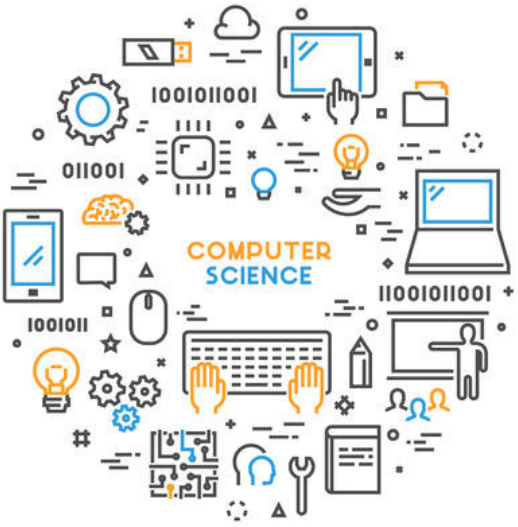
Essentiel d'informatique

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 :

Structures de données :

Informatique théorique :

Introduction

I Introduction au C

I.1 Variables

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

```c
int mango = 0;
```

Il est possible de définir plusieurs variables en même temps :

```c
int banana = apple = 12;
int pear, orange = 14; // pear est non initialisée et orange vaut 14
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 assignations

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 :

```c
if (x == 1) {
  // Do code
} else if (x > 12) {
  // Do code bis
} else {
  // Do code ter
}
```

> **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 :

```c
// De 0 à n - 1
for (int i = 0; i < n; i++) {

}

// De 0 à n - 1 tant que cond
for (int i = 0; i < n && cond; i++) {

}
```

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 :

```c
int my_func(int a, int b) {
  // Do code
  return 1;
}
```

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 appeller une fonction on fait :

```c
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 :

```c
int my_func(int);
```

I.5 Tableaux en C

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

Pour initialiser un tableau on a les manières suivantes :

```c
int[4] test = {0, 1, 2, 3}; // Initialise un tableau de taille 4 avec 0,1,2,3
int[] test = {0, 1, 2, 3}; // Initialise un tableau avec 0,1,2,3 (avec 4 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 :

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

Pour faire des tableaux de tableaux on fait :

```c
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
int[4][4] test ={ {0} }; // Initialise un tableau de taille 4x4 avec des 0
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

I.6 Pointeurs

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

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

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

```c
int a = 12;
int * b = &a;
// c est la valeur de a
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` :

```c
int * a = malloc(sizeof(int));

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

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 :

```c
int * a = malloc(sizeof(int));

// Do code

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 :

```c
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 `&` :

```c
int tab[4] = {0, 1, 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 :

```c
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 :

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

I.7 Types construits

Pour définir un alias on utilise `typedef` :

```c
typedef int my_type;

my_type a = 12;
```

Pour définir une structure on utilise `struct` :

```c
struct my_struct {
  int a;
  int b;
};

struct my_struct s;
s.a = 12;
s.b = 14;
```

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

```c
typedef struct my_struct {
  int a;
  int b;
} my_struct;

my_struct s;
s.a = 12;
s.b = 14;
```

On peut aussi initialiser une structure de la manière suivante :

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

On peut ainsi faire des structures récursives :

```c
typedef struct my_struct {
  int a;
  struct my_struct * next;
} my_struct;
```

Comme en OCaml on peut définir des énumérations :

```c
typedef enum my_enum {
  A,
  B,
  C
} 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 :

```c
typedef struct my_union {
  enum {
    A,
    B
  } type,
  union {
    int a;
    float b;
  } value;
} my_union;

my_union a = {.type = A, .value.a = 12};
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 (celà 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	`a + b`
Soustraction	`a - b`

Multiplication	`a * b`
Division	`a / b`
Modulo	`a mod 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`, `abs`, `acos`, `asin`, `atan`... avec `log` la fonction logarithme népérien

Comme en C on a les opérateurs binaires `&&` (et logique), `||` (ou logique) et `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 :

```
let a = 12;;

let a = a + 1;; (* a est maintenant 13 *)
```

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

```
int a = 12;
int b = a;
```

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

```
let a = 12;;
let b = a;;
```

Il est possible de définir plusieurs variables en même temps :

```
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` :

```
let a = 12 and b = 14 in
    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` :

```
let a = 12 in
    let b = 14 in
        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 :

```
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 :

```
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 :

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

Pour appeller une fonction on fait :

```
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 :

```
sum -12 12;; (* Erreur, on a -, 12 et 12 comme arguments *)
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 :

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

```
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 *)
(* 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` :

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

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

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

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

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

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

Ainsi le mot clé fonction 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` :

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

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

II.5 Expressions plus complexes

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

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

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

```
let my_func a =
  if a = 0 then
    begin
      let b = 1 in
      b * a (* Si a est 0 on renvoie a *)
    end
  else if a = 1 then
    (let b = 2 in b * a) (* Si a est 1 on renvoie 2a *)
  else
    3 * a;; (* Sinon on renvoie 3a *)
(* 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` :

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

```
let a = (12, 14);; (* a est un couple de 12 et 14 *)

print_int (fst a);; (* Affiche 12 *)
```

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

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

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

Il est possible de définir ses propres exceptions avec `exception` :

```
exception My_exception of string;; (* Définit une exception My_exception qui
prend un argument de type string *)

raise (My_exception "Erreur");; (* Lève l'exception My_exception avec le message
"Erreur" *)
```

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

```
try
  let a = 1 / 0 in
  a (* Renvoie a *)
with
  | 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 `[]` :

```
let lst = [];; (* Définit une liste vide *)

let lst = [1; 2; 3];; (* Définit une liste avec 1, 2 et 3 *)
(* int list *)

let lst = [[1; 2]; [3; 4]];; (* Définit une liste de listes *)
(* 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 `::` :

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

Il est possible de concaténer deux listes avec `@` :

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

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

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

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

II.7.c Fonctions sur les listes

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

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

```
let lst = [1; 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` :

```
let lst = [1; 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` :

```
let lst = [1; 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) :

```
let lst = [1; 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` ($\exists$) :

```
let lst = [1; 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) :

```
let lst = [1; 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` :

```
let lst = [1; 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`) :

```
let lst = [1; 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` :

```
let lst = [1; 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 :

```
let lst = [3; 2; 1];;

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` :

```
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 :

```
type basket = Fruit of int | Empty;; (* Définit un type fruit qui peut être Apple, Banana, Pear, Orange ou Fruit avec une quantité *)

let empty = Empty;; (* Définit un panier vide *)
let my_basket = Fruit 12;; (* Définit un panier avec 12 fruits *)
```

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

```
type point = { x: int; y: int };; (* Définit un type point qui a deux champs x et y *)

let origin = { x = 0; y = 0 };; (* Définit un point d'origine *)
print_int origin.x;; (* Affiche 0 *)
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` :

```
type point = { mutable x: int; mutable y: int };; (* Définit un type point qui a deux champs x et y mutables *)

let origin = { x = 0; y = 0 };; (* Définit un point d'origine *)
origin.x <- 12;; (* Modifie la valeur de x *)
```

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

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

```ocaml
type tree = Leaf | Node of tree * tree;; (* Définit un type arbre qui peut être une feuille ou un noeud avec deux sous arbres *)

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 `;` :

```ocaml
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 :

```ocaml
let my_func a =
  print_int a;
  print_string " ";
  print_int (a + 1);
  print_newline ();;
(* 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` :

```ocaml
let my_func a =
  if a = 0 then
    begin
      print_int 1;
      print_newline ()
    end
  else
    begin
      print_int a;
      print_newline ()
    end;;
(* int -> unit *)
```

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

```
let my_func a =
  match a with
  | 0 -> begin
    print_int 1;
    print_newline ()
  end
  | _ -> begin
    print_int a;
    print_newline ()
  end;;
(* 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` :

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

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

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

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

```
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` :

```
for i = 0 to 10 do
  print_int i;
  print_string " "
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` :

```
for i = 10 downto 0 do
  print_int i;
  print_string " "
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` :

```
let i = ref 0 in
while !i <= 10 do
  print_int !i;
  print_string " ";
  i := !i + 1
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 `[||]` :

```
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) :

```
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 :

```
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 $O(1)$) :

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

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

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

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

```
let tab = Array.make_matrix 3 3 0;; (* Définit un tableau de 3x3 initialisé à 0
*)

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 :

```c
typedef struct list {
  int value;
  struct list * next;
} list;
```

Pour ajouter un élément à une liste chaînée on fait :

```c
void add(list * lst, int value) {
  list * new = malloc(sizeof(*new));
  new->value = value;
  new->next = lst->next;
  return new;
}
```

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

```c
int get(list * lst) {
  if (lst == NULL) { // On empêche une segmentation fault
    return -1; // On renvoie une valeur par défaut
  }

  return lst->value;
}
```

Pour récupérer le reste de la liste chaînée on fait :

```c
list * next(list * lst) {
  if (lst == NULL) { // On empêche une segmentation fault
    return NULL;
  }

  return lst->next;
}
```

Pour parcourir une liste chaînée on fait :

```c
void print(list * lst) {
  while (lst != NULL) {
    printf("%d ", lst->value);
    lst = lst->next;
  }
}
```

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 :

```c
void free_list(list * lst) {
  while (lst != NULL) {
    list * tmp = lst;
    lst = lst->next;
    free(tmp);
  }
}
```

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` :

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

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

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

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

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

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

```ocaml
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` :

```
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 :

```c
typedef struct cell {
  int value;
  struct cell * next;
} cell;
typedef struct stack {
  cell * top;
} stack;
```

On définit alors les fonctions suivantes :

```c
stack * create_stack() {
  stack * s = malloc(sizeof(*s));
  s->top = NULL;
  return s;
}
```

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

```c
void push(stack * s, int value) {
  cell * new = malloc(sizeof(*new));
  new->value = value;
  new->next = s->top;
  s->top = new;
}
```

```c
int top(stack * s) {
  if (is_empty(s)) { // Très important pour éviter une segmentation fault
    return -1; // On renvoie une valeur par défaut
  }

  return s->top->value;
}
```

```c
int pop(stack * s) {
  if (is_empty(s)) { // Très important pour éviter une segmentation fault
    return -1; // On renvoie une valeur par défaut
  }

  int value = s->top->value;
  cell * tmp = s->top;
  s->top = s->top->next;
  free(tmp); // /!\ Il faut libérer la mémoire
  return value;
}
```

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.

```c
typedef struct stack {
  int * values;
  int size;
  int capacity;
} stack;
```

Ainsi les fonctions s'adaptent :

```c
stack * create_stack() {
  stack * s = malloc(sizeof(*s));
  s->values = malloc(4 * sizeof(int));
  s->size = 0;
  s->capacity = 4;
  return s;
}
```

```c
bool is_empty(stack * s) {
  return s->size == 0;
}
```

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

```c
void push(stack * s, int value) {
  if (s->size == s->capacity) {
    s->capacity *= 2;
    int * new_values = malloc(s->capacity * sizeof(int));

    for (int i = 0; i < s->size; i++) {
      new_values[i] = s->values[i];
    }

    free(s->values);
    s->values = new_values;
  }

  s->values[s->size] = value;
  s->size++;
}
```

```c
int top(stack * s) {
  if (is_empty(s)) { // Très important pour éviter une segmentation fault
    return -1; // On renvoie une valeur par défaut
  }

  return s->values[s->size - 1];
}
```

```c
int pop(stack * s) {
  if (is_empty(s)) { // Très important pour éviter une segmentation fault
    return -1; // On renvoie une valeur par défaut
  }

  int value = s->values[s->size - 1];
  s->size--;
  return value;
}
```

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` :

```ocaml
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 :

```c
void push(queue * q, int value) {
  cell * new = malloc(sizeof(*new));
  new->value = value;
  new->next = NULL;
  new->prev = q->back;

  if (q->back != NULL) { // Si la file n'est pas vide
    q->back->next = new;
  }

  q->back = new;

  if (q->front == NULL) { // Si la file est vide
    q->front = new;
  }
}
```

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

```c
int top(queue * q) {
  if (is_empty(q)) { // Très important pour éviter une segmentation fault
    return -1; // On renvoie une valeur par défaut
  }

  return q->front->value;
}
```

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

```c
int pop(queue * q) {
  if (is_empty(q)) { // Très important pour éviter une segmentation fault
    return -1; // On renvoie une valeur par défaut
  }

  int value = q->front->value;
  cell * tmp = q->front;
  q->front = q->front->next;

  if (q->front == NULL) { // Si la file est vide
    q->back = NULL;
  } else {
    q->front->prev = NULL;
  }

  free(tmp); // /!\ Il faut libérer la mémoire
  return value;
}
```

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

I.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 :

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

I.4.a En OCaml

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

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

```ocaml
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` :

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

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

```ocaml
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` :

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

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

```
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 considérér 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 $i \bmod n$.

```c
typedef struct cell {
  char * key;
  int value;
  struct cell * next;
} cell;
typedef struct dict {
  cell ** values;
  int size;
  int nb_keys;
} 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 :

```c
dict * create_dict(int size) {
  dict * d = malloc(sizeof(*d));
  d->values = malloc(size * sizeof(cell*));
  d->size = size;
  d->nb_keys = 0;

  for (int i = 0; i < size; i++) {
    d->values[i] = NULL;
  }

  return d;
}
```

```c
char * find(dict * d, char * key) {
  uint32_t h = hash(key) % d->size; // On calcule le hachage modulo la taille du tableau
  cell * c = d->values[h];

  while (c != NULL) { // On regarde toute la liste chaînée
    if (strcmp(c->key, key) == 0) {
      return c->value;
    }

    c = c->next;
  }

  return NULL;
}
```

```c
void add(dict * d, char * key, int value) {
  uint32_t h = hash(key) % d->size; // On calcule le hachage modulo la taille du tableau
  cell * c = d->values[h];

  while (c != NULL) { // On regarde toute la liste chaînée
    if (strcmp(c->key, key) == 0) {
      // On a trouvé la clé, on modifie la valeur
      c->value = value;
      return;
    }

    c = c->next;
  }

  // On ajoute un élément en tête de liste
  cell * new = malloc(sizeof(*new));
  new->key = key;
  new->value = value;
  new->next = d->values[h];
  d->values[h] = new;
  d->nb_keys++;
}
```

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 α vers un noeud β, on dit que α est le **parent** de β et que β est un **enfant** de α.

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 α pour tout noeud β tel qu'il existe un chemin de α à β.

On parle d'**antécédent** d'un noeud β pour tout noeud α tel qu'il existe un chemin de α à β.

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

On appelle **sous-arbre** d'un noeud α l'arbre formé par α 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 :

```
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 :

```
let rec dfs f = function
  | Node (a, children) -> print_string a; List.iter (dfs f) children
(* ('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 :

```
let bfs f tree =
  let queue = Queue.create () in
  Queue.push tree queue;
  while not (Queue.is_empty queue) do
    let Node (a, children) = Queue.pop queue in
      f a;
      List.iter (fun x -> Queue.push x queue) children
  done;;
(* ('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 :

```
type 'a binary_tree =
  | Leaf of 'a
  | 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 structurelle, 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 structurelle 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 `dfs f left;f a;dfs f right` ou `dfs f left;dfs f right;f a`.

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

Il peut être intéressant de numéroter 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 :

```
let rec to_list = function
  | Nil -> []
  | 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 $O(|A|)$:

```
let to_lst t =
  let rec aux lst = function
    | Nil -> lst
    | Node (x, lchild, rchild) -> aux (x::aux lst rchild) lchild
  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 :

```
typedef struct node {
  int value;
  struct node * left;
  struct node * right;
} 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 :

```
let rec mem y = function
  | Nil -> false
  | Node (x, left, right) when x = y -> true (* On a trouvé l'élément *)
  | Node (x, left, right) when x > y -> mem y left (* On va à gauche *)
  | Node (x, left, right) -> mem y right (* On va à droite *)
(* '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 $y = \inf_{\{z \in A \mid z < x\}} z$

On peut vouloir partitionner un arbre binaire de recherche par rapport à une valeur x :

```
let rec partition x = function
  | Nil -> Nil, Nil
  | Node (y, left, right) when y <= x ->
    let l, r = partition x left in Node (y, left, l), r
  | Node (y, left, right) ->
    let l, r = partition x right in l, Node (y, r, right)
(* 'a -> 'a tree -> 'a tree * 'a tree *)
```

Il est intéressant de noter que cet algorithme est en $O(h(A))$.

Si on veut vérifier qu'un arbre de binaire est un arbre de recherche, on doit vérifier :

- Que les sous-arbres gauches sont bien inférieurs à la racine
- Que les sous-arbres droits sont bien supérieurs à la racine

On a aussi des fortes contraintes sur les enfants de droite d'un enfant de gauche, les valeurs doivent être comprises entre la valeur de l'enfant de gauche et la valeur de son parent.

Il existe une deuxième manière de vérifier si un arbre binaire est un arbre de recherche, c'est de faire un parcours en profondeur infixe et de vérifier que les valeurs sont triées.

Il peut être intéressant de réaliser des opérations sur les arbres binaires de recherche, comme l'insertion d'un élément, on peut insérer soit au niveau des feuilles, soit au niveau de la racine.

```ocaml
let rec insert x = function
  | Nil -> Node (x, Nil, Nil)
  | Node (y, left, right) when x <= y -> Node (y, insert x left, right)
  | Node (y, left, right) -> Node (y, left, insert x right)
(* 'a -> 'a tree -> 'a tree *)
```

Pour insérer au niveau de la racine on peut utiliser la fonction de partition.

Pour la suppression d'un élément, on peut distinguer 3 cas :

- Si l'élément est une feuille, on le supprime
- Si l'élément a un seul enfant, on le remplace par son enfant
- Si l'élément a deux enfants, on le remplace par son successeur

Ainsi on a l'algorithme de suppression :

```ocaml
let rec pop_min = function
  | Nil -> raise Empty
  | Node (x, Nil, right) -> x, right
  | Node (x, left, right) -> let y, l = pop_min left in y, Node (x, l, right);;

let rec fusion t1 t2 = match t1, t2 with
  | Nil, t -> t
  | t, Nil -> t
  | _ -> let min, bst = pop_min t2 in Node (min, t1, bst);;

let rec remove x = function
  | Nil -> raise Not_found
  | Node (y, left, right) when x = y -> fusion left right
  | Node (y, left, right) when x < y -> Node (y, remove x left, right)
  | Node (y, left, right) -> Node (y, left, remove x right);;
(* 'a -> 'a tree -> 'a tree *)
```

On remarquera qu'il est possible d'utiliser des arbres binaires de recherche comme des dictionnaires.

II.8 Arbres binaires de recherche équilibrés

Les arbres binaires de recherche peuvent devenir très déséquilibrés, on peut alors avoir une complexité en $O(|A|)$ pour les opérations qui devraient être en $O(\log(|A|))$.

Arbre équilibré :
On dit que $\mathcal{E}$ est un ensemble d'arbre **équilibrés** si pour tout arbre $\mathcal{A}$, $|\mathcal{A}| = O(\log(|\mathcal{A}|))$

On va voir ici les **arbres rouges–noirs** qui sont des arbres binaires de recherche équilibrés.

Dans un arbre rouge-noirs on a les conditions suivantes :

- Chacun des noeuds rouges a un parent noir
- Toutes les branches ont le même nombre de noeuds noirs

Pour avoir un arbre binaire strict, on considère les `Nil` comme des noeuds noirs. On définit la **hauteure noire** d'un arbre rouge-noir $h_{n(A)}$ comme le nombre de noeuds noirs sur le chemin de la racine à une feuille.

Attention :
Les définitions varient beaucoup selon la source

Ensemble des arbres rouge–noirs :
L'ensemble des arbres rouge-noirs est un ensemble d'arbres équilibrés

> **Preuve :**
> Démontrable par récurrence sur la hauteur de l'arbre ou avec les arbres 2-3-4

Pour les ajouts et suppression, voir poly de Dewaele p217-219

II.9 Tas binaires

Un **tas binaire max** est un arbre binaire $\mathcal{A}$ tel que tout noeud excepté la racine a une valeur inférieure à celle de son parent.

Un **tas binaire min** est un arbre binaire $\mathcal{A}$ tel que tout noeud excepté la racine a une valeur supérieure à celle de son parent.

Dans un tas chacun des niveaux est rempli au maximum (excepté le dernier qui peut ne pas l'être) et les feuilles sont remplies de gauche à droite.

Ensemble des tas binaires :
L'ensemble des tas binaires est un ensemble d'arbres équilibrés

> **Preuve :**
> On a $2^{h(A)} \leq |A| < 2^{h(A)+1}$ d'où $h(A) \leq \log(|A|)$

Il est intéressant de représenter un tas binaire avec un tableau, en effet en utilisant la numérotation de Sosa-Stradonitz on peut représenter un tas binaire avec un tableau en mettant le noeud numéroté k en case $k-1$.

Ainsi la racine est en case 0, le fils gauche de la case k est en case $2k+1$ et le fils droit en case $2k+2$, et le parent de la case k est en case $\left\lfloor \frac{k-1}{2} \right\rfloor$.

Si un élément dépasse son père on peut le remonter en échangeant les deux éléments, et le faire récursivement.

Un élément peut aussi diminuer et devenir plus petit qu'un (ou ses deux) fils, on peut alors le descendre en échangeant avec le plus grand des deux fils, et le faire récursivement.

Ainsi on peut contruire un tas binaire en insérant les éléments et en les remontant, pour une complexité en $O(n\log(n))$, mais on peut aussi construire un tas binaire en $O(n)$ en insérant les éléments et en les descendant.

La bonne complexité d'un tas nous invite à faire un tris par tas : on prend une liste, et la racine est le plus grand élément, on l'échange avec le dernier élément et on refait un tas avec les éléments restants, puis on échange le premier élément avec l'avant-dernier, etc.

```ocaml
let heapsort t =
  let h = {data = t; n = Array.length t} in
  create_heap h;
  for i = h.n - 1 downto 1 do
    swap h 0 i;
    h.n <- h.n - 1;
    repair_down h 0;
  done;
```

Les tas permettent aussi de faire une file de priorité avec un tas, en comparant sur un tuple (p, x) avec p la priorité et x la valeur.

III Graphes

A faire À venir

III.1 Définitions

III.1.a Définitions générales

Un graphe non orienté est défini par un ensemble fini de sommets V et un ensemble fini d'arêtes E (une arête est une paire de sommets).

On dit qu'un graphe est **planaire** si il existe une représentation plane du graphe où les arêtes ne se croisent pas.

Pour une arrête $e_i = (v_j, v_k)$, les sommets v_j et v_k sont les **extrémités** de l'arête. On dit que e_i est **incidente** à v_j et v_k.

Les sommet v_j et v_k sont **adjacents**, et v_k est **voisin** de v_j.

Si $v_j = v_k$ on dit que l'arête est une **boucle**.

Le **degré** (ou la **valence**) d'un sommet est le nombre d'arêtes incidentes à ce sommet, on note $\deg(v)$ le degré de v.

L'**ordre** d'un graphe est le nombre de sommets, et le **degree** d'un graphe est le maximum des degrés des sommets.

Lemme des poignées de main :
Pour tout graphe non orienté on a :

$$\sum_{v \in V} \deg(v) = 2|E|$$

> **Preuve :**
> Chaque arête est incidente à deux sommets, donc chaque arête est comptée deux fois

III.1.b Graphes particuliers

Un graphe est dit **simple** si il n'a pas de boucles ni d'arêtes multiples, dans le cas contraire on parle de **multigraphe**.

