaît une seule fois, et un **maxterme** est une disjonction de littéraux où chaque variable apparaît une seule fois

On parle de **forme normale conjonctive** de  $f$  si on a une formule  $f'$  équivalente à  $f$  telle que  $f'$  est sous forme conjonctive, et de **forme normale disjonctive** si on a une formule  $f'$  équivalente à  $f$  telle que  $f'$  est sous forme disjonctive





















### VI.4 Problème $k$ -SAT

Tout problème  $k$ -SAT peut être ramené à un problème 3-SAT















































































On va classer les problèmes en fonction de leur complexité :




- $P$  pour les problèmes qui peuvent être résolus en temps polynomial
- $NP$  pour les problèmes qui peuvent être vérifiés en temps polynomial

## Liste d'algorithmes

Liste chaînée en C 	21
Pile en C avec liste chaînée 	23
Pile en C avec tableau dynamique 	24
File en C 	26
Dictionnaire en C avec liste chaînée 	29
Parcours en profondeur (Arbres) 	32
Parcours en largeur (Arbres) 	32
Parcours en profondeur préfixe (Arbres binaires) 	34
Arbre binaire en C 	35
Recherche dans un arbre binaire de recherche 	36
Tri par sélection 	38
Tri bulle 	39
Tri par insertion 	40
Tri rapide 	41
Partition (Lomuto) 	41
Dichotomie (Récursive) 	41
Dichotomie (Impérative) 	42
Tri fusion 	43
Tri fusion 	44
Découpage en mots 	49

# Table des matières

	I Introduction au C	3		équilibrés	36
	I.1 Variables	3		II.9 Tas binaires	36
	I.2 Opérateurs	3		III Graphes	36
	I.3 Structures de contrôle	3		III.1 Définitions	36
	I.4 Fonctions	4		III.2 En OCaml	36
	I.5 Tableaux en C	5		III.3 En C	36
	I.6 Pointeurs	5		III.4 Étiquetage	36
	I.7 Types construits	7		III.5 Parcours de graphes	36
	II Introduction au OCaml	8		III.6 Cycles	36
	II.1 Expressions	8		III.7 Recherche de plus courts chemin	36
	II.2 Typage fort	9			
	II.3 Définitions	9		III.7.a Graphes avec poids négatifs	36
	II.4 Fonctions	10			
	II.5 Expressions plus complexes	12		I Bases	38
	II.6 Exceptions	13		I.1 Fonctions	38
	II.7 Listes	14		I.2 Complexité	38
	II.7.a Créer une liste	14		I.3 Algorithmes de tri	38
	II.7.b Opérations sur les listes	14		I.3.a Tri par sélection	38
	II.7.c Fonctions sur les listes	15		I.3.b Tri bulle	39
	II.8 Types construits	17		I.3.c Tri par insertion	40
	II.9 Programmation impérative	18		I.3.d Tri rapide	41
	II.9.a Blocs d'instructions	18		I.4 Algorithmes classiques	41
	II.9.b Références	19		I.4.a Dichotomie	41
	II.9.c Boucles	19		II Récursion	42
	II.10 Tableaux	20		II.1 Terminaison	42
	I Piles, files, dictionnaires	21		II.2 Récursion terminale	42
	I.1 Listes chaînées	21		II.3 Retour sur trace	43
	I.2 Piles	22		II.4 Programmation dynamique	43
	I.2.a En OCaml	22		III Stratégies algorithmiques	43
	I.2.b En C	23		III.1 Algorithmes gloutons	43
	I.3 Files	25		III.2 Diviser pour régner	43
	I.3.a En OCaml	25		IV SQL	44
	I.3.b En C	26		IV.1 Généralités	44
	I.4 Dictionnaires	28		IV.2 Requêtes	44
	I.4.a En OCaml	28		IV.3 Fonctions	45
	I.4.b En C	29		IV.4 Sous requêtes	46
	II Arbres	30		IV.5 Combiner les tables	46
	II.1 Définitions	30		IV.6 Créer une BDD	47
	II.2 Représentation en OCaml	31		IV.7 Type entités	48
	II.3 Arbres binaires stricts	32		V Algorithmes des textes	49
	II.4 Arbres binaires unaires	33		V.1 Bases	49
	II.5 Complexité	35		V.2 Algorithmes	49
	II.6 En C	35		V.3 Recherche de motifs	50
	II.7 Arbres binaires de recherche	35		VI Formules propositionnelles	51
	II.8 Arbres binaires de recherche			VI.1 Bases	51

 VI.2 Table de vérité	51
 VI.3 Formes normales et canoniques	
52	
 VI.4 Problème $k$ -SAT	52
Liste d'algorithmes	53
Table des matières	54