Plan du cours

1. Introduction au langage Caml ;
2. Programmation impérative ;
3. Preuve de programmes : invariants de boucles ;
4. Programmation fonctionnelle ;
5. Récursivité ;
6. Induction structurelle : listes ;
7. Complexité des algorithmes : tris élémentaires ;
8. Algorithmes "diviser pour régner" ;
9. Piles, expressions algébriques ;
10. Logique, circuits combinatoires.

Page web du cours

http://asoyeur.free.fr/mp2/option_info/index.html

Sites caml à l'inria

http://pauillac.inria.fr/caml/index-fra.html

http://pauillac.inria.fr/caml/distrib-caml-light-fra.html

Introduction à Caml

Evaluation en Caml

```
#2 + 2;;

- : int = 4

  #2. +. 3. ;;

- : float = 5.13
```

Liaisons

```
#let a = 3;;

a : int = 3

#a + 1;;

- : int = 4
```

Une fois évaluée, une expression ne change plus

```
#let a = 1;;
a : int = 1
#let b = a + 1;;
b : int = 2
#let a = 2;;
a : int = 2
#b;;
- : int = 2
```

Liaisons locales

```
#let a = 3 in a + 4;;

- : int = 7

#a;;

Entrée interactive:
>a;;<EOF>
>^
L'identificateur a n'est pas défini.
```

Liaisons multiples :

```
#let a = 2 and b = 3 in
  (a + b) * 2;;

- : int = 10
```

Dépendance de liaisons

```
#let a = 2 and b = 2 * a in
  a + b;;
```

```
Entrée interactive:
>let a = 2 and b = 2 * a in
>                       ^
L'identificateur a n'est pas défini.
```

```
#let a = 2 in
let b = 2 * a in
  a + b;;
```

```
- : int = 6
```

Les parenthèses servent à forcer l'ordre d'évaluation

```
#(2 + 3) * 5;;
- : int = 25
```

Il en est de même pour toutes les constructions en Caml .

Commentaires

```
(* Un commentaire *)
2 + 2;;
```

Entiers codés sur 31 bits : $-2^{30} \leq n \leq 2^{30} - 1$

Quotient et reste d'une division euclidienne : $22 = 7 \times 3 + 1$

```
#22 / 3;;
- : int = 7
#22 mod 3;;
- : int = 1
```

`print_int` affiche un entier.

Réels

```
#4. *. 5.;;
- : float = 20.0
#2.E4 /. 2.4;;
- : float = 8333.33333333
```

`print_float` affiche un flottant.

Conversion caractère - code ASCII

```
#char_of_int 65;;

- : char = ‘A‘

#int_of_char ‘T‘;;

- : int = 84
```

`print_char` affiche un caractère.

Chaîne de caractères

```
#let chaine = "bonjour!";;
```

```
chaine : string = "bonjour!"
```

Concaténation de deux chaînes

```
#let chaine1 = "Bonjour" and chaine2 = "le monde" in
print_string  chaine1 ^ chaine2;;
```

```
Bonjourle monde- : unit = ()
```

Longueur d'une chaîne et accès au ième caractère ($0 \leq i \leq n-1$).

```
#let chaine = "Bonjour";;
chaine : string = "Bonjour"
#string_length chaine;;
- : int = 7
#chaine.[0];;
- : char = ‘B‘
#chaine.[6];;
- : char = ‘r‘
#chaine.[7];;
Exception non rattrapée: Invalid_argument "nth_char"
```

Les chaînes de caractères sont *mutables*

```
#let ch = "abc";;
ch : string = "abc"
#ch.[1] <- `d`;;
- : unit = ()
#ch;;
- : string = "adc"
```

Modification de chaînes

```
#let c1 = "abcd";;

c1 : string = "abcd"

#let c2 = c1;;

c2 : string = "abcd"

#c2.[0] <- `A`;;

- : unit = ()

#c1;;

- : string = "Abcd"
```

Voir les vecteurs.

Couples et n-uplets

```
#(2, 3);;

- : int * int = 2, 3

#5, true, 4.;;

- : int * bool * float = 5, true, 4.0
```

Remarquez le symbole * pour désigner des produits cartésiens.

Les parenthèses sont facultatives pour définir des n-uplets.

Type `unit` à une seule valeur `()` : effet de bord

```
#print_string "Bonjour";;

Bonjour- : unit = ()
```

Fonction à un argument :

```
let f x =
x + 2;;
```

```
f : int -> int = <fun>
```

```
#f 3;;
```

```
- : int = 5
```

```
#let f x =
  x +. 2. ;;

f : float -> float = <fun>

#f 3. *. 2. ;;

- : float = 10.0

#f (3. *. 2.) ;;

- : float = 8.0
```

Fonction à plusieurs arguments

```
#let f x y =
  x * y + 2;;
```

```
f : int -> int -> int = <fun>
```

```
#f 3 5;;
```

```
- : int = 17
```

Chaque fonction retourne une unique valeur (qui peut être `()`)

```
#let f x =
  print_int x;
  print_newline();;
f : int -> unit = <fun>
#f 3;;
3
- : unit = ()
```

Style impératif

Le style de programmation *impératif* est proche du fonctionnement d'un ordinateur : on spécifie une série d'*instructions* que l'ordinateur doit effectuer dans un *ordre* donné.

Ces instructions modifient des *variables*, effectuent des branchements...

Les langages C, Pascal, Fortran, ADA ... sont des langages qui utilisent ce style de programmation.

CAML n'est pas un langage impératif (c'est un langage fonctionnel), mais il est doté de certaines caractéristiques qui permettent de mimer la programmation impérative.

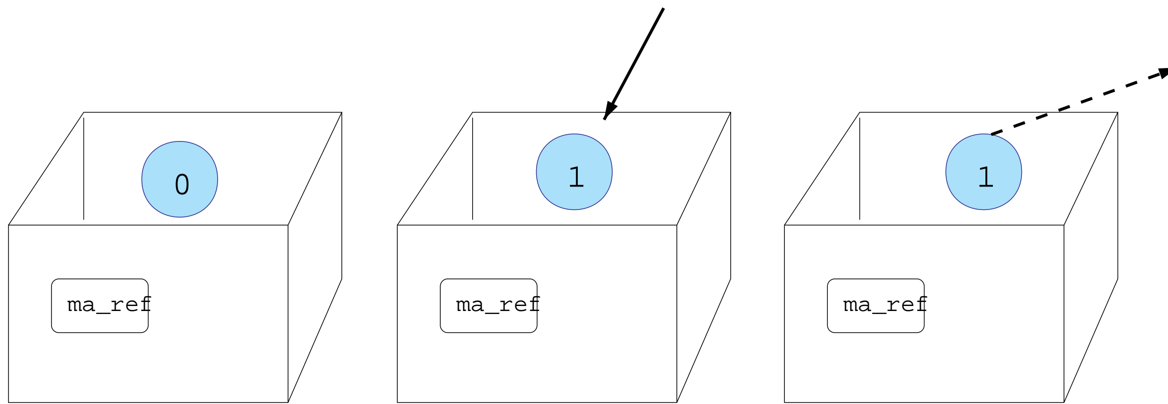


Figure 1: Référence

Références

```
#let ma_ref = ref 0;;

ma_ref : int ref = ref 0

#ma_ref := 1;;

- : unit = ()

#!ma_ref;;

- : int = 1

#ma_ref;;

- : int ref = ref 1
```

Ne pas confondre la référence (`ma_ref` de type `int ref`) avec son contenu (`!ma_ref` de type `int`).

Vecteurs (tableaux)

```
#let tab = [| 1; 2; 3 |];;

tab : int vect = [|1; 2; 3|]

#vect_length tab;;

- : int = 3

#tab.(0);;

- : int = 1

#tab.(2);;

- : int = 3

#tab.(3);;

Exception non rattrapée: Invalid_argument "vect_item"
```

Modification d'un tableau

```
#tab.(1) <- 5;;

- : unit = ()

#tab;;

- : int vect = [|1; 5; 3|]
```

Création d'un vecteur initial

```
#let t = make_vect 10 0;;

t : int vect = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Les vecteurs ne sont jamais recopiés

```
#let t = [|0; 1; 2|];;
t : int vect = [|0; 1; 2|]
#let f tab = tab.(0) <- -1;;
f : int vect -> unit = <fun>
#f t;;
- : unit = ()
#t;;
- : int vect = [|-1; 1; 2|]
```

Attention aussi aux liaisons

```
#let t1 = [| 1; 2; 3 |];;

t1 : int vect = [|1; 2; 3|]

#let t2 = t1;;

t2 : int vect = [|1; 2; 3|]

#t2.(0) <- 5;;

- : unit = ()

#t1;;

- : int vect = [|5; 2; 3|]
```

Structures de contrôle

Test simple

```
#let x = 1;;

x : int = 1

#if x > 0 then print_string "positif";;

positif- : unit = ()
```

Test conditionnel

```
#if x > 0 then
    print_string "positif"
 else
    print_string "négatif";;
```

Blocs `begin ... end` : une seule instruction

```
#let f x =
  if x > 0 then  begin
      print_string "x > 0";
      print_newline()
    end  else  begin
      print_string "x <= 0 ";
      print_newline()
    end ;;
```

Boucles FOR

```
#let i =2;;

i : int = 2

#for i = 1 to 3 do
  print_int i;
  print_newline()
done;;

1
2
3
- : unit = ()

#i;;

- : int = 2
```

Boucles WHILE (tant que)

```
let log_int x  =
  let puiss2 = ref 1 and expo = ref  0 in
    while !puiss2 < x do
      puiss2 := 2 * !puiss2;
      expo := !expo + 1
    done;
    !expo;;
```

Opérateurs booléens

Le ”et” s’écrit `&` ou encore `&&`.

Le ”ou” s’écrit `or` ou encore `||`.

`Exercice 1.` Ecrire une fonction `maximum : int vect -> int` qui détermine le plus grand élément d'un tableau d'entiers.

Preuve de programmes impératifs : invariants de boucles

Preuve de programmes

Nécessité en programmation impérative de *prouver* :

1. La *terminaison* des programmes ;
2. La *correction* des programmes.

Fonctions calculables

Toute fonction calculable par un ordinateur peut être codée par un entier.

Il existe des fonctions $\mathbb{N} \mapsto \mathbb{N}$ qui ne sont pas calculables !

Indécidabilité de la terminaison

S'il existait une fonction universelle `termine : int -> bool` qui prend le code d'une fonction comme argument et qui dit si la fonction se termine, on pourrait considérer la fonction suivante :

```
let absurde () =
  while  (termine code(absurde)) do
  done;;
```

C'est à l'humain de prouver la terminaison de sa fonction !

Preuve de programmes : précondition, postcondition

- *Bloc d'un programme* ;
- *Contexte d'un bloc de programme* ;
- Deux propriétés C_1 et C_2 portant sur le contexte sont appelées *précondition* et *postcondition* d'un bloc B lorsque

$$C_1 \xrightarrow{B} C_2$$

- Théorie de la “ preuve formelle ” de programmes.

Invariant de boucle

C'est une assertion I portant sur le contexte d'une boucle qui est vraie à chaque passage.

$$I \xrightarrow{B} I$$

L'assertion décrit l'état des variables modifiées dans la boucle (références, tableaux ...)

Montrer que la fonction `maximum` est correcte :

```
let maximum t =
  let n = vect_length t in
  let m = ref t.(0) in
    for i = 1 to n - 1 do
      if t.(i) > !m then
        m := t.(i)
    done;
    !m;;
```

Méthode de preuve à l'aide d'un invariant de boucle

1. Définir les *préconditions* (état des variables avant d'entrer dans la boucle) ;
2. Définir un *invariant de boucle* ;
3. Prouver l'invariant (correspond à $\mathcal{P}(n) \Rightarrow \mathcal{P}(n+1)$) ;
4. Montrer la terminaison du programme ;
5. *Condition de sortie de boucle + invariant de boucle ⇒ postcondition.*

Un invariant de boucle est un bon commentaire !

```
(* Renvoie le plus grand élément d'un tableau *)
let maximum t =
  let m = ref t.(0) in
    for i = 1 to vect_length t - 1 do
      if t.(i) > !m then
        m := t.(i);
       (* INV :  m = max{t.(0), ... , t.(i)} *)
    done;
   (* m = max{t.(0), ... , t.(n-1)} *)
    !m;;
```

Schéma de Hörner

$$P = a_0 + a_1 X + \cdots + a_{n-1} X^{n-1},\ a_i \in \mathbb{R}$$

$$p = [|a_0;\ a_1; \ldots;\ a_{n-1}|]$$

$$P(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 = \big((a_3 \times x + a_2) \times x + a_1\big) \times x + a_0$$

```
horner : float vect -> float -> float
```

Multiplication de deux entiers

```
MULTIPLIE x y   (* renvoie xy *)
 m <- 0,
 a <- x,
 b <- y
 TANT QUE b > 0 FAIRE
    SI (b mod 2) = 1 ALORS m <- m + a FIN SI
    a <- a * 2
    b <- b / 2
 FIN TANT QUE
 RETOURNER m
```

À éviter

1. Écrire du code sans réfléchir ;
2. Le tester pour s'apercevoir qu'il ne marche pas ;
3. Modifier le programme en tâtonnant sans savoir où l'on va ;
4. Retester le programme sur quelques exemples et penser qu'il fonctionne ;
5. Ajouter des commentaires bidons.

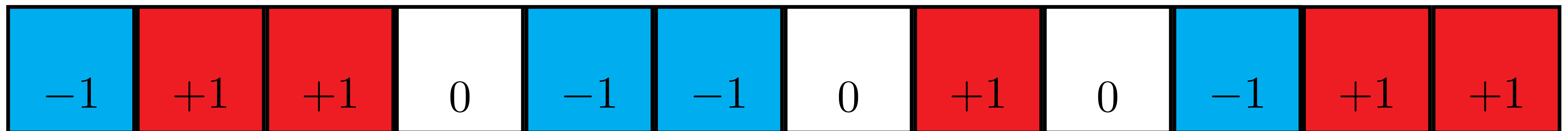
Conséquences

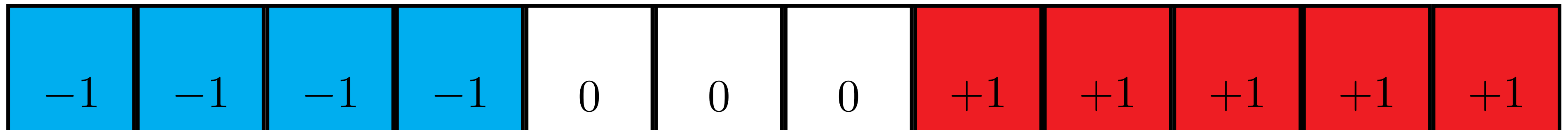
1. Programmes souvent incorrects ;
2. Programmes compliqués et incompréhensibles ;
3. Perte de temps.

Conception à l'aide d'un invariant de boucle

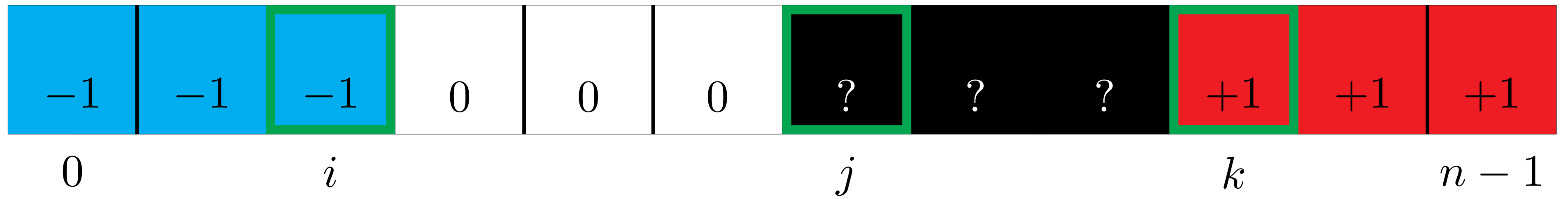
1. Trouver un invariant : il décrit un état *intermédiaire* entre l'état initial des variables et l'état final que l'on veut obtenir ;
2. Écrire les instructions de boucles pour maintenir cet invariant ;
3. Initialiser les variables pour que l'invariant soit vrai au premier passage ;
4. Écrire la condition de sortie de boucle en se servant de l'invariant ;

Drapeau hollandais de Dijkstra





Invariant de boucle



Couple de Bezout

Algorithme d'Euclide :

$$r_0 = a,\ r_1 = b,\ \ \ldots, r_n \neq 0,\ r_{n+1} = 0$$

$$\forall k \in [\![1, n]\!], \quad r_{k-1} = q_k r_k + r_{k+1}, \quad 0 \leq r_{k+1} < r_k$$

$$r_n = \operatorname{pgcd}(a, b)$$

Trouver $(u, v) \in \mathbb{Z}^2$ tels que

$$au + bv = \operatorname{pgcd}(a, b)$$

Programmation fonctionnelle

Style impératif

- Le programmeur spécifie des *instructions* à effectuer dans un *ordre* donné ;
- Les instructions modifient des *variables* (état de la mémoire de la machine modifié) ;
- Preuve : invariants de boucles.

Séquence d'instructions

$$\sigma_0 \longrightarrow \sigma_1 \longrightarrow \cdots \longrightarrow \sigma_n$$

Programmation fonctionnelle

$$f : \left\{ \begin{array}{ccc} A & \longrightarrow & B \\ a & \longmapsto & b \end{array} \right.$$

Style fonctionnel

- Le programmeur définit des *fonctions* ;
- Il *déclare* des propriétés de calcul ;
- Le compilateur se charge d'*évaluer* des *expressions* (réécriture selon des règles), et de *calculer* une *valeur* ;
- Preuves : récurrence, induction structurelle.

Avantages de la programmation fonctionnelle

- Fonctions plus claires, proches de la formulation mathématique du problème ;
- Pas de modifications de mémoire $\Rightarrow$ moins d'erreurs ;
- Le compilateur effectue le travail pénible (cf tours de Hanoï).

Inconvénients de la programmation fonctionnelle

- Performance (?);
- Beaucoup de problèmes résolus naturellement avec des boucles (voir la récursivité terminale) ;
- Gestion des “effets de bord” problématique.

Types fonctionnels

Contraintes sur les types

```
  let f x n =
    if n = x then (n, x)
    else (x, n);;
```

```
f : 'a -> 'a -> 'a * 'a = <fun>
```

Fonctions : objets de premier ordre

```
let minim comp a b =
  if comp a b then a
  else  b;;
```

```
minim : ('a -> 'a -> bool) -> 'a -> 'a -> 'a = <fun>
```

Curryfication

$$\phi : \left\{ \begin{array}{ccc} E \times F & \longrightarrow & G \\ (e,f) & \mapsto & \phi(e,f) \end{array} \right. \hookrightarrow \phi_1 : \left\{ \begin{array}{ccc} E & \longrightarrow & \mathcal{F}(F,G) \\ e & \mapsto & \phi_1(e) \end{array} \right.$$

où $\phi_1(e) : \left\{ \begin{array}{ccc} F & \longrightarrow & G \\ f & \mapsto & \phi(e,f) \end{array} \right. \quad \phi(e,f) = \big[\phi_1(e)\big](f).$

$$\phi_1 : \left\{ \begin{array}{ccc} E & \longrightarrow & \mathcal{F}(F,G) \\ e & \mapsto & \phi_1(e) \end{array} \right. \hookrightarrow \phi : \left\{ \begin{array}{ccc} E \times F & \longrightarrow & G \\ (e,f) & \mapsto & \phi_1(e)(f) \end{array} \right.$$

$$\mathcal{F}(E \times F, G) \approx \mathcal{F}(E, \mathcal{F}(F,G))$$

Types curryfiés

```
let f a b = ...
```

`‘a -> ‘b -> ‘c` ≡ `‘a -> (‘b -> ‘c)`

Mais attention : `(‘a -> ‘b) -> ‘c` ≢ `‘a -> (‘b -> ‘c)`

Associativité à *droite* mais pas à *gauche*.

Exemples

```
#let succ i = i + 1;;
succ : int -> int = <fun>

#let fonc2_zero f  =
  f (f 0);;
fonc2_zero : (int -> int) -> int = <fun>

#fonc2_zero succ ;;
- : int = 2
```

Fonctions anonymes

```
#let plus_n n =
  (fun x -> x + n);;

plus_n : int -> int -> int = <fun>

#plus_n 3;;
- : int -> int = <fun>
#plus_n 3 5;;
- : int = 8
```

Exercice

La fonction suivante est-elle valide, et si oui, quel-est son type ?

```
let f g h =
  fun x y -> h (g x) y;;
```

Les définitions suivantes sont-elles correctes ?

```
let g x = x + 1;;
let h x y = print_int x; print_string y ;;
f g h 3 ;;
```

Filtrage

Filtrage

```
let  f = function
  | 0 -> 0
  | 1 -> 2
  | n -> 2 * n + 1;;
```

En nommant l'argument

```
let f  n =
   match n with
  | 0 -> 0
  | 1 -> 2
  | _ -> 2 * n + 1;;
```

Filtrage de couples

```
let somme x y =
  match (x, y) with
    | (0, _) -> 0
    | (_, 0) -> 1
    | _ -> x + y;;
```

```
#let egalite_composantes = function
    | (x, x) -> true
    | _ -> false;;
```

```
Entrée interactive:
>    | (x, x) -> true
>          ^
L'identificateur x est défini plusieurs fois
dans ce motif.
```

Garde de filtrage

```
let egalite_composantes = function
  | (x, y) when x = y -> true
  | _ -> false;;
```

```
egalite_composantes : 'a * 'a -> bool = <fun>
```

La fonction suivante détermine si l'un des arguments est nul ou si les deux arguments sont égaux :

```
let egalite_ou_zero x y  =
  match (x, y) with
    | (0, _) -> true
    | (_, 0) -> true
    | (x, y) when x = y -> true
    | _ -> false;;
```

Chapitre 2
Fonctions récursives

Récursivité

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * fact(n-1) & \text{si } n > 1 \end{cases}$$

```
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n - 1);;
```

Filtrage pour la récursivité

1. Cas terminaux ;
2. Cas général avec appel récursif.

```
#trace "fact";;

La fonction fact est dorénavant tracée.
- : unit = ()
#fact 4;;
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
- : int = 24
```

Fonctions simultanément récursives

```
let rec pair n =
  match n with
    | 0 -> true
    | _ -> impair (n - 1)
and
  impair n =
  match n with
    | 0 -> false
    | _ -> pair (n - 1);;
```

Thèse de Church

Tout ce qui peut être calculé par une procédure impérative peut être calculé par des fonctions récursives et réciproquement.

Terminaison d'une fonction récursive

Une fonction qui ne se termine pas

```
let rec f n =
  match n with
    | 0 -> 1
    | _ -> f (n + 1);;

f 1;;
```

Conjecture de Collatz (problème de Syracuse)

```
let rec collatz  = function
    | n when n <= 1 -> 0
    | n when n mod 2 = 0 -> collatz (n / 2)
    | n -> collatz (3 * n + 1);;
```