Ici on ne s'intéressera qu'aux graphes simples.

Un graphe est **complet** si les sommets sont tous adjacents les uns aux autres.

Nombre d'arêtes dans un graphe complet :
Un graphe complet de n sommets a $\frac{n(n-1)}{2} = \binom{n}{k}$ arêtes.

Preuve :
Chaque sommet est adjacent à tous les autres sauf lui-même, donc il y a $n-1$ arêtes par sommet, d'où le résultat en utilisant le principe de double comptage

Un graphe $G' = (V', E')$ est un **sous-graphe** de $G = (V, E)$ si $V' \subset V$ et $E' \subset E$.

Le **sous graphe induit** par $V' \subset V$ est le sous-graphe de G avec les sommets de V' et les arêtes de E qui ont leurs deux extrémités dans V'.

Une **clique** est un sous-graphe complet.

III.1.c Cycles/Chemins

Un **chemin** de longueur n est une suite de $n+1$ sommets $\alpha = v_0, ..., v_n = \beta$ tel que v_i et $v_{\{i+1\}}$ soient adjacents. On appelle α et β les **extrémités** du chemin.

On dit qu'un chemin est **simple** si toutes les arêtes sont distinctes.

On dit qu'un chemin est élémentaire si tous les sommets sont distincts (à l'exception des extrémités).

On dit qu'un chemin est **ouvert** si les extrémités sont distinctes, et **fermé** si les extrémités sont identiques.

On notera $v_0 \triangleright ... \triangleright v_n$ un chemin de v_0 à v_n.

Chemin élémentaire :
De toute chemin on peut extraire un chemin élémentaire et un chemin simple

Preuve :
On suppose un chemin non élémentaire, alors il existe un sommet qui est visité deux fois, on peut alors faire disparaitre le cycle en supprimant les sommets entre les deux visites

En continuant on obtient un chemin élémentaire

On parle de **circuit** pour un chemin simple, fermé de longueur non nulle.

On parle de **cycle** pour un chemin simple et élémentaire fermé de longueur non nulle.

Un graphe est **acyclique** s'il ne contient pas de cycle.

Existence d'un cyle :
Si le degré de tous les sommets d'un graphe est supérieur ou égal à 2 alors le graphe contient un cycle

Preuve :
On construit une suite de sommet telle que $v_i \notin \{v_0, ..., v_{i-1}\}$ et v_i est voisin de v_{i-1}, cette construction se terminant par finitude des sommets du graphe

Ainsi on considère v_k le dernier sommet de la suite, ainsi il possède au moins deux voisins distincts dans la liste : v_{k-1} et $v_j \neq v_{k-1}$, d'où on peut considérer $v_k \triangleright v_j \triangleright v_{j+1} \triangleright ... \triangleright v_{k-1} \triangleright v_k$ un cycle car élémentaire, fermé par construction et de longueur au moins 3 (ça ne peut être un allez-retour)

Un chemin est dit **euclidien** si il s'agit d'un chemin simple passant par toutes les arêtes du graphe.

Un chemin est dit **hamiltonien** si il s'agit d'un chemin simple et élémentaire passant par tous les sommets du graphe.

Extraction de cycle :
De tout circuit dans un graphe on peut extraire un cycle

Preuve :
On note $v_0 \triangleright ... \triangleright v_n = v_0$ un tel circuit, ainsi on peut extraire un chemin simple et élémentaire C' de $v_1 \triangleright ... \triangleright v_n$, et en ajoutant devant ce chemin $v_0 \triangleright v_1$ on construit un chemin simple, élémentaire et fermé.

III.1.d Distance/Connexité

La **distance** entre deux sommets est la longueur du plus court chemin entre ces deux sommets. Un chemin de cette longueur est dit **géodésique**.

On considère que la distance est infinie si deux sommets ne sont pas accessibles.

Si il existe au moins un chemin de α à β alors β est **accessible** depuis α.

Un graphe est dit **connexe** si tout sommet est accessible depuis tout autre sommet.

Connexité :
Un graphe est connexe si et seulement si il existe un sommet qui soit à une distance finie de tous les autres sommets du graphe.

Preuve :
Le caractère nécessaire est évident d'après la définition.

Pour le caractère suffisant, on utilise l'inégalité triangulaire, si α et β sont à une distance finie de γ alors ils sont à une distance finie l'un de l'autre ($d(\alpha, \beta) \leq d(\alpha, \gamma) + d(\gamma, \beta)$)

On peut définir une relation $\mapsto$ sur l'ensemble V des sommets d'un graphe non-orienté G, par $\alpha \mapsto \beta$ si et seulement si α et β sont adjacents.

De manière générale, on peut définir une relation $\overset{*}{\mapsto}$ sur V par $\alpha \overset{*}{\mapsto} \beta$ si et seulement si il existe un chemin de α à β.

Cloture réflexive et transitive :
$\overset{*}{\mapsto}$ est la plus petite reltaion sur V contenant $\mapsto$ qui soit réflexive et transitive. On dit que $\overset{*}{\mapsto}$ est la **cloture réflexive et transitive** de $\mapsto$.

Graphe non orienté connexe :
Un graphe non-orienté connexe d'ordre n contient au moins $n-1$ arrêtes

Preuve :
On fait une récurrence sur le nombre de sommets d'un graphe connexe

C'est vrai pour $n=1$

Si $n>1$ supposons la propriété vraie pour un graphe d'ordre $n-1$, deux cas :
- Si il existe dans G un sommet v de degré 1, alors $G \setminus v$ est un graphe d'ordre $n-1$ connexe, d'où il contient au moins $n-2$ arêtes, et G contient donc au moins $n-1$ arêtes
- Sinon on a au moins la somme des degrés des sommets qui est supérieure à $2n$ d'où $|E| \geq n$ ce qui conclut

Graphe non orienté acyclique :
Un graphe non-orienté acyclique d'ordre n contient au plus $n-1$ arrêtes

Preuve :
Pour $n=1$ c'est vrai

Supposons la propriété acquise au rang $n-1$ et considérons un graphe olympique d'ordre n.

Il existe au moins un sommet de degré 0 ou 1 ainsi on le supprime ainsi que sa potentielle arête, le sous graphe obtenu, encore acyclique content au plus $n-2$ arêtes d'où par HR ça conclut

III.1.e Arbres

Un **arbre** est un graphe non-orienté connexe acyclique.

Caractérisation des arbres :
Les propriétés suivantes sont équivalentes :
- G est un arbre
- G est non-orienté connexe avec $n-1$ arêtes
- G est non-orienté acyclique avec $n-1$ arêtes

Un graphe non-orienté acyclique et non connexe est appelé **forêt**.

Si un tel graphe a p composantes connexes, alors on a $n-p$ arêtes.

III.1.f Graphes bipartis

Un graphe non orienté G est dit **biparti** si il existe une partition (V_1, V_2) de V tel que pour tout arête (v_i, v_j) de E, v_i et v_j ne soient pas dans le même ensemble V_1 ou V_2.

Longueur d'un chemin fermé :
Tout chemin fermé dans un graphe biparti a une longueur paire

Preuve :
On considère un chemin fermé $v_0 \triangleright ... \triangleright v_n = v_0$ dans un graphe biparti, on note V_1 et V_2 les deux ensembles de la partition.

Si $v_0 \in V_1$, alors les $V_{2i} \in V_1$ et les $V_{2i+1} \in V_2$, d'où $v_0 = v_n \in V_1$ d'où n pair

On parle de **graphe biparti complet** si chaque sommet de V_1 est adjacent à chaque sommet de V_2.

III.1.g Graphe orienté

Dans un graphe orienté, on distingue les arêtes menant d'un sommet α à un sommet β des arêtes menant de β à α. On parle d'**arc** pour une telle arête.

On parle de **degré entrant** d'un sommet pour le nombre d'arcs entrants, et de **degré sortant** pour le nombre d'arcs sortants.

On définit un **chemin** dans un graphe orienté de la même manière que pour un graphe non-orienté, mais en remplaçant les arêtes par des arcs.

On définit aussi la **distance** entre deux sommets de la même manière, mais elle n'est pas nécessairement symétrique.

Dans un graphe orienté on parle de **chaîne** entre deux sommets α et β une suite de $n+1$ somments tel que pour tout v_i et v_{i+1} il existe un arc de v_i à v_{i+1} ou de v_{i+1} à v_i.

Un graphe est dit **connexe** si il existe une *chaîne* entre tout couple de sommets.

Un graphe est dit **fortement connexe** si pour tout α, β il existe un chemin de α à β et de β à α.

III.2 En OCaml

On peut représenter de plusieurs manières des graphes en OCaml.

III.2.a Représentation mathématique

On peut représenter un graphe par un ensemble de sommets et un ensemble d'arêtes.

```
1 type 'a graph = {vertices : 'a list; edges : ('a * 'a) list};;
```

On pourra assez facilement implémenter des fonctions pour savoir si il existe une arête entre deux sommets, ou pour avoir la liste des voisins

III.2.b Liste d'adjacence

Pour accélérer la recherche on peut définir une liste d'adjaence, c'est à dire pour chaque sommet une liste de ses voisins.

```
1 type 'a vertex = { id: 'a; neighbors: 'a list };;
2 type 'a graph = 'a vertex list;;
```

III.2.c Tableau d'adjacence

Pour accélérer la recherche d'un sommet on peut utiliser un tableau d'adjacence, c'est à dire un tableau de listes de voisins.

```
1 type 'a graph = int list array;;
```

Il est aussi possible de stocker le graph dans une `Hashtbl`.

```
1 type 'a graph = ('a, 'a list) Hashtbl.t;;
```

III.2.d Matrice d'adjacence

Pour un graphe non orienté on peut utiliser une matrice d'adjacence, c'est à dire une matrice M telle que $M_{\{i,j\}} = 1$ si il existe une arête entre v_i et v_j.

```
1 type 'a graph = int array array;;
```

Cette représentation est très pratique mais prend beaucoup de place en mémoire.

Une propriété intéressante est que la puissance de la matrice d'adjacence donne le nombre de chemins de longueur n entre deux sommets.

III.3 En C

Comme d'habitude le C est cringe.

On utilise souvent une matrice d'adjacence représentée par un tableau de taille $n \times n$ et on accède à l'élément $M_{i,j}$ avec `M[i * n + j]`.

On peut aussi faire une liste d'adjacence (avec des listes chaînées) mais le programme demande de garder des tablaux pour une liste d'adjacence.

La première solution consiste à stocker le nombre d'éléments du tableau dans la première case, et ensuite les éléments.

La deuxième solution consiste à remplir le tableau et à garder une sentinelle à la fin (par exemple -1).

III.4 Étiquetage

Un étiquetage d'un graphe est la donnée d'une fonction $f : V \to E$ qui associe à chaque sommet un élément de E.

L'étiquetage a plusieurs usages : renseigner sur le nombre de ressources, sur les distances, sur les couleurs, etc.

Un dictionnaire est particulièrement adapté pour stocker un étiquetage.

Mais on peut aussi étiqueter les arêtes, notamment pour les pondérer. On parle dans ce cas de **graphe pondéré** et de **poids** d'une arête.

III.5 Parcours de graphes

La différence majeure avec un arbre est que l'on peut revenir sur un sommet déjà visité.

Ainsi il faut adapter les algorithmes de parcours pour ne pas tomber dans des boucles infinies en stockant les sommets déjà visités.

On peut donc faire un parcours en profondeur et en largeur par exemple.

On voit rapidement l'implémentation de l'un de ces parcours :

```ocaml
let dfs f graph start =
  let visited = Hashtbl.create 97 in
  let rec aux v =
    if not (Hashtbl.mem visited v) then begin
      f v;
      Hashtbl.add visited v ();
      List.iter aux (neighbors v)
    end
  in aux start;;
```

Il est bien sûr aussi possible de faire un parcours en largeur avec une file, et de faire un parcours en profondeur avec une pile.

Lien avec la forte connexité :
Lorsque qu'ils opèrent sur un graphe orienté fortement connexe, les parcours en profondeur et en largeur visitent tous les sommets du graphe quel que soit le sommet de départ

Inversement, si le parcours - quel que soit le sommet de départ - visite tous les sommets alors le graphe est fortement connexe.

Preuve :
Pour la première affirmation si on suppose V' l'ensemble des sommets non visités dans un parcours issu de v.

Soit $v' \in V'$, alors puisque le graphe est fortement connexe il existe un chemin de v à v', et donc v' est visité.

Lorsque l'on suit ce chemin, puisque $v \in V \setminus V'$ et $v' \in V'$ on visite deux sommets consécutifs tel que le premier est dans $V \setminus V'$ et le second dans V', ce qui est absurde.

La seconde affirmation est triviale

Si on considère un arbre (au sens de la théorie des graphes) et qu'on le parcourt en conservant les arcs dans l'ordre de parcours, on obtient un **arbre enraciné**

Arbre enraciné :
On considère $G = (V, E)$ et un sommet quelconque $v \in V$. Il existe une unique façon d'orienter les arêtes de E pour obtenir un arbre enraciné avec v comme racine.

Preuve :
On suppose qu'on peut construire deux arbres enracinés distincts avec v comme racine, on a ainsi une paire v_j et v_k tel que le sens de l'arc $v_j \triangleright v_k$ est différent dans les deux arbres.

Dans l'arbre où l'arc est orienté de v_j à v_k si on considère le chemin de v à v_j, alors on peut considérer le chemin de v à v_k en suivant l'arc $v_j \triangleright v_k$. Ainsi c'est l'unique chemin de v à v_k dans cet arbre.

Dans le second arbre enraciné il n'y a plus d'arc de v_j à v_k d'où il n'y a pas de chemin de v à v_k ce qui est absurde.

Pour trouver toutes les composantes connexes d'un graphe on peut utiliser un parcours en profondeur ou en largeur. On peut aussi utiliser une **recherche en profondeur d'abord** (DFS) qui consiste à parcourir le graphe en profondeur en partant d'un sommet, puis en continuant avec un autre sommet non visité. Et on peut construire des listes de sommets visités pour chaque composante connexe.

III.6 Cycles

III.6.a Cas des graphes non orientés

Il est aisé de savoir si un non-orienté graphe contient un cycle en regardant le nombre d'arêtes, en effet si il y a plus de $n-1$ arêtes alors il y a un cycle.

Mais identifier un cycle est plus compliqué, on peut utiliser un parcours en profondeur et regarder si on revient sur un sommet déjà visité.

On stocke le sommet depuis lequel on est parti pour chaque sommet, et si on revient sur un sommet déjà visité on peut remonter la chaîne pour trouver le cycle.

III.6.b Cas des graphes orientés

Trouver un cycle dans un graphe orienté est plus délicate, l'algorithme précédent fonctionne mais le résultat dépend du sommet de départ.

On va donc procéder à un **tri topologique** qui consiste à ordonner les sommets de sorte qui se un sommet v_i apparaît avant un sommet v_j alors il n'y a pas de chemin de v_j à v_i.

Degré sortant nul :
Dans un graphe orienté acyclique il existe au moins un sommet de degré sortant nul

Preuve :
Si il n'existait pas de sommet de degré sortant nul, alors on peut construire un chemin infini en suivant les arcs sortants d'un sommet à l'autre car le graphe est acyclique ce qui est absurde dans un graphe de taille finie

L'algorithme de tri topologique est le suivant :

- On trouve un sommet de degré sortant nul et on le supprime du graphe en le mettant à la fin de la liste
- On continue en prenant les sommets de degré sortant nul en les ajoutant petit à petit à la fin de la liste (en gardant le premier sommet de degré sortant nul tout au bout de la liste)

Mais il suffit en fait de remarquer que en faisant un parcours en profondeur on peut obtenir un tri topologique : on arrivera toujours à un sommet de degré sortant nul dans un graphe acyclique.