```
collatz 3
```

$$3 \mapsto 10 \mapsto 5 \mapsto 16 \mapsto 8 \mapsto 4 \mapsto 2 \mapsto 1 \mapsto 0$$

Définition 1. **Ordre bien fondé**

Soit E un ensemble, et $\preceq$ une relation d'ordre sur E (pas nécessairement totale). On note $\prec$ l'ordre strict correspondant :

$$\forall x \in E, \quad x \prec y \Longleftrightarrow x \preceq y \text{ et } x \neq y$$

On dit que l'ordre $\preceq$ est *bien fondé* s'il n'existe pas de suite d'éléments de E strictement décroissante.

Élément minimal

Soit un ensemble ordonné $(E, \preceq)$ et une partie $A \subset E$. Soit $m \in A$. On dit que m est *minimal* dans A lorsque :

$$\forall a \in A, \quad a \preceq m \Rightarrow a = m$$

Remarque 1. Si l'ordre est total,

$$m \text{ minimal dans } A \iff m = \min A$$

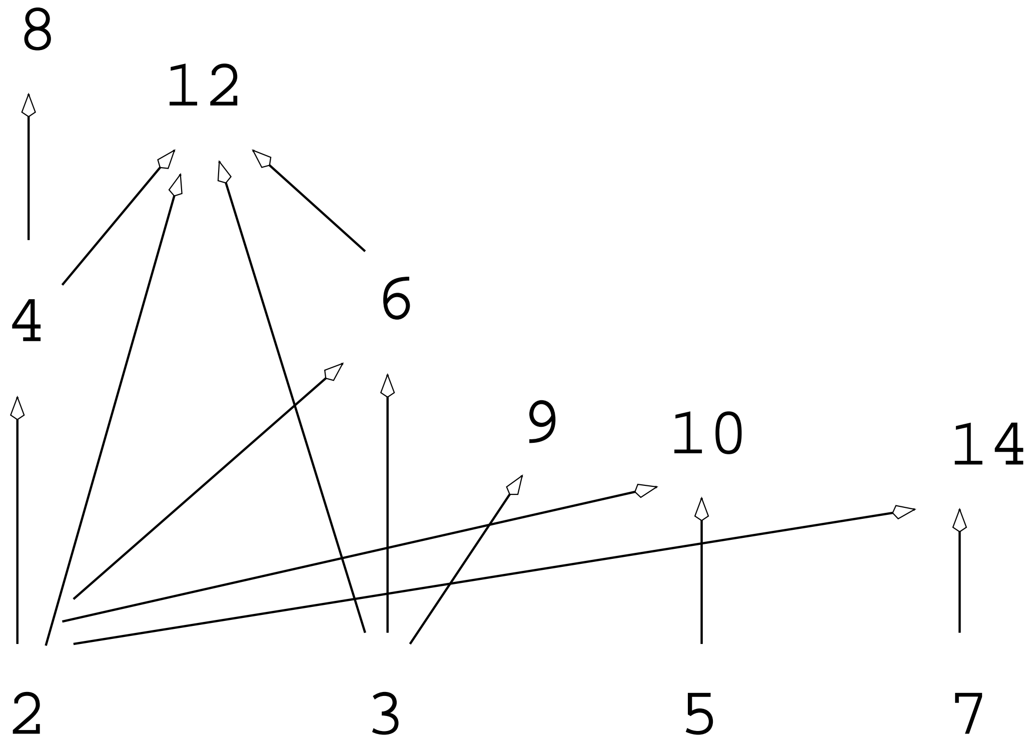
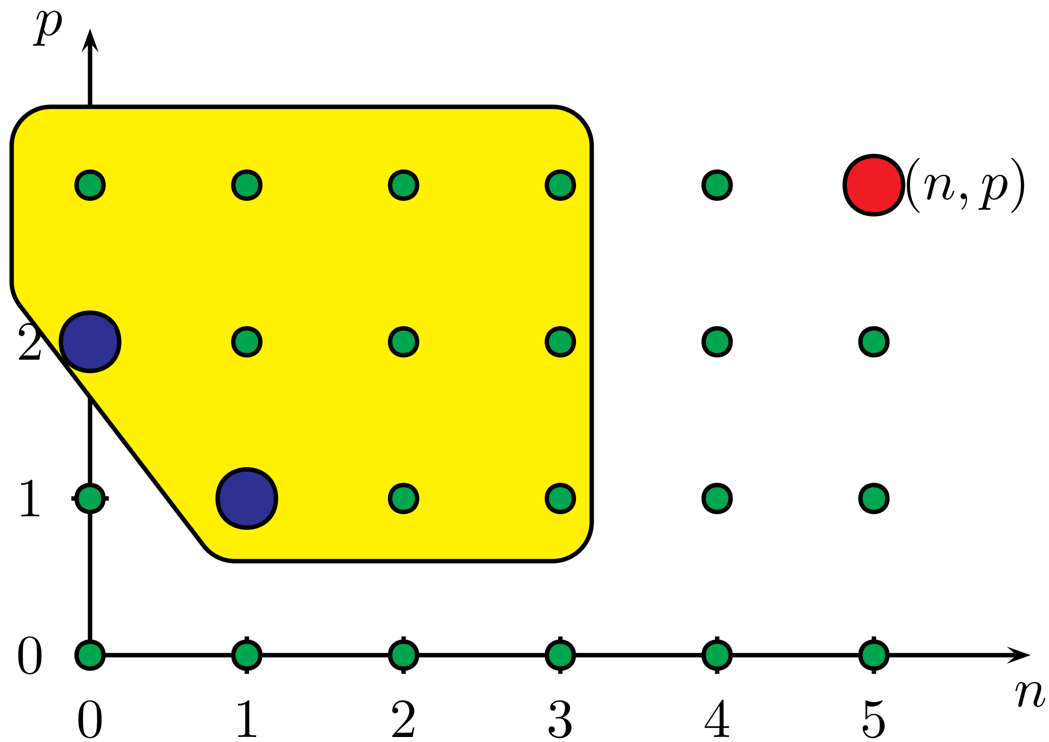


Figure 2: Ordre de la divisibilité sur $\mathbb{N} \setminus \{0, 1\}$

Caractérisation d'un ordre bien fondé

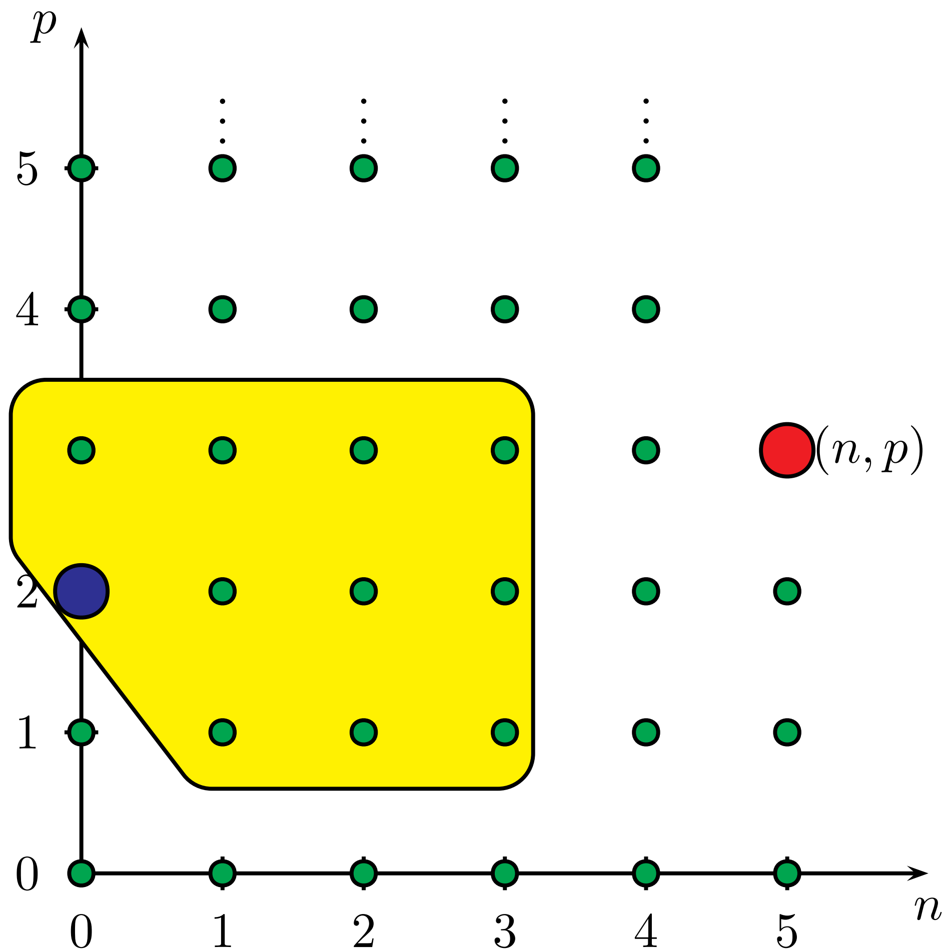
Théorème 1. *L'ordre* $\preceq$ *est bien fondé si et seulement si toute partie* A *non-vide de* E *possède un élément minimal.*

Ordre produit sur $\mathbb{N}^2$: $(n,p) \preceq (n',p') \iff n \leq n'$ et $p \leq p'$.



Ordre lexicographique sur $\mathbb{N}^2$:

$(n,p) \preceq (n',p') \iff n \leq n'$ ou $n = n'$ et $p \leq p'$.



Théorème 2. ***Induction sur un ensemble bien fondé***

Soit $(E, \preceq)$ *un ensemble bien fondé, et* $\mathbf{p}$ *un prédicat sur* E*. On note* $\mathcal{B}$ *l'ensemble des éléments minimaux de* E*. Si l'on a :*

1. $\forall x \in \mathcal{B},\ \mathbf{p}(x)$,
2. $\forall x \in E,\ \big(\forall y \prec x,\ \mathbf{p}(y)\big) \Rightarrow \mathbf{p}(x)$,

Alors $\forall x \in E,\ \mathbf{p}(x)$.

Théorème 3. ***Justification de la terminaison d'une fonction récursive***

Soit $(E, \preceq)$ *un ensemble muni d'un ordre bien fondé,* $f : A \mapsto B$ *une fonction récursive et* $\phi : A \mapsto E$ *une application. On note*

$$\mathcal{M} = \{x \in A \ tq \ \phi(x) \ est \ minimal \ dans \ \phi(A)\}$$

On fait les hypothèses suivantes :

1. *le calcul de* $f(x)$ *se termine pour tous les éléments* $x \in \mathcal{M}$ *;*
2. *pour tout* $x \in A$*, le calcul de* $f(x)$*, n'utilise qu'un nombre fini de calculs* $f(y_1) \ldots f(y_k)$ *où* $\phi(y_i) \prec \phi(x)$*.*

Alors le calcul de $f(x)$ *se termine pour toute valeur de* $x \in A$*.*

Graduation

Lorsque $\phi : A \mapsto (\mathbb{N}, \leq)$, on dit que ϕ est une *graduation*.

```
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n - 1);;
```

$\phi(n) = n$ peut être utilisé.

Théorème 4. ***Preuve de correction d'une fonction récursive***

Soit $f : A \mapsto B$ *une fonction récursive, et* $\phi : A \mapsto E$ *une application où* $\preceq$ *est un ordre bien fondé sur* E*. On note*

$$\mathcal{M} = \{x \in A \mid \phi(x) \text{ est minimal dans } \phi(A)\}$$

Soit $\mathbf{p_f}(x)$ *un prédicat faisant intervenir* $f(x)$*. On fait les hypothèses suivantes :*

1. $\forall x \in \mathcal{M}$, $\mathbf{p_f}(x)$ *est vrai ;*
2. *Si* $x \in A$*, le calcul de* $f(x)$ *n'utilise qu'un nombre fini de calculs de* $f(y_1) \dots f(y_k)$ *avec* $\phi(y_1) \prec \phi(x), \dots, \phi(y_k) \prec \phi(x)$ *et*

$$\mathbf{p_f}(y_1) \dots \mathbf{p_f}(y_k) \Rightarrow \mathbf{p_f}(x)$$

alors, $\forall x \in A$, $\mathbf{p_f}(x)$ *est vrai.*

1. Trouver un ordre bien fondé adapté au problème ;
2. Quels sont les cas de base pour lesquels on renvoie directement le résultat ? Les éléments “minimaux” sont-ils bien inclus dans les cas de base ?
3. (La partie cruciale). Soit x un élément arbitraire. En supposant que l'on sache calculer $f(y)$ pour tous les éléments $y \prec x$, comment en déduire le calcul de $f(x)$ en n'utilisant qu'un nombre fini d'éléments strictement inférieurs ?
4. La terminaison et la correction de la fonction découlent des raisonnements précédents.

Ecrire une fonction `binomial : int -> int -> int` telle que `binomial n p` retourne l'entier $\binom{n}{p}$,

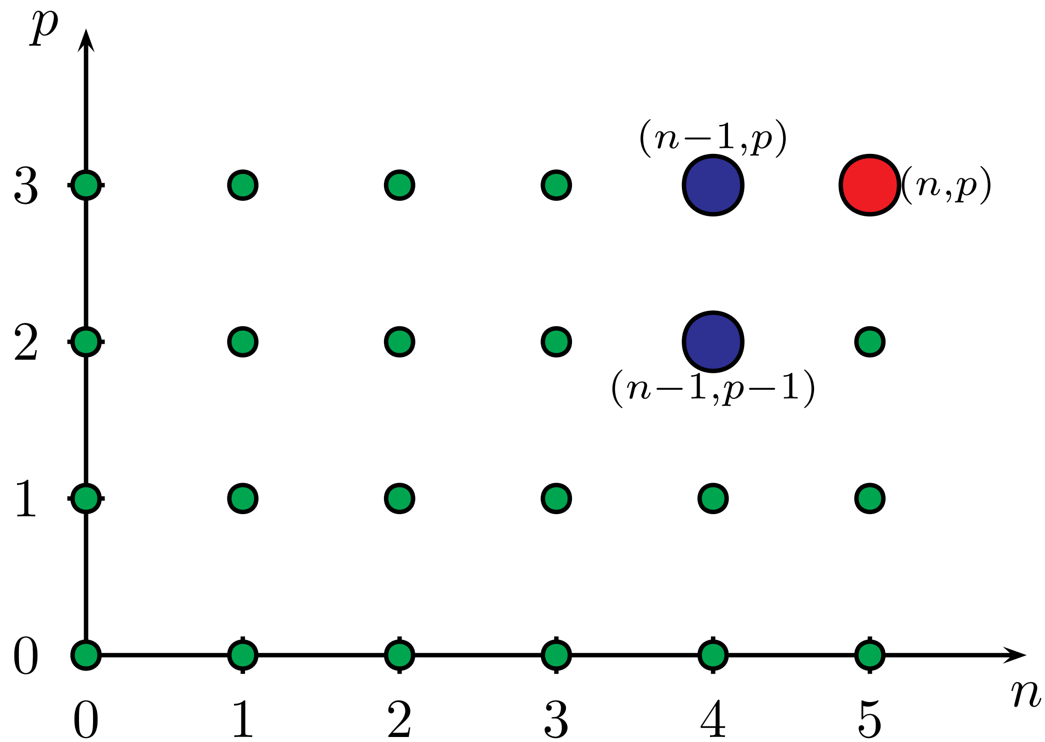
a) en utilisant la propriété :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

b) en utilisant la propriété :

$$\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$$

Ordre produit sur $\mathbb{N}^2$: $(n-1,p-1) \prec (n,p)$ et $(n-1,p) \prec (n,p)$:



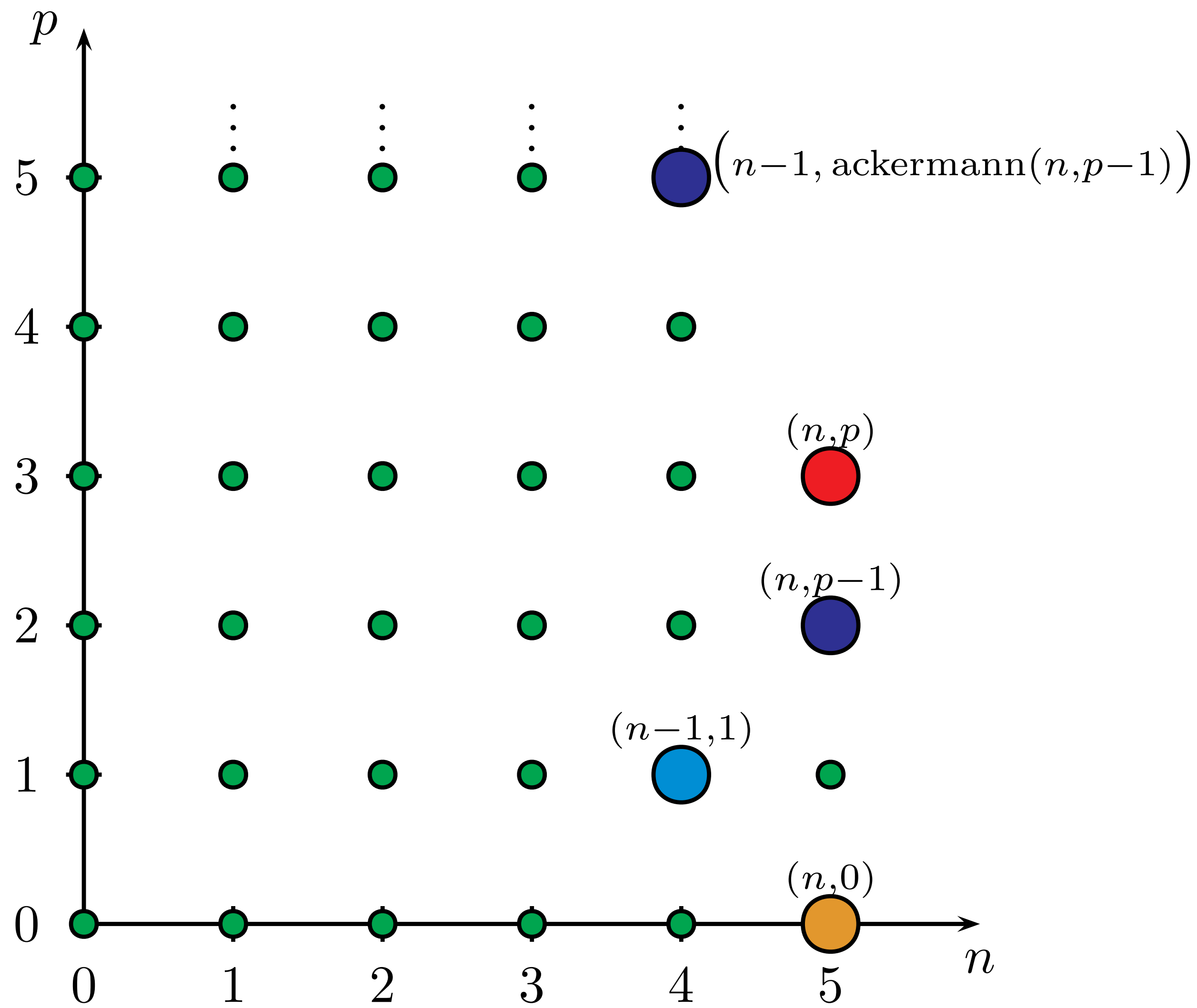
```
let rec binomial n p =
  match (n, p) with
    | _, 0 -> 1
    | 0, _ -> 0
    | _ -> (binomial (n - 1) (p - 1) )
          + (binomial (n - 1) p);;

let rec binomial n p =
  match (n, p) with
    | _, 0 -> 1
    | 0, _ -> 0
    | _ -> (n * binomial (n - 1) (p - 1)) / p;;
```

Fonction d'Ackermann