Ainsi il suffit de faire un parcours en profondeur et d'ajouter les sommets à la fin de la liste.

Une implémentation en OCaml est la suivante :

```ocaml
let topological_sort gr vertices =
  let visited = Hashtbl.create 97 and let result = ref [] in
    let rec explore v =
      Hashbl.add visited v true;
      (* On fait notre dfs *)
      List.iter
        (fun w -> if not (Hashtbl.mem visited w) then explore w)
        (neighbors gr v);
        sorted := v :: !sorted
        (* Appel de la fonction explore pour chaque sommet *)
    in List.iter
      (fun v -> if not (Hashtbl.mem visited v) then explore v)
      vertices;
  !sorted;;
```

Cet algorithme sert de base à beaucoup d'autres, on considère qu'on colore tous les sommets en blanc, et que quand on traite un sommet on le met en gris, et quand on a fini on le met en noir.

Ainsi le graphe contient un cycle si et seulement si on retombe sur un sommet gris dans le graphe, les sommets gris formant un chemin dans ce dernier.

III.7 Recherche de plus cours chemin

On a déjà parlé de **pondération** d'un graphe. On pose $w : E \to R$ une fonction qui associe à chaque arête un poids, et on l'étend a $V \times V$ en posant $w(v_i, v_i) = 0$ et si il n'y a pas d'arête entre v_i et v_j on pose $w(v_i, v_j) = +\infty$.

On peut donc voir un graphe pondéré comme une matrice W telle que $W_{i,j} = w(v_i, v_j)$.

La **longueur** d'un chemin est la somme des poids des arêtes du chemin.

III.7.a Algorithme de Floyd–Warshall

L'algorithme de Floyd-Warshall permet de trouver le plus court chemin entre tous les sommets d'un graphe pondéré.

Ainsi on travaille sur une matrice D telle que $D_{i,j} = w(v_i, v_j)$, et on va calculer les exponentiations successives de cette matrice.

Principe de sous-optimalité :
Si il existe un chemin de v_i à v_j passant par v_k alors le plus court chemin de v_i à v_j est le plus court chemin de v_i à v_k suivi du plus court chemin de v_k à v_j

Preuve :
Si ce n'est pas le cas, on peut alors construire un plus court chemin ce qui est contradictoire

On remarque aussi que le terme $D_{i,j}^{(k+1)}$ peut se calculer à partir des coefficients de D^k : $D_{i,j}^{(k+1)} = \min\left(D_{i,j}^k, D_{i,k+1}^k + D_{k+1,j}^k\right)$

En effet un plus court chemin ne passe soit pas par v_k et est donc de longueur $D_{i,j}^k$, soit il passe par v_k et est donc de longueur $D_{i,k+1}^k + D_{k+1,j}^k$.

Enfin on remarquera qu'on peut bien évidemment calculer ces termes avec $D^{(k+1)}$

On peut donc implémenter l'algorithme de Floyd-Warshall en OCaml :

```
let floyd_warshall w =
  let n = Array.length w in
  let m = Array.make_matrix n n infinity in
    for i = 0 to n - 1 do
      for j = 0 to n - 1 do
        m.(i).(j) <- w.(i).(j)
      done
    done;
    for k = 0 to n - 1 do
      for i = 0 to n - 1 do
        for j = 0 to n - 1 do
          m.(i).(j) <- min m.(i).(j) (m.(i).(k) + m.(k).(j))
        done
      done
    done;
    m;;
```

Il est bien évidemment ensuite possible d'adapter Floyd-Warshall pour trouver les chemins et non seulement les longueurs.

III.7.b Algorithme de Dijkstra

L'algorithme de Dijkstra permet de trouver le plus court chemin entre un sommet de départ et tous les autres sommets d'un graphe pondéré.

On pose S l'ensemble des sommets pour lesquels on connait sa plus courte distance au départ et $\overline{S}$ son complémentaire.

On pose δ de la manière suivante :

- Pour les sommets $v \in S, \delta(v) = d(v_i, v)$
- Pour les sommets $v \in \overline{S}, \delta(v)$ correspond à la longueur du plus court chemin dans le graphe menant de v_i à v et passant par les sommets de S

Initialement on pose $S = \{v_i\}$ et $\overline{S} = V \setminus \{v_i\}$, et on pose $\delta(v_i) = 0$ et $\delta(v) = +\infty$ pour tout $v \in \overline{S}$ (on peut aussi poser $\delta(v) = w(v_i, v)$ pour tout voisin de v_i).

Invariants de Dijkstra :
Après chaque étape de l'algorithme de Dijkstra, on a $\delta(v) = d(v_i, v)$ pour tout $v \in S$ et $\delta(v)$ la longueur du plus court chemin menant de v_i à v et passant par les sommets de S pour tout $v \in \overline{S}$

Preuve :
Par récurrence :

- Initialement seul le sommet v_i est dans S et on a bien $\delta(v_i) = 0$ et $\delta(v) = +\infty$ pour tout $v \in \overline{S}$ d'où les deux invariants sont vérifiés.
- On suppose les deux invariants vérifiés à l'étape k et on considère l'étape $k+1$. On a 3 cas pour étudier un sommet v :
  - Soit le sommet v est déjà dans S, alors on a $\delta(v) = d(v_i, v)$ et on ne fait rien
  - Soit le sommet est celui qui entre dans S, alors on a $\delta \geq d(v_i, v)$ il suffit de montrer que $\delta = d(v_i, v)$. Pour faire on considère un chemin de longueur minimale dans la totalité du graphe menant de v_i à v et on note u le premier sommet de ce chemin

    Alors on a $d(v_i, v) = d(v_i, u) + d(u, v) \geq d(v_i, u) \geq \delta(u) \geq \delta(v)$, et on a donc $\delta(v) = d(v_i, v)$
  - Soit le sommet reste dans $\overline{S}$, notons u le sommet rentrant à cette étape et considérons un chemin de longueur minimale dans la totalité du graphe menant de v_i à v et passant par les sommets de S. On a encore 3 cas :
    - Soit ce chemin ne passe pas par u, la valeur de $\delta(v)$ est donc inchangée
    - Soit ce chemin passe par u en dernier, ainsi on a $d(v_i, u) + w(u, v) = \delta(u) + w(u, v)$, et ainsi on a cette valeur qui est nécessairement inférieure ou égale à $\delta(v)$
    - Soit ce chemin passe par u mais il existe un point u' de S suivant u dans le chemin de v_i à v que l'on avait et $\delta(u') \leq \delta(u)$ puisque u' était présent avant, donc on peut trouver un chemin de longueur égale ou inférieure menant de v_i à u' ne passant pas par u et donc on garde la valeur de $\delta(v)$

Ainsi ça conclut sur la correction de l'algorithme

Correction de Dijkstra :
Quand Dijkstra termine, on a $\delta(v) = d(v_i, v)$ pour tout $v \in V$

Preuve :
D'après les invariants c'est vrai pour tout sommet dans S

Et si il existe des sommets dans $\overline{S}$ alors on n'a pu trouver d'arc menant de v_i à v d'où $\delta(v) = +\infty$

Pour implémenter Dijkstra on va faire un parcours en profondeur de l'arbre et pour chaque sommet v, si $v \notin S$ (en arrivant de u), alors $\delta(v) = \delta(u) + w(u, v)$. Si $v \in S$, alors on pose $\delta(v) = \min(\delta(v), \delta(u) + w(u, v))$.

En OCaml on peut utiliser une `Heapq` pour Djikstra avec `Heapq.create` pour créer une file de priorité, `Heapq.push` pour ajouter un élément, `Heapq.pop` pour retirer l'élément de priorité minimale et `Heapq.is_empty` pour savoir si la file est vide.

Ainsi on peut implémenter Dijkstra en OCaml :

```ocaml
let dijkstra gr s =
  let distances = Hashtbl.create 97 and openv = Heapq.create () in Heapq.push openv ((s, []), 0.0);

  while not (Heapq.is_empty openv) do
    let ((u, ch), du) = Heapq.pop openv in
      if not (Hashtbl.mem distances u) then begin
        Hashtbl.add distances u (du, List.rev (u::ch));
        List.iter
          (fun (v, w) -> Heapq.push openv ((v, u::ch), du +. w))
          (neighbors gr u)
      end
  done;
  distances;;
```

On remarque ici que la `Heapq` nous garantit que quand on retire un élément de la file, c'est bien le plus petit chemin.

III.7.c Algorithme A*

L'algorithme de Dijkstra permet d'obtenir avec certitude le plus court chemin, mais il peut être très coûteux en temps de calcul, il n'est pas très efficace pour des graphes de grande taille.

L'algorithme A* est une amélioration de Dijkstra qui permet de trouver le plus court chemin en utilisant une heuristique pour guider la recherche : une **heuristique** est une fonction qui estime la distance entre un sommet et le sommet d'arrivée, qu'on notera $h(v)$.

Ainsi on choisit le sommet suivant avec $\delta(v) + h(v)$ et non plus $\delta(v)$.

Heuristique admissible :
Une heuristique est dite **admissible** si :
- $\forall v \in V, h(v) \leq d(v, v_j)$
- $h(v_j) = 0$

Correction de A* :
Si l'heuristique est admissible, alors A* trouve le plus court chemin

Preuve :
On va prouver par contraposée, supposons un chemin de v_i à v_j qui ne soit pas le plus court et que $h(v_j) = 0$

On considère un chemin plus court, ainsi il y a au moins un arc $v \triangleright v'$ qui n'a pas été parcouru et on choisit cet arc.

Si l'arc menant à v_j a été sélectrionné avant c'est nécessairement que $\delta(v_j) + h(v_j) \leq \delta(v') + h(v')$ or $h(v_j) = 0$ et $\delta(v_j) \geq d(v_i, v_j)$ d'où $\delta(v') + h(v') > d(v_i, v_j)$.

D'où par principe de sous optimalité et puisque l'on est sur le plus court chemin, $\delta(v') = d(v', v_1)$ et on a donc $d(v_i, v') + h(v') > d(v_i, v_j)$ d'où $h(v') > d(v', v_j)$ ainsi l'heuristique n'est pas admissible

III.7.d 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 $O(n \times p)$

Voir dernière page du cours pour les infos

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 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 :

```c
void selection_sort(int arr[], int n) {
  for (int i = 0; i < n; i++) {
    // Les i premiers éléments sont bien triés
    int min_i = i;

    for (int j = i+1; j<n; j++) {
      if (arr[j] < arr(min_i)) {
        min_i = j;
      }
    }

    // On échange les éléments en i et min_i
    int tmp = arr[i];
    arr[i] = arr[min_i];
    arr[min_i] = tmp;
  }
}
```

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 inconvéniant d'avoir une complexité quadratique et de ne pas être stable

I.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 :

```c
let bubble_sort(int arr[], int n) {
  for (int i = 0; i < n; i++) {
    // Les i premiers éléments sont bien placés
    int k_last_perm = n-1;
    int smallest = arr[n-1];

    for (int j = n-1; j > i; j--) {
      if (arr[j-1] <= smallest) {
        // On change de bulle
        arr[j] = smallest;
        smallest = arr[j-1];
      } else {
        // On fait descendre la bulle
        arr[j] = arr[j-1];
        k_last_perm = j - 1;
      }
    }

    arr[i] = smallest;
    // On n'a pas besoin de regarder les éléments entre i+1 et k_last_perm car on n'a fait aucune modification
    i = k_last_perm + 1;
  }
}
```

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 :

```c
int insertion_sort(int arr[], int n) {
  for (int i = 0; i < n; i++) {
    // Les i premiers éléments sont bien triés
    int j = i;
    int elem = arr[i];

    for (; j>0 && elem < arr[j-1]; j--) {
      arr[j] = arr[j-1];
    }

    arr[j] = elem;
  }
}
```

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 :

```c
void quick_sort(int * arr, int n) {
  if (n <= 1) { // Déjà trié
    return;
  }

  int pivot = partition(arr, n);
  quick_sort(arr, pivot);
  quick_sort(&arr[pivot+1], n-pivot-1);
}
```

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

```c
int partition(int arr[], int n) {
  int pivot = arr[0];
  int p = 1;

  for (int i = 1; i<n; i++) {
    if (arr[i] < pivot) {
      arr_swap(arr, i, p); // On échange les éléments i et p
      p++;
    }
  }

  arr_swap(arr, 0, p-1); // On échange le pivot et le dernier élément plus petit que le pivot
  return p-1;
}
```

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 :

```c
let index(int * arr, int n, int elem) {
  if (n == 0) {
    return -1; // On ne peut pas trouver
  }

  int m = n/2;

  if (arr[m] == elem) { // On a trouvé!
    return m;
  } else if (arr[m] > m) { // L'élément se situe peut être dans la partie gauche
    return index(arr, m, elem);
  } else { // L'élément se situe peut être dans la partie droite
    int idx = index(&arr[m+1], n-m-1, elem);

    if (idx != -1) {
      idx += m+1;
    }

    return idx;
  }
}
```

On peut aussi faire de manière itérative :

```c
let index(int * arr, int n, int elem) {
  int l = 0, r = n;

  while (l < r) { // On recherche dans le tableau avec deux compteurs
    int m = (l+r)/2;

    if (arr[m] == val) { // On a trouvé!
      return m;
    } else if (arr[m] > m) { // L'élément se situe peut être dans la partie
gauche
      r = m;
    } else { // L'élément se situe peut être dans la partie droite
      l = m + 1;
    }
  }

  return -1; // Pas trouvé!
}
```

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

On dit que $(\mathcal{E}, \leq)$ est un **ensemble bien fondé** si toute partie non vide de $\mathcal{E}$ admet un élément minimal ($\forall x \in A, a \leq x \implies x = a$)

On dit que $(\mathcal{E}, \leq)$ est un **ensemble bien ordonné** si toute partie non vide de $\mathcal{E}$ admet un plus petit élément ($\forall x \in A, a \leq x$)

Suite infinie :
Il n'existe pas de suite infinie strictement décroissante dans un ensemble bien fondé

Preuve :
On suppose qu'il existe une suite infinie strictement décroissante $a_0 > a_1 > a_2 > ...$

On considère l'ensemble $A = \{a_i \mid i \in \mathbb{N}\}$, cet ensemble est non vide et admet un élément minimal a_n

Ainsi $\forall k \geq n$, on a $a_k \geq a_n$ d'où $a_k = a_n$ d'où la suite est stationnaire ce qui est absurde

Principe d'induction (prédicat) :
Soit $(\mathcal{E}, \leq)$ bien fondé, et M l'ensemble des éléments minimaux. Alors si un prédicat P est vrai pour tout $x \in M$ et si $\forall x \in \mathcal{E}, (\forall y < x, P(y)) \Rightarrow P(x)$, alors $\forall x \in \mathcal{E}, P(x)$

Preuve :
Par l'absure en supposant qu'il existe X tel que $\forall x \in X, \neg P(x)$ cet ensemble admet un élément minimal x_0 nécessairement dans $\mathcal{E} \setminus M$

Or on a $\forall y < x_0, P(y)$ d'où $P(x_0)$ ce qui est absurde

Variant de boucle :
Si dans une boucle on peut exhiber un variant choisi dans un ensemble bien fondé qui décroit strictement à chaque itération, alors la boucle se termine

Preuve :
Par principe d'induction :
- Si $x \in M$ alors la boucle se termine car il ne peut y avoir d'itération supplémentaire
- Si pour $x \in \mathcal{E}$ donné la boucle se termine $\forall y < x$, alors l'itération suivante sera un de ces y d'où la boucle se termine