```
let rec ackermann n p =
  match (n, p) with
    | 0, p -> p + 1
    | n, 0 -> ackermann (n - 1) 1
    | n, p -> ackermann (n - 1) (ackermann n (p - 1));;
```

Montrer la terminaison de cette fonction.

Ordre lexicographique sur $\mathbb{N}^2$: $\big(n-1, \text{ackerman}(n,p-1)\big) \prec (n,p)$

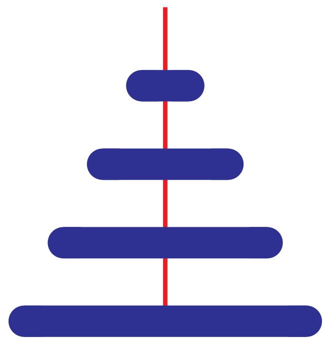


La fonction d'Ackermann est célèbre en informatique pour sa croissance extrêmement rapide :

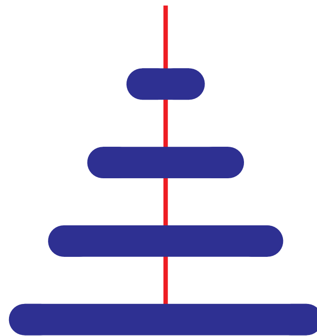
```
let rec ackermann n p =
  match (n, p) with
    | 0, p -> p + 1
    | n, 0 -> ackermann (n - 1) 1
    | n, p -> ackermann (n-1) (ackermann n (p - 1));;
```

Montrer que $\forall (n,p) \in \mathbb{N}^2$, $\text{ackermann}(n,p) > p$.

Tours de Hanoï : hanoi *"A"* *"B"* *"C"* n



tige A tige B tige C



tige A tige B tige C

Affichage des déplacements

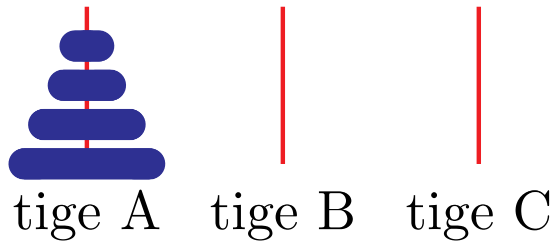
```
let deplace de vers =
print_string  (de ^ " --> " ^ vers );
print_newline();;
```

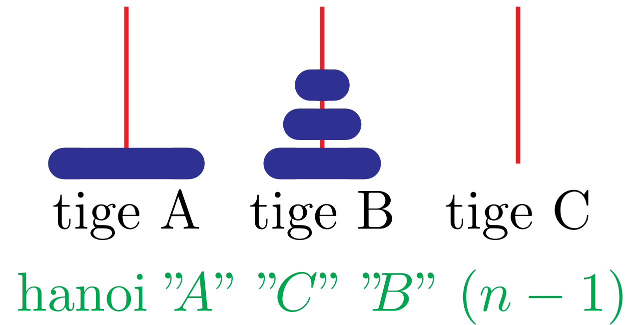
```
deplace : string -> string -> unit = <fun>
```

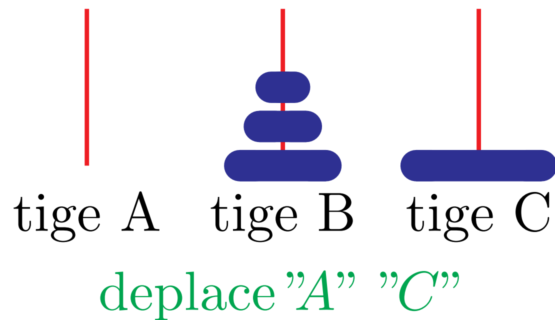
```
#deplace "A" "B";;
```

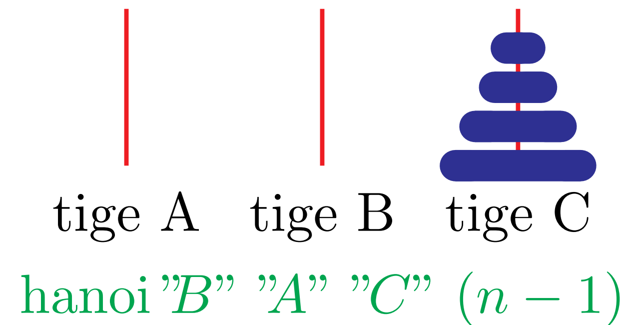
```
A --> B
- : unit = ()
```

Raisonnement inductif : `hanoi orig milieu dest n`









Fonction hanoï

```
let rec hanoi orig  milieu dest n =
  match n with
    | 0 -> ()
    | _ ->
        hanoi orig dest milieu (n - 1);
        deplace orig dest;
        hanoi milieu orig dest (n - 1);;
```

```
hanoi : string -> string -> string -> int -> unit = <fun>
```

Induction structurelle

1. On se donne un ensemble $\mathcal{B}$ d'objets de *base* ;
2. On se donne un ensemble de *constructeurs*. A partir de k objets $x_1, \ldots, x_k$, un constructeur produit un nouvel objet

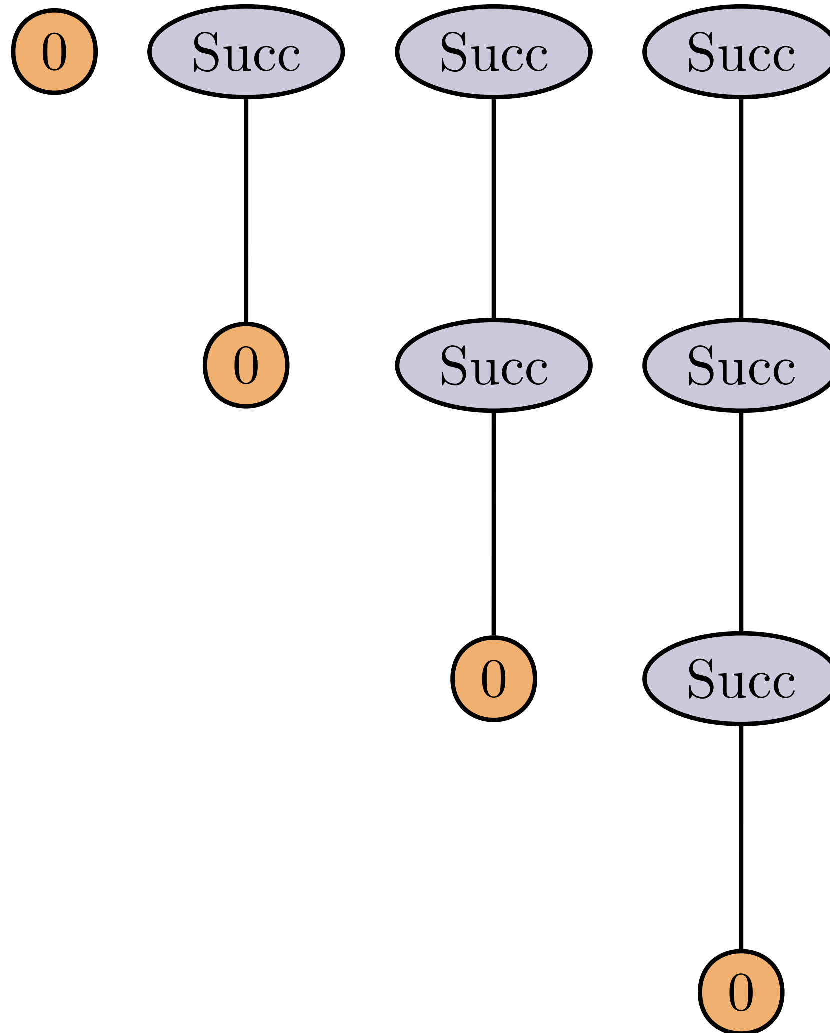
   $$\mathbf{C}(x_1, \ldots, x_k)$$

   Le constructeur $\mathbf{C}$ est dit d'*arité* k s'il s'applique à k objets ;
3. $\mathcal{I}$ est le plus petit ensemble qui contient tous les objets obtenus par application d'un nombre fini de fois les constructeurs à partir des objets de la base $\mathcal{B}$.

On peut définir les entiers naturels en les “comptant” de la façon suivante :

1. Un seul objet de base, l’objet 0 ;
2. Un seul constructeur d’arité 1 : `Succ` qui prend un objet et qui lui associe son “successeur”.

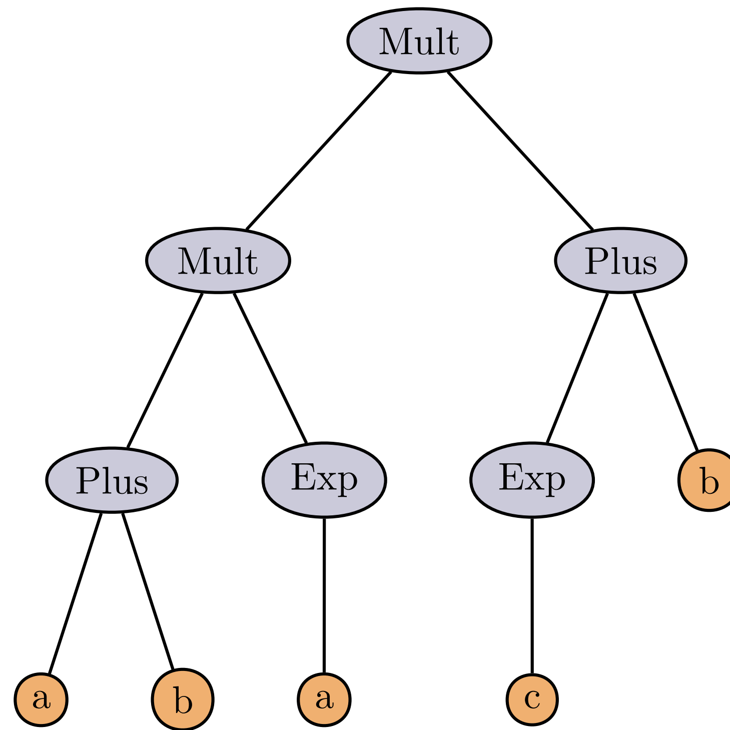
Représentations arborescentes des entiers 0, 1, 2, et 3



En calcul formel, on peut définir une “expression algébrique” (simplifiée) de la façon suivante :

- On définit l'ensemble $\mathcal{B}$ des “variables” ;
- On définit les constructeurs suivants :
  - `Plus` (d'arité 2) : Si $A1$ et $A2$ sont deux expressions algébriques, `Plus(A1, A2)` est encore une expression algébrique,
  - `Mult` (d'arité 2) : Si $A1$ et $A2$ sont deux expressions algébriques, `Mult(A1, A2)` est encore une expression algébrique,
  - `Exp` (d'arité 1) : Si A est une expression algébrique, `Exp A` est encore une expression algébrique.

$$\Big((a+b)*exp(a)\Big)*\Big(exp(c)+b\Big)$$



Théorème 5. ***Raisonnement par induction structurelle***

Soit $\mathcal{P}(x)$ un prédicat sur un ensemble $\mathcal{I}$ défini par induction structurelle. Si :

1. *$\mathcal{P}(x)$ est vrai pour tout élément x de $\mathcal{B}$;*
2. *Pour tout constructeur $\boldsymbol{C}$ d'arité k, pour tout k-uplet d'éléments $(x_1, \ldots, x_k)$ de $\mathcal{I}$:*

$$\Big(\forall i \in [1, k], \mathcal{P}(x_i) \text{ vraie}\Big) \Rightarrow \Big(\mathcal{P}\big(\boldsymbol{C}(x_1, \ldots, x_k)\big) \text{ vraie }\Big)$$

Alors la propriété $\mathcal{P}$ est vraie pour tout élément de $\mathcal{I}$.

- Vérifier la propriété sur les objets de base ;
- Vérifier que la propriété est encore vraie après application de chaque règle de production.

Listes

Si α est un type connu, on définit *inductivement* les listes d'éléments de type α par :

1. Elément atomique : la liste vide, notée NIL ou [] ;
2. Une infinité de constructeurs paramétrés `CONS_x` pour tout x de type α : `CONS_x(q)` est une liste notée `x :: q` en CAML .

1. `l0 = []` ;

2. `l1 = CONS_4(l0) = 4 :: []` ;

3. `l2 = CONS_1(l1) = 1 :: 4 :: []` ;

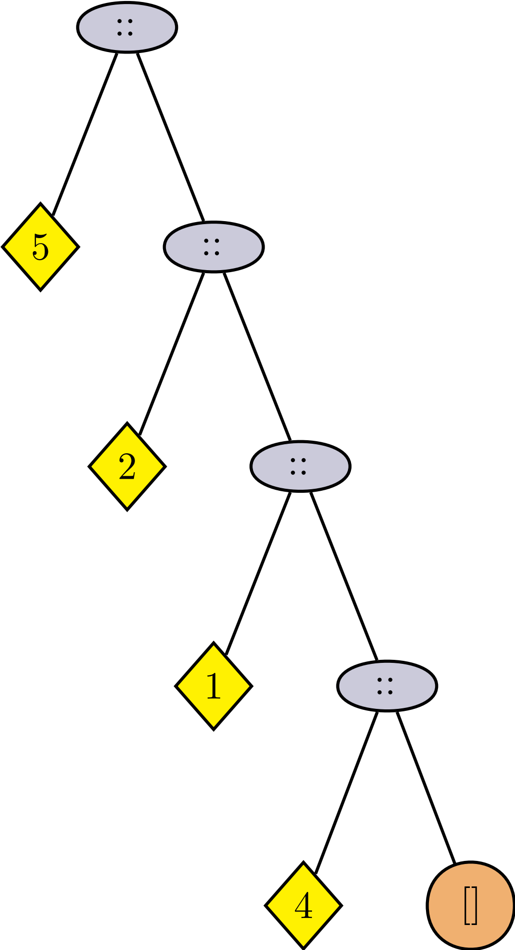
4. `l3 = CONS_2(l2) = 2 :: 1 :: 4 :: []` ;

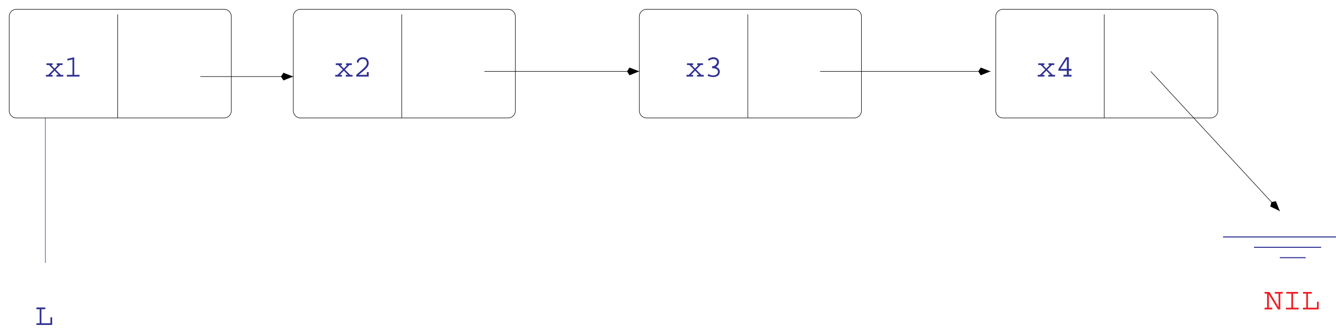
5. `l4 = CONS_5(l3) = 5 :: 2 :: 1 :: 4 :: []`.

Listes en Caml

```
#let l = [0; 2; 3] ;;

l : int list = [0; 2; 3]

#tl l;;

- : int list = [2; 3]

#hd l;;

- : int = 0

#4 :: l;;

- : int list = [4; 0; 2; 3]
```

la liste `[5; 2; 1; 4]` :




x1
x2
x3
x4
L
NIL

	Vecteurs	Listes
Taille	Fixe	Dynamique
Accès (temps constant)	À chaque élément	Au premier élément

Filtrage sur les listes

Une liste est soit :

- vide ;
- de la forme `x :: q` où q est la *queue* (une liste) et x la *tête* (un élément).

```
let rec f l =
match l with
| [] -> ...
| x :: q -> ... ;;
```

Raisonnement inductif sur les listes

1. On vérifie que la fonction renvoie le bon résultat pour la liste vide ;
2. On suppose que la fonction renvoie le bon résultat pour une liste q et on montre que la fonction renvoie le bon résultat pour la liste `x :: q`.

Alors la fonction se termine et renvoie le bon résultat pour toute liste.

Longueur d'une liste

```
let rec longueur l =
  match l with
    | [] -> 0
    | x :: q -> 1 + longueur q;;
```

```
longueur : 'a list -> int = <fun>
```

(fonction `list_length` en Caml).

mem

l : une liste d'éléments de type `'a`, x : un élément de type `'a'`.

Écrire un prédicat (fonction à valeur dans `bool`) qui teste si x est dans la liste.

La fonction `mem` est dans la bibliothèque Caml .

Ecrire une fonction qui concatène deux listes

```
#concatene [1; 3] [5; 6];;

- : int list = [1; 3; 5; 6]
```

- $l_1 = [x_1; x_2; \ldots; x_n]$, $l_2 = [y_1; \ldots; y_p]$;
- $l_1 = x :: q$ avec $x = x_1$ et $q = [x_2; \ldots; x_n]$;
- hypothèse d'induction : concatene q $l2 = [x_2; \ldots; x_n; y_1; \ldots; y_n]$.

```
let rec concatene l1 l2 =
  match l1 with
    | [] -> l2
    | x :: q -> x :: concatene q l2;;
```

```
concatene : 'a list -> 'a list -> 'a list = <fun>
```

Nombre d'appels récursifs $T(n, p)$?

Opérateur de concaténation : `@`

```
#let l1 = [1; 2]
and l2 = [3; 4]
in l1 @ l2;;
```

```
- : int list = [1; 2; 3; 4]
```

Ecrire une fonction qui insère un élément à la fin d'une liste.

```
#insere_fin 7 [1; 2; 3];;
- : int list = [1; 2; 3; 7]
```

Insertion à la fin d'une liste

- $l = [x_1; x_2; \ldots; x_n]$, $q = [x_2; \ldots; x_n]$;
- hypothèse d'induction : insere_fin $a\ q = [x_2; \ldots; x_n; a]$.

```
let rec insere_fin a l =
match l with
  | [] -> [a]
  | x :: q -> x :: (insere_fin a q);;
```

Nombre d'appels récursifs $S(n)$?

Image miroir d'une liste

En utilisant la fonction précédente, écrire une fonction qui renvoie l'image miroir d'une liste :

```
#miroir [1; 2; 3; 4];;
- : int list = [4; 3; 2; 1]
```

Image miroir d'une liste

- $l = [x_1; x_2; \ldots; x_n]$, $q = [x_2; \ldots; x_n]$;
- hypothèse d'induction : miroir $q = [x_n; \ldots; x_2]$.

```
let rec miroir l =
  match l with
    | [] -> []
    | x :: q -> insere_fin x (miroir q);;
```

Nombre d'appels récursifs $T(n)$?

Filtrage sur les listes à plus d'un élément

```
let f l =
match l with
| [] -> ...
| [x] -> ...
| x :: y :: q -> ... ;;
```

Ecrire une fonction `sont_egaux` de type `‘a list -> bool` qui teste si tous les éléments d’une liste sont égaux.

Filtrage sur un couple de listes

```
let f l1 l2 =
match (l1, l2) with
| [], [] -> ...
| [], _ -> ...
| _, [] ->
| x1 :: q1, x2 :: q2 -> ... ;;
```

Ecrire une fonction `egales` de type `‘a list -> ‘a list -> bool` qui teste si deux listes sont égales.

Récursivité terminale

Nombres de Fibonacci

```
let rec fibo n =
match n with
| 0 -> 1
| 1 -> 1
| _ -> fibo (n - 1) + fibo (n - 2);;
```

$T(0) = T(1) = 1$ et $T(n) = T(n-1) + T(n-2)$.

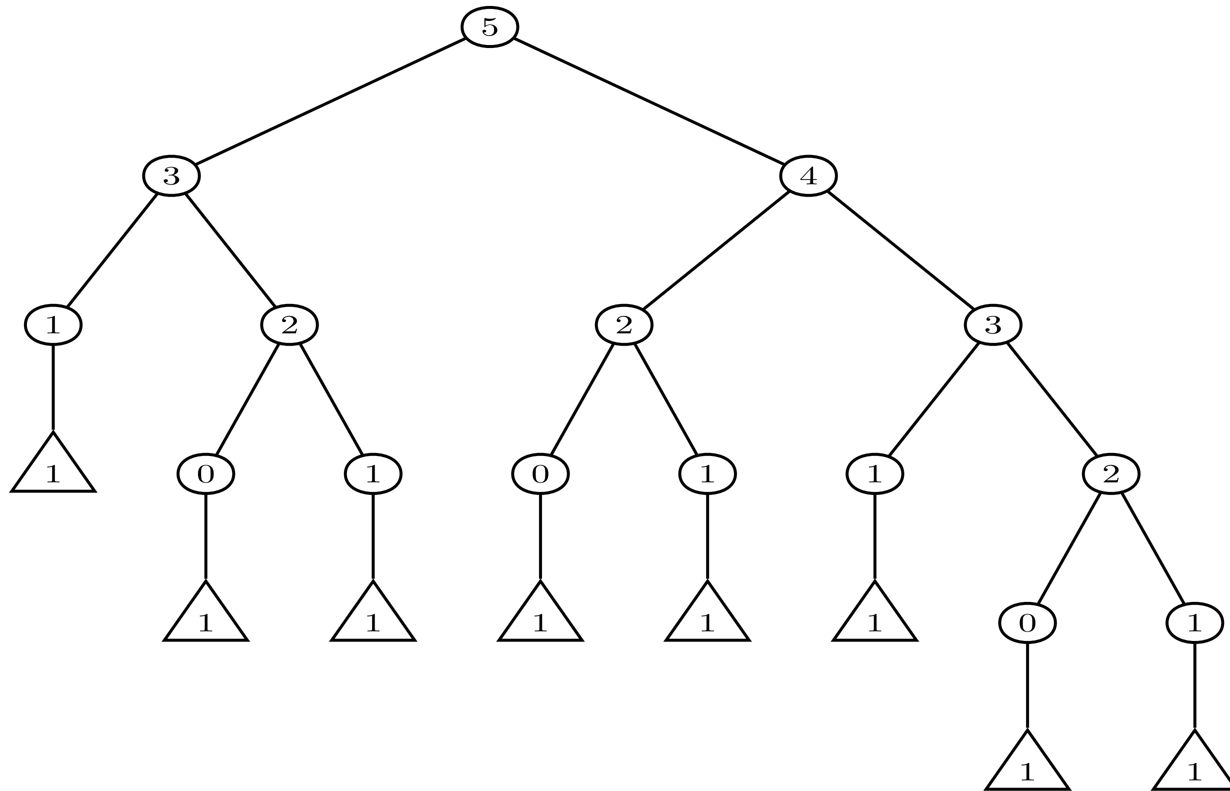


Figure 3: Appels récursifs de fibo 5

Pile d'exécution

```
let rec fact n =
  match n with
    | 0 -> 1
    | n -> n * fact (n - 1);;
```

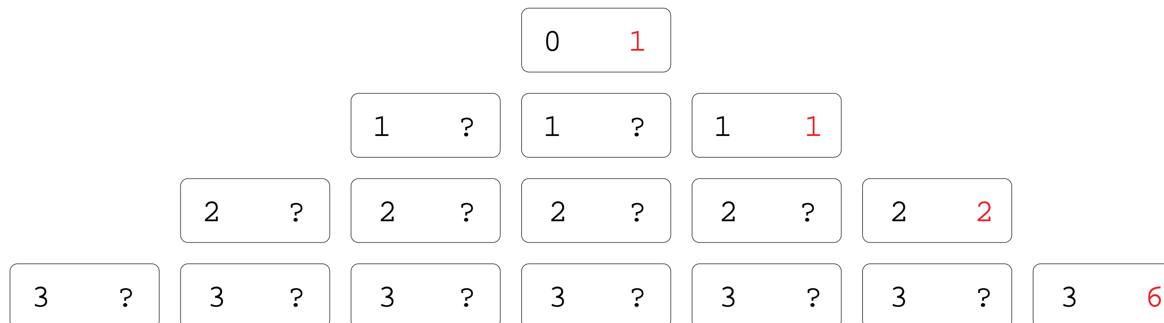


Figure 4: Calcul de 3!

Définition 2. **Fonctions récursives terminales**

On dit qu'une fonction récursive est *terminale* si le seul appel récursif qu'elle fait se trouve en dernière position.

```
let rec terminale n =
  match n with
  | 0 -> print_newline()
  | n -> print_int n; terminale (n - 1);;

let rec  non_terminale n =
  match n with
  | 0 -> print_newline();
  | n ->  non_terminale (n - 1); print_int n;;
```

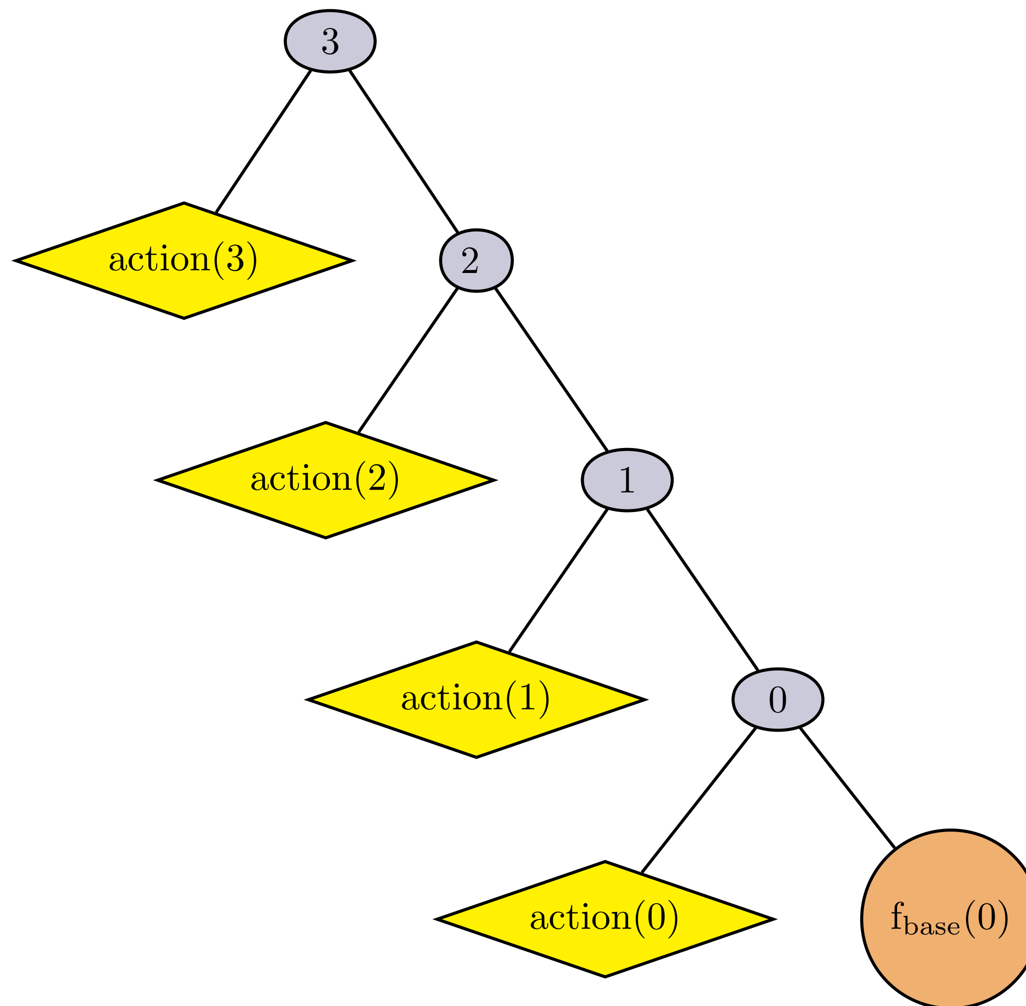
Dérécursification

Transformation d'une fonction récursive terminale

```
let rec f x =
  match cas_de_base x with
  | true -> f_base x;
  | false  -> action x; f (sigma x);;
```

```
let proc x =
  let arg = ref x in
  while not (cas_de_base !arg) do
     action !arg;
     arg := sigma !arg
  done;
  f_base !arg;;
```

Arbre des appels récursifs d'une fonction récursive terminale



```
let rec terminale n =
   match n with
   | 0 -> print_newline()
   | n -> print_int n; terminale (n - 1);;

let terminale_imp n =
  let arg = ref n in
    while !arg <> 0 do
      print_int !arg;
      arg := !arg - 1
    done;
    print_newline();;
```

Toute fonction récursive est équivalente à une procédure impérative utilisant une pile et une boucle $\Rightarrow$ théorie de la *dérécursification*.

Importance dans l'analyse d'algorithmes.

Factorielle récursive terminale

```
let rec fact n =
  match n with
    | 0 -> 1
    | _ -> n * fact (n - 1);;
```

Ecrivons une fonction auxiliaire avec un *accumulateur* qui calcule accu $\times n \times (n-1) \times \cdots \times 1$:

```
let rec fact_t n accu =
  match n with
    | 0 -> accu
    | n -> fact_t (n - 1) (accu * n);;
```

```
let fact n =
  let rec fact_t n accu =
    match n with
      | 0 -> accu
      | n -> fact_t (n - 1) (accu * n)
  in
    fact_t n 1;;
```

```
let fact  =
  let rec fact_t  accu n =
    match n with
      | 0 -> accu
      | n -> fact_t (accu * n) (n - 1)
  in
    fact_t 1;;
```

```
fact_t : int -> int -> int
fact_t 1 : int -> int
```

Image miroir d'une liste

Image miroir déjà écrite

```
let rec miroir l =
  match l with
    | [] -> []
    | x :: q -> (miroir q) @ [x];;
```

Complexité : $T(0) = 0$, $T(n) = T(n-1) + (n-1) \Rightarrow T(n) = \Theta(n^2)$

Image miroir d'une liste avec accumulateur

`l = [l_0; ...; l_n]`, `accu = [a_0; ... ; a_p]`

`miroir_aux l accu : [ l_n; ... ; l_0; a_0; ... ; a_p ].`

```
let rec miroir l =
let rec miroir_aux l accu =
  match l with
    | [] -> accu
    | x :: q -> miroir_aux q (x :: accu)
in
  miroir_aux l [];;
```

$l = [l_0; l_1; \ldots; l_n] = l_0 :: q$, $accu = [a_0; \ldots; a_p]$,

$$\text{miroir_aux}\ q\ (x :: accu) \ = \ [l_n; \ldots; l_1; \underbrace{l_0; a_0; \ldots; a_p]}_{l_0 :: accu}$$

```
let  miroir  =
  let rec miroir_aux accu l =
    match l with
      | [] -> accu
      | x :: q -> miroir_aux (x :: accu) q in
    miroir_aux [];;
```

Nombre d'appels récursifs de `miroir_aux` : $T(n, p)$?

Ecrire une fonction `bits` qui renvoie la liste des chiffres en base 2 d'un entier n :

$$n = a_p.2^p + \cdots + a_1.2 + a_0 \mapsto [a_p; \ldots; a_0]$$

```
let bits n =
  let rec bits_aux n accu =
    match n with
      | 0 -> accu
      | _ -> bits_aux (n / 2) ((n mod 2) :: accu) in
    bits_aux n [];;

let  bits  =
  let rec bits_aux  accu n =
    match n with
      | 0 -> accu
      | _ -> bits_aux  ((n mod 2) :: accu) (n / 2) in
    bits_aux  [];;
```

Chapitre 3

Analyse mathématique des algorithmes

Complexité

- Complexité temporelle : temps d'exécution d'un algorithme ;
- Complexité spatiale : espace mémoire nécessaire à l'exécution d'un algorithme.

Comparaison des algorithmes

- On peut “compter” précisément toutes les opérations élémentaires, mesurer le coût d’une opération élémentaire sur un ordinateur et en déduire le temps d’exécution (long et pas très utile) ...
- Définir la notion de “taille” du problème (en général un entier) (nombre de bits, longueur d’un tableau ou d’une liste ...) ;
- Choisir une mesure approximative de la complexité (nombre d’appels récursifs, nombre de multiplications, de comparaisons ...) adaptée à l’algorithme et donner un “ordre de grandeur” de la complexité pour de grandes données.

Trois types de complexités intéressantes

On note D_n l'ensemble des données du problème de "taille" n, et $C(d)$ le coût de traitement d'une donnée d.

- Complexité *dans le pire des cas* :

$$\mathrm{Cmax}(n) = \max\{C(d)\,;\, d \in D_n\}$$

- Complexité *dans le meilleur des cas* :

$$\mathrm{Cmin}(n) = \min\{C(d)\,;\, d \in D_n\}$$

- Complexité *en moyenne* :

$$\mathrm{Cmoy}(n) = \sum_{d \in D_n} p(d)C(d)$$

où $p(d)$ représente la probabilité d'apparition de la donnée d parmi toutes les données de taille n.

Notations de Landau

- On dit que (u_n) est *dominée* par (v_n) et l'on note

$$u_n = O(v_n)$$

  lorsque : $\exists M > 0, \exists n_0 \in \mathbb{N}$ tel que $\forall n \geq n_0, \quad u_n \leq M v_n$;

- On dit que (u_n) *domine* (v_n) lorsque $v_n = 0(u_n)$ et l'on note

$$u_n = \Omega(v_n)$$

- On dit que (u_n) et (v_n) sont de *même ordre* et l'on note

$$u_n = \Theta(v_n)$$

  lorsque $u_n = O(v_n)$ et $v_n = O(u_n)$;

Complexités usuelles

- complexité *logarithmique* : en $\Theta(\log n)$ (très efficace) ;
- complexité *linéaire* : en $\Theta(n)$ (efficace) ;
- complexité *quasi-linéaire* : en $O(n \ln n)$ mais pas en $O(n)$ (efficace) .
- complexité *polynomiale* : en $O(n^k)$ (moyennement efficace lorsque k est inférieur à 3, inefficace sinon) ;
- complexité *exponentielle* : $T(n) \geq a^n$ avec $a > 1$ (totalement inefficace sauf pour de petites données).

Ordre de grandeur des temps de calcul

En supposant qu'une opération élémentaire prend $1\ \mu\ s = 10^{-6}\ s$:

	$\log_2 n$	n	$n.log_2n$	n^2	n^3	2^n
$n = 10^2$	6,6 μs	0,1 ms	0,6 ms	10 ms	1 s	4×10^{16} a
$n = 10^3$	9,9 μs	1 ms	9,9 ms	1 s	16,6 min	∞
$n = 10^4$	13,3 μs	10 ms	0,1 s	100 s	11,5 j	∞
$n = 10^6$	19,9 μs	1 s	19,9 s	11,5 j	$31,7 \times 10^3$ a	∞

On note ∞ lorsque la valeur dépasse 10^{100} a (a : années).

Complexité quasi-linéaire

$$T(n) = n \log_2 n$$

Lorsque $n \approx 10^6$, $\log_2 n \approx 20$. Donc $\log_2 n$ se comporte comme une constante lorsque n est de taille humaine.

Ex : traitement d'une image : $n = 1024 \times 768 = 7,810^5$.

- Transformée de Fourier en n^2 : $T \sim 172h$;
- Transformée de Fourier rapide en $n \log n$: $15,4s$.

Taille maximale des données en fonction du temps de calcul

	$\log_2 n$	n	$n.log_2n$	n^2	n^3	2^n
1 s	∞	10^6	$6,3 \times 10^4$	10^3	100	19
1 min	∞	6×10^7	$2,8 \times 10^6$	$7,7 \times 10^3$	390	25
1 h	∞	$3,6 \times 10^9$	$1,3 \times 10^8$	6×10^4	$1,5 \times 10^3$	31
1 j	∞	$8,6 \times 10^{10}$	$2,7 \times 10^9$	$2,9 \times 10^5$	$4,4 \times 10^3$	36

Evolution de la taille maximale si la puissance de la machine est multipliée par un facteur α

$\log_2 n$	n	$n.log_2 n$	n^2	n^3	2^n
n^α	$\alpha \times n$	$\sim \alpha \times n$	$\sqrt{\alpha} \times n$	$\sqrt[3]{\alpha} \times n$	$n + \log_2 \alpha$

Si la machine calcule 100 fois plus vite, la taille maximum des données pour traiter un problème exponentiel en un jour ne progresse que de 6 unités !

Recherche d'un élément dans un tableau

Problème : Chercher un élément x dans un tableau $t = [|t_0; \ldots; t_{n-1}|]$

Recherche linéaire

```
let cherche x t =
  let n = vect_length t
  and i = ref 0
  and trouve = ref false
  in
    while (!i < n && not !trouve) do
      trouve := ( t.(!i) = x );
      i := !i + 1
    done;
    !trouve;;
```

```
cherche : 'a -> 'a vect -> bool = <fun>
```

Nombre d'accès au tableau dans le pire des cas ?

Complexité moyenne de la recherche linéaire

Hypothèses

1. Les éléments des tableaux considérés sont distincts ;
2. La probabilité pour que x soit dans t vaut $q \in [0, 1]$;
3. Si l'élément x est dans t, il peut se trouver à chaque endroit avec la même probabilité.

Notations

- D_n : ensemble des tableaux de taille n ;
- $D_{n,i}$: ensemble des tableaux de taille n tels que $x = t_i$ ($i \in [0, n-1]$) ;
- $D_{n,n}$: ensemble des tableaux de taille n ne contenant pas x.

Les calculs de complexité moyenne dépendent fortement des hypothèses que l'on fait sur les données.

Hypothèses :

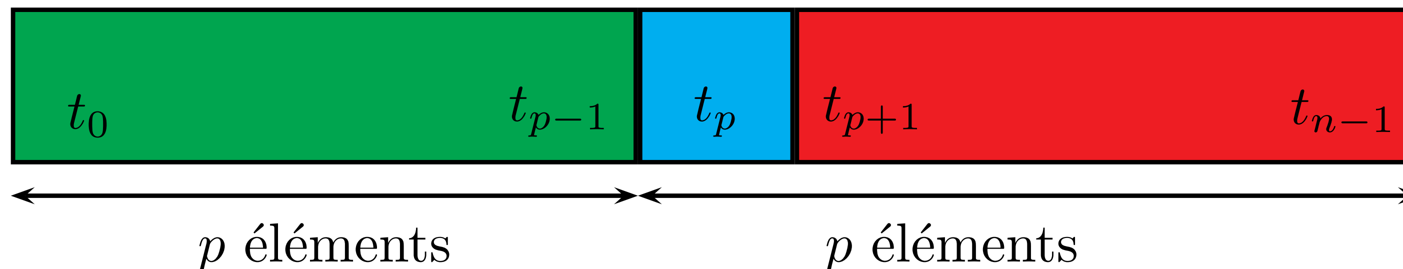
1. Les éléments du tableau et l'élément recherché x sont des entiers compris entre 1 et p ;
2. Les éléments peuvent apparaître éventuellement plusieurs fois dans le tableau.

On trouve sous ces hypothèses que

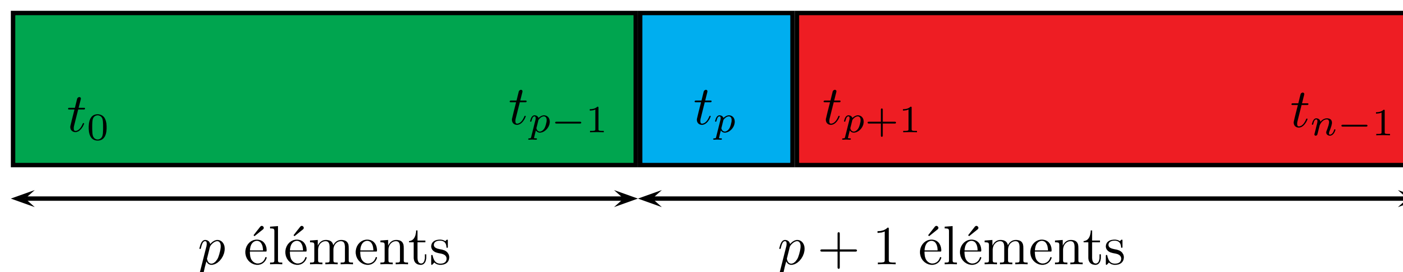
$$C_{moy}(n) = p - \frac{(p-1)^n}{p^{n-1}}$$

Partage d'un tableau

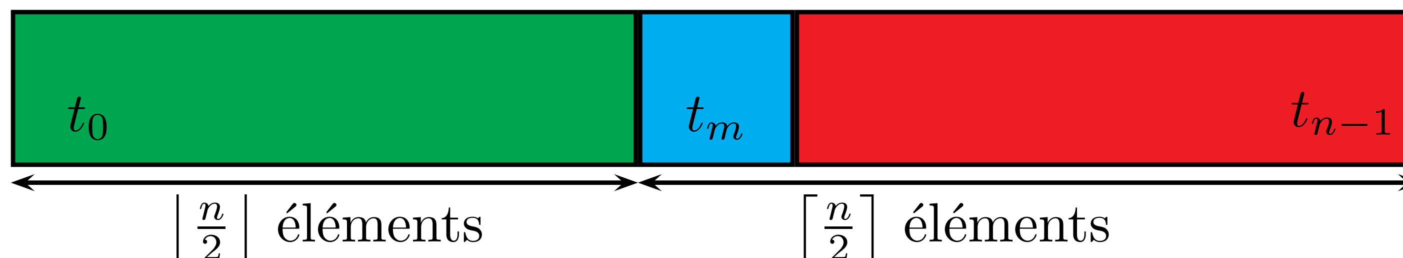
- $n = 2p$: $m = n/2 = p$, $\frac{n}{2} = p$, $\left\lfloor \frac{n}{2} \right\rfloor = p$, $\left\lceil \frac{n}{2} \right\rceil = p$.



- $n = 2p + 1$, $m = n/2 = p$, $\frac{n}{2} = p + \frac{1}{2}$, $\left\lfloor \frac{n}{2} \right\rfloor = p$, $\left\lceil \frac{n}{2} \right\rceil = p + 1$.



- Dans tous les cas, $m = n/2 = \left\lfloor \frac{n}{2} \right\rfloor$,

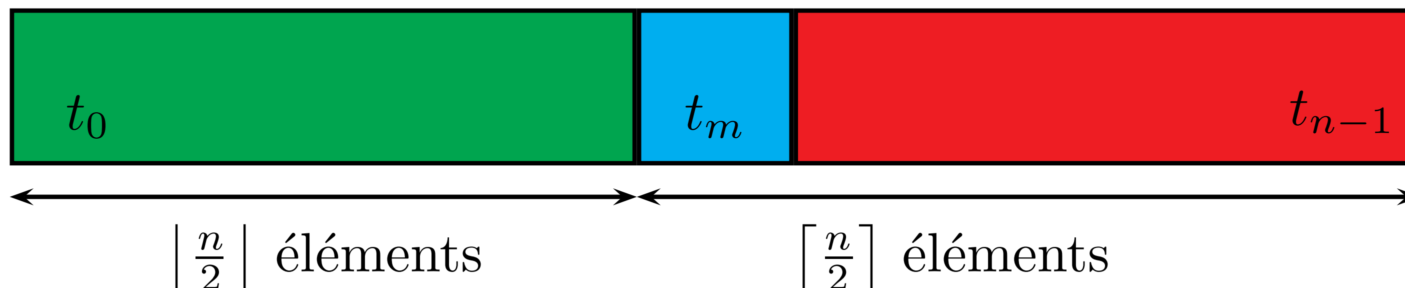


Recherche dichotomique

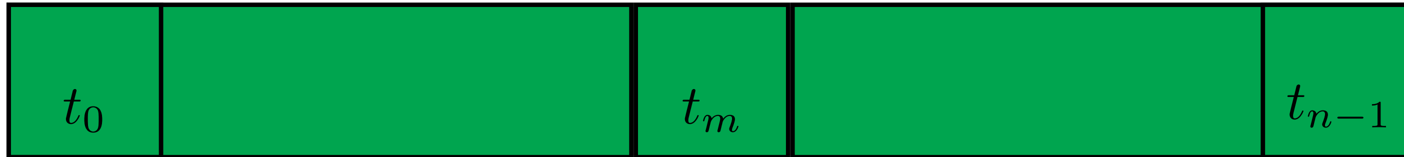
On suppose le tableau t *trié par ordre croissant* :

$$t.(0) \leq t.(1) \leq \cdots \leq t.(n-1)$$

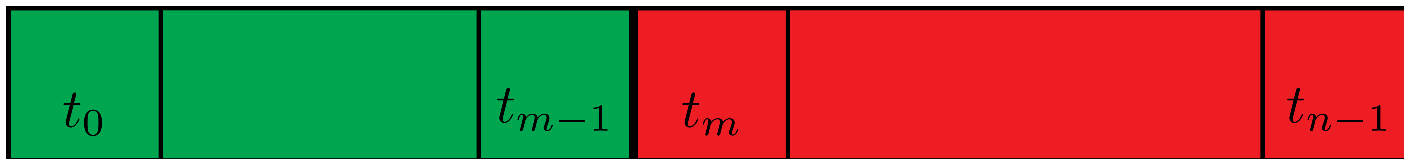
Si le tableau est de taille supérieure à 1, on compare x avec l'élément `t.(m)` situé au milieu du tableau : $m = n/2 = \left\lfloor \frac{n}{2} \right\rfloor$.



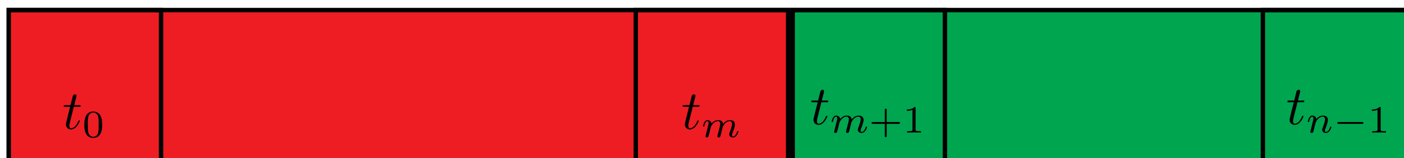
$x = t_m$



$x < t_m$



$x > t_m$



```
let recherche_dicho x t =
  let rec cherche_entre g d =
    if d < g  then false
    else
      let m = (g + d) / 2 in
        if x = t.(m) then true
        else
          if x < t.(m) then
            cherche_entre g (m - 1)
          else
            cherche_entre (m + 1) d
  in
    cherche_entre 0 (vect_length t - 1);;
```

- Preuve de la terminaison et de la correction de `recherche_dicho`.
- Le nombre de comparaisons lors d'une recherche dichotomique est en $\Theta(\log n)$ dans le pire des cas.

`Exercice 2.` Écrire une version impérative de `recherche_dicho`. Prouvez votre fonction grâce à un invariant de boucle.

`Exercice 3.` On veut trouver l'indice de la *première* occurrence de x s'il est présent dans le tableau. Comment modifier la recherche dichotomique pour cela ? Prouver la terminaison de votre fonction, et déterminer le nombre maximal de comparaisons.

Tris élémentaires

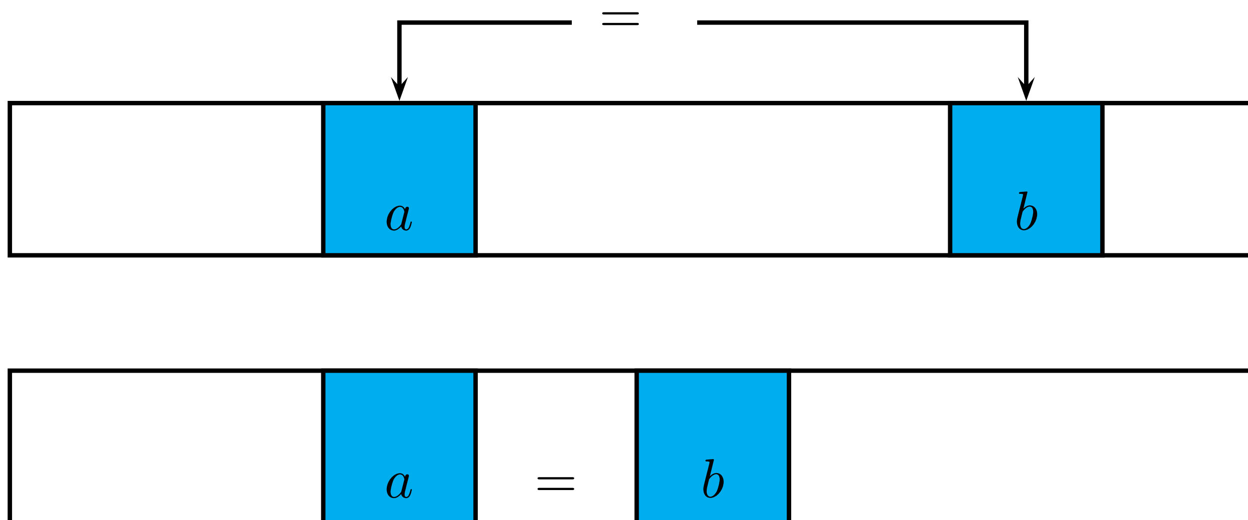
Clés à trier

- Exemple de donnée : `{nom, prénom, age, note math}` ;
- Clé : note math ;
- On dispose d'un *ordre* sur les clés ;
- But : trier sur place un tableau d'enregistrements par ordre croissant sur les clés.

Complexité d'un tri :

- nombre de *comparaisons* $C(n)$ entre clés ;
- nombre de *transferts* $T(n)$ d'enregistrements.

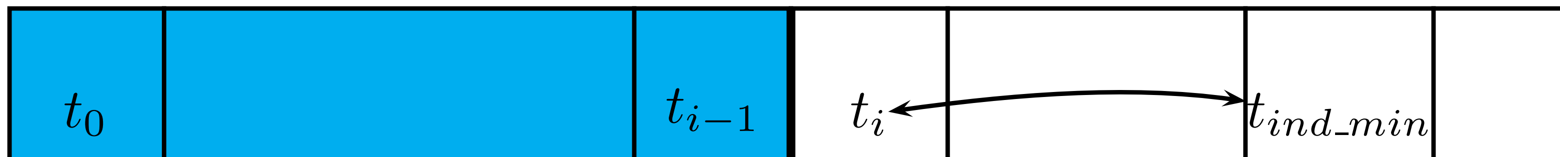
Stabilité d'un tri



Tri par sélection

Idée du tri par sélection

On cherche le plus petit élément de `t.(0)` à `t.(n-1)` que l'on échange avec `t.(0)`, puis le plus petit élément de `t.(1)` à `t.(n-1)` que l'on échange avec `t.(1)` et ainsi de suite.



```
let tri_selection t =
  let echange  i j = ... in
  let n = vect_length t in
  let ind_min = ref 0 in
    for i = 0 to n - 2 do
      ind_min := i;
      for j = i + 1  to n - 1 do
        if t.(j) < t.(!ind_min) then
            ind_min := j;
 (* INV t.(ind_min) = min (t.(i), ... , t.(j)) *)
      done;
(* t.(ind_min) = min( t.(i) ... t.(n-1) ) *)
      echange  i !ind_min;
(* INV i < k < n ==> t.(0) <= ... <= t.(i) <= t.(k) *)
    done;;
```

- Le nombre de comparaisons du tri par sélection pour un tableau de taille n vaut :

$$C_{min}(n) = C_{moy}(n) = C_{max}(n) = n(n-1)/2 = \Theta(n^2)$$

- Le nombre de transferts du tri par sélection vaut :

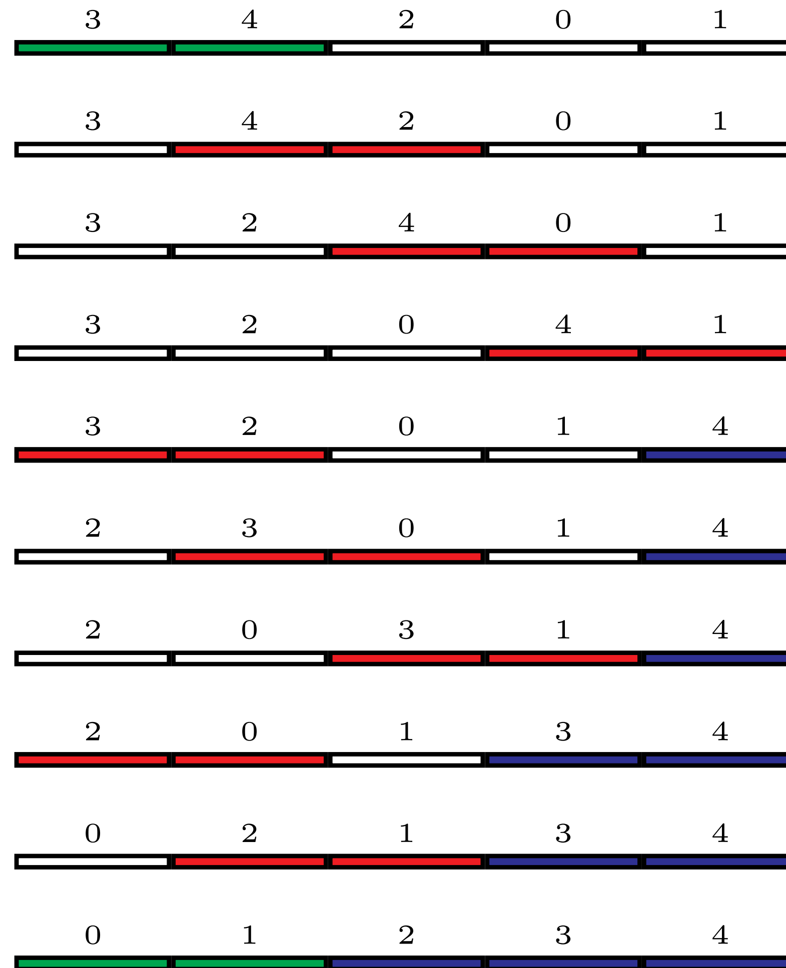
$$T_{min}(n) = T_{moy}(n) = T_{max}(n) = 3(n-1) = \Theta(n)$$

- Le tri par sélection n'est pas stable :

$$[|1_1; 1_2; 0|] \mapsto [|0; 1_2; 1_1|]$$

Tri bulle

Idée du tri bulle



```
let tri_bulle t =
  let echange i j = ... in
  let n = vect_length t  in
    for k = n - 2 downto 0 do
      for i = 0 to k do
        if t.(i) > t.(i + 1) then
          echange i (i + 1)
      done;
    done;;
```

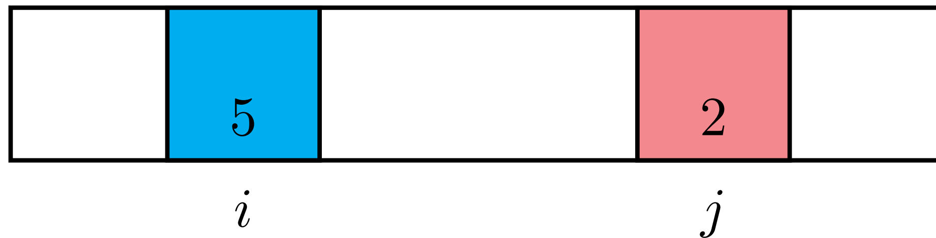
Permutation associée à un tableau

10	15	5	41	20	2
1	2	3	4	5	6

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 3 & 1 & 2 & 5 & 4 \end{pmatrix}$$

Inversions d'une permutation

Un couple (i, j) avec $i < j$ est une inversion de la permutation σ lorsque $\sigma(j) < \sigma(i)$.



Si $t = [|1; 5; 2; 4|]$, quelles sont les inversions ?

- Image miroir d'une permutation :

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 2 & 3 & 4 \end{pmatrix} \mapsto \tilde{\sigma} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 3 & 2 & 1 & 5 \end{pmatrix}$$

- (i, j) est une inversion de $\sigma \iff (n+1-j, n+1-i)$ n'est pas une inversion de σ' ;
- Nombre d'inversions dans σ et σ' : $\binom{n}{2}$;
- Nombre moyens d'inversions dans une permutation $\sigma \in S_n$:

$$\boxed{\frac{n(n-1)}{4}}$$

Analyse du tri bulle

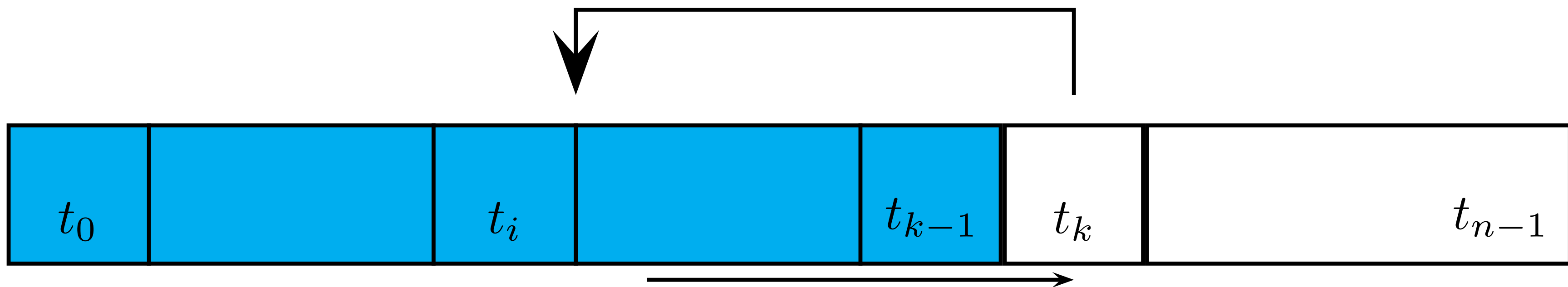
- Dans tous les cas, le nombre de comparaisons du tri bulle vaut :

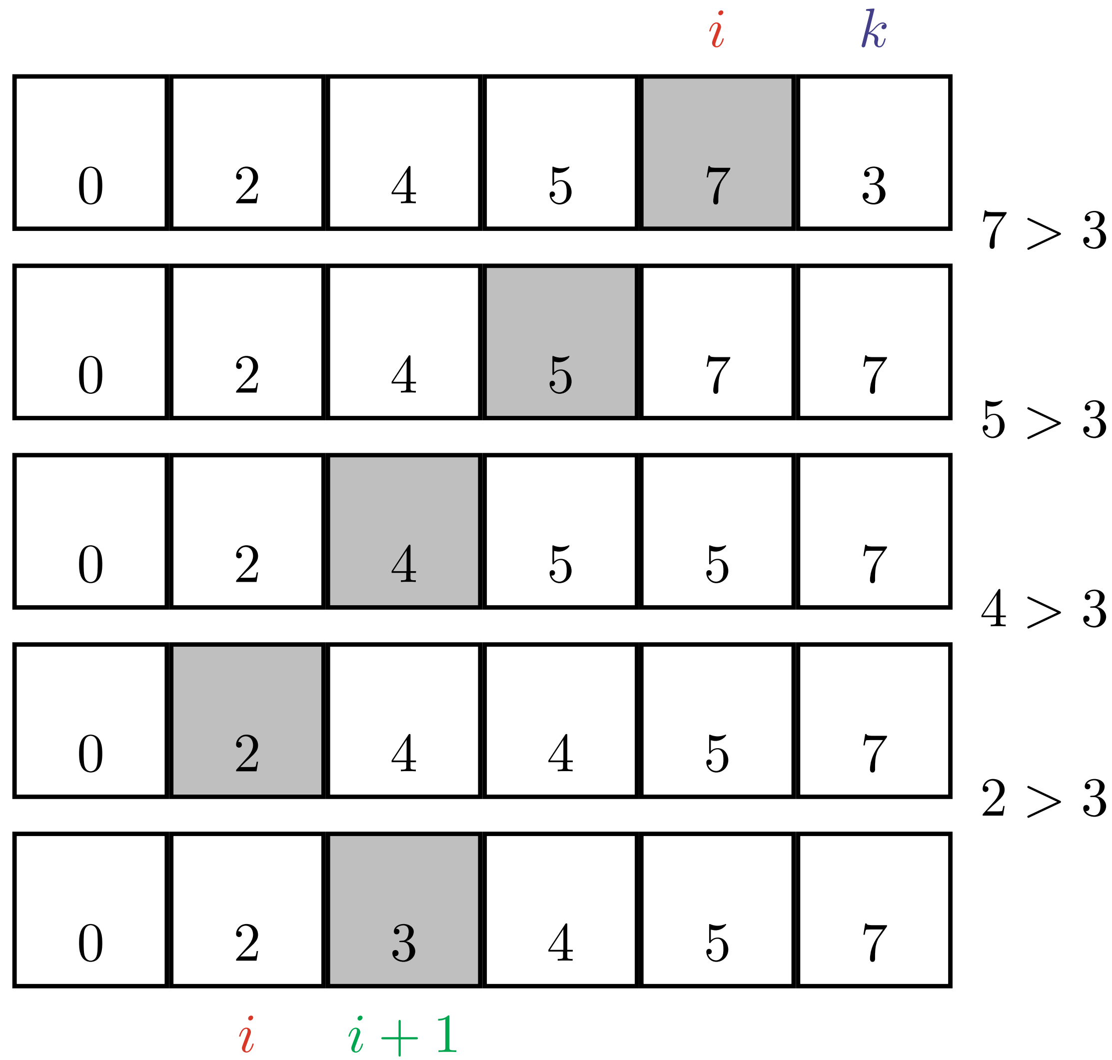
$$C_{min}(n) = C_{moy}(n) = C_{max}(n) = n(n-1)/2 = \Theta(n^2)$$

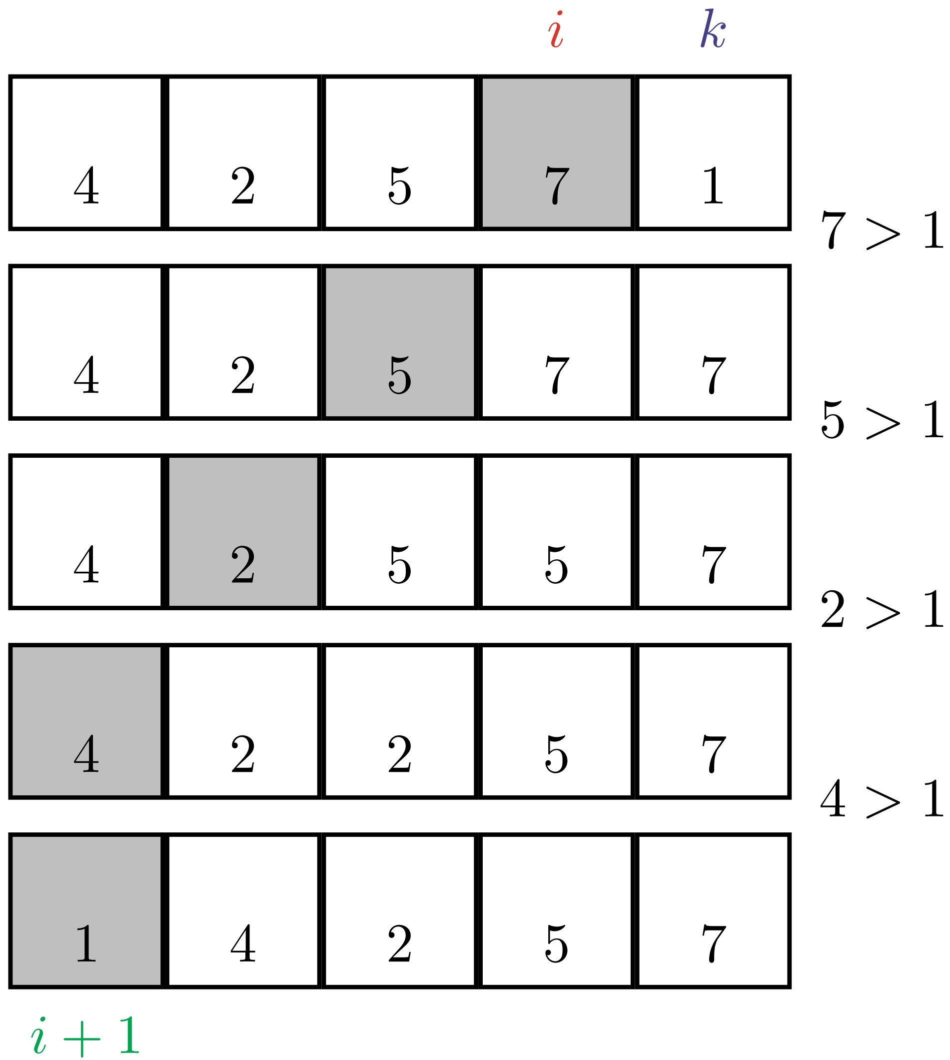
- Le nombre de transferts du tri bulle vaut :
  - $C_{min}(n) = 0$ dans le meilleur cas (le tableau est trié),
  - $C_{moy}(n) = 3n(n-1)/4 = \Theta(n^2)$ dans le cas moyen,
  - $C_{max}(n) = 3n(n-1)/2 = \Theta(n^2)$ dans le cas le pire ;
- Le tri bulle est stable.

Tri par insertion

Idée du tri par insertion




i
k
0
2
4
5
7
3
7 > 3
0
2
4
5
7
7
5 > 3
0
2
4
5
5
7
4 > 3
0
2
4
4
5
7
2 > 3
0
2
3
4
5
7
i
i + 1



i
k
4 2 5 7 1
$7 > 1$
4 2 5 7 7
$5 > 1$
4 2 5 5 7
$2 > 1$
4 2 2 5 7
$4 > 1$
1 4 2 5 7
i
$i+1$

```
let tri_insertion t =
  let n = vect_length t
  and i = ref(1)
  and temp = ref t.(0) in
    for k = 1 to n - 1 do
      temp := t.(k);
      i := k - 1;
      while (!i >= 0) && (t.(!i) > !temp) do
        t.(!i + 1) <- t.(!i);
        i := !i - 1;
      done;
      t.(!i + 1) <- !temp;
    done;;
```

Utilisation d'une sentinelle pour trier $[|t_1; \ldots; t_{n-1}|]$

sentinelle $= t_0 < t_1 \ldots t_{n-1}$

```
let tri_insertion_sentinelle t =
  let n = vect_length t and
    i = ref 1 and temp = ref 0 in
    for k = 2 to n - 1 do
      temp := t.(k);
      i := k - 1;
      while t.(!i) > !temp do
        t.(!i + 1) <- t.(!i);
        i := !i - 1;
      done;
      t.(!i + 1) <- !temp;
    done;;
```

0	2	4	5	7	3	$7 > 3$ $temp \leftarrow 3$
0	2	4	5	7	7	$5 > 3$ $t_5 \leftarrow 7$
0	2	4	5	5	7	$4 > 3$ $t_4 \leftarrow 5$
0	2	4	4	5	7	$2 > 3$ $t_3 \leftarrow 4$
0	2	3	4	5	7	$t_2 \leftarrow temp$

4 comparaisons et 5 transferts.

Analyse du tri par insertion