L'ensemble des flottants positifs et bien fondé et bien ordonné mais les erreurs d'arrondis peuvent poser problème

Principe d'induction (récursif) :
Soit f récursive de $\mathcal{E}$ vers $\mathcal{F}$ et φ de $\mathcal{E}$ vers un ensemble bien fondé telle que pour tout $x \in \mathcal{E}$:

- Soit f renvoie un résultat
- Soit la fonction calcule le résultat par un nombre fini d'appels récursifs avec $\varphi(y_i) < \varphi(x)$

Alors la fonction termine pour tout $x \in \mathcal{E}$

Ordres utiles :

- L'ordre des entiers $(\mathbb{N}, \leq)$
- L'ordre lexicographique $(\mathbb{N}^2, \underset{l}{\leq})$: $(a, b) \underset{l}{\leq} (c, d) \Leftrightarrow a < c \vee (a = c \wedge b \leq d)$ est bien ordonné (et donc bien fondé)
- L'ordre produit $(\mathbb{N}^2, \underset{\times}{\leq})$: $(a, b) \underset{\times}{\leq} (c, d) \Leftrightarrow a \leq c \wedge b \leq d$ n'est pas ordonné mais est bien fondé

II.2 Récursion terminale

La récursion terminale est une récursion où le résultat de l'appel récursif est directement le résultat de la fonction, ie l'appel récursif est la dernière opération de la fonction

II.3 Retour sur trace

Le retour sur trace est une technique de programmation qui consiste à sauvegarder l'état de la fonction à chaque appel récursif pour pouvoir revenir en arrière si besoin : on peut ainsi faire des essais et revenir en arrière (par exemple faire une hypothèse et revenir en arrière si elle est fausse et essayer une autre hypothèse)

On pourra utiliser le type `Option` du OCaml pour gérer les retours sur trace :

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

II.4 Programmation dynamique

Les approches récursives sont souvent inefficaces car elles recalculent plusieurs fois les mêmes valeurs, la programmation dynamique consiste à stocker les valeurs déjà calculées pour ne pas les recalculer.

Par exemple pour trouver la distance d'édition entre deux chaînes de caractères on peut utiliser la programmation dynamique. Pour déterminer cette distance on peut se baser sur les propriétés suivantes :

A faire (DISTANCE DE TEXTE)

On peut **mémoïser** les résultats pour ne pas les recalculer, notamment avec une `Hashtbl` ou un tableau

On peut aussi au lieu de mémoïser faire une approche de type **bottom-up** où on part des valeurs les plus petites pour arriver aux valeurs les plus grandes : pour Fibonnaci on n'est pas obligés de stocker toutes les valeurs mais juste les deux dernières.

III Stratégies algorithmiques

III.1 Algorithmes gloutons

Il existe des problèmes pour lesquels on peut utiliser des algorithmes gloutons, qui consistent à prendre la meilleure solution locale à chaque étape.

Par exemple pour le problème du rendu de monnaie on peut prendre la plus grosse pièce à chaque fois.

Pour montrer qu'un algorithme glouton est optimal on suppose qu'il existe une solution optimale qui n'utilise pas la solution gloutonne, on obtient alors une contradiction.

L'approche gloutonne est donc une bonne solution, mais il faut faire attention à la correction de l'algorithme.

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 :

```
let rec partition = function
  | h1::h2::t -> let l,r = partition t in h1::l, h2::r
  | lst -> lst, [];;

let rec merge l1 l2 = match l1,l2 with
  | (h1::t1), (h2::t2) when h1 <= h2 -> h1::(merge t1 l2)
  | (h1::t1), (h2::t2) -> h2::(merge l1 t2)
  | l1, [] -> l1;;

let rec fusion_sort lst = match split lst with
  | lst, [] -> lst
  | 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 (A faire ça)

Pour rechercher un pic dans un tableau par exemple, on peut utiliser une approche en diviser pour régner on considère m et $m-1$ au milieu du tableau, si `t[m-1] >= t[m]` alors on a un pic à gauche et sinon on a un pic à droite

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 :

```sql
# Seulement les colonnes spécifiées
SELECT col1, ..., coln FROM table

# Toutes les colonnes
SELECT * FROM table

# Toutes les colonnes mais sans doublon
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 :

```sql
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 chaine 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

```sql
# Triés par valeur croissante
SELECT * FROM table ORDER BY col

# Triés par valeur décroissante
SELECT * FROM table ORDER BY col DESC
```

Pour limiter le nombre de valeurs on utilise

```sql
# On prend au maximum 3 éléments
SELECT * FROM table LIMIT 3

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

IV.3 Fonctions

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

```sql
# Nombre d'éléments dans la table
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 :

```sql
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

```sql
# Renvoie des groupes des valeurs de col
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

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

Pour sélectionner des groupes on peut utiliser :

```sql
# 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
SELECT col1, MIN(col2) FROM table GROUP BY col1 HAVING MIN(col2) > x
```

Les opérations sont executées 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

```sql
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 :

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

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

```sql
# Ici on va sélectionner seulement les lignes dont la valeur de col correspond à la condition cond
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

```sql
# Ainsi on renvoie la moyenne d'une colonne col2 telle que ses éléments vérifient la condition
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

```sql
# Sélectionne dans le produit cartésien des deux tables
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

```sql
# Sélectionne dans le produit cartésien des deux tables seulement les éléments 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 :

```sql
# On sélectionne les éléments de la table1 en ajoutant la table2 si la condition 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

```sql
# On sélectionne les éléments de la table1 en concaténant les éléments dont la 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

```sql
# On a les éléments qui vérifient la cond1 ou cond2
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

```sql
# On a les éléments qui vérifient la cond1 et cond2
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

```sql
CREATE TABLE IF NOT EXISTS table (
  col1 TYPE1,
  col2 TYPE2,
  col3 TYPE3
)
```

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 :

```sql
CREATE TABLE IF NOT EXISTS table (
  col1 TYPE1 PRIMARY KEY,
  ...
)
```

Les autres attibuts 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 autorise 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 un clé étrangère qui vont être des liens entre les différentes tables

```sql
CREATE TABLE IF NOT EXISTS table (
  ...,
  FOREIGN KEY (col) REFERENCES table(col)
)
```

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

Pour insérer dans une table on utilise :

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

On peut modifier un élement :

```sql
UPDATE table SET col1 = value WHERE cond
```

On peut aussi supprimer un élement :

```sql
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 chaine en C on fera :

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

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) :

```c
void is_sentence(char * s) {
  if (s[0] == '\0') {
    return;
  }

  int n = strlen(s);
  int * arr = malloc((n+1) * sizeof(*arr));
  arr[0] = 0;

  for (int i = 1; i <= n; i++) {
    arr[i] = -1; // On initialise à false car le malloc ne le fait pas
    char tmp = s[i];
    s[i] = '\0';
    for (int j = i-1; arr[i] != -1 && j >= 0; --j) {
      if (arr[j] != -1 && is_word(&s[j])) {
        arr[i] = j;
      }
    }
    s[i] = tmp;
  }
  // Le tableau arr contient l'indice du début du mot précédent (ou -1 si il n'y en a pas)
  free(arr);
}
```

Il est intéressant de mémoïser 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 dicothomique : 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 dictonnaire 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 $\bot$ 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 $(\mathcal{A} \wedge \top) \vee ((\mathcal{B} \wedge \bot) \wedge \mathcal{C})$ 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 `^`)
- `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 $e \vDash f$

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

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

Ainsi $\{\wedge, \neg\}$, $\{\vee, \neg\}$, $\{$`NAND`$\}$ et $\{$`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 $\bot$ 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 $\bot$ 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 $f_1 \wedge ... \wedge f_n$) et on appelle **disjonction** une suite de littéraux connectés par des $\vee$ (par exemple $f_1 \vee ... \vee f_n$)

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 conjonctions 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

Table des matières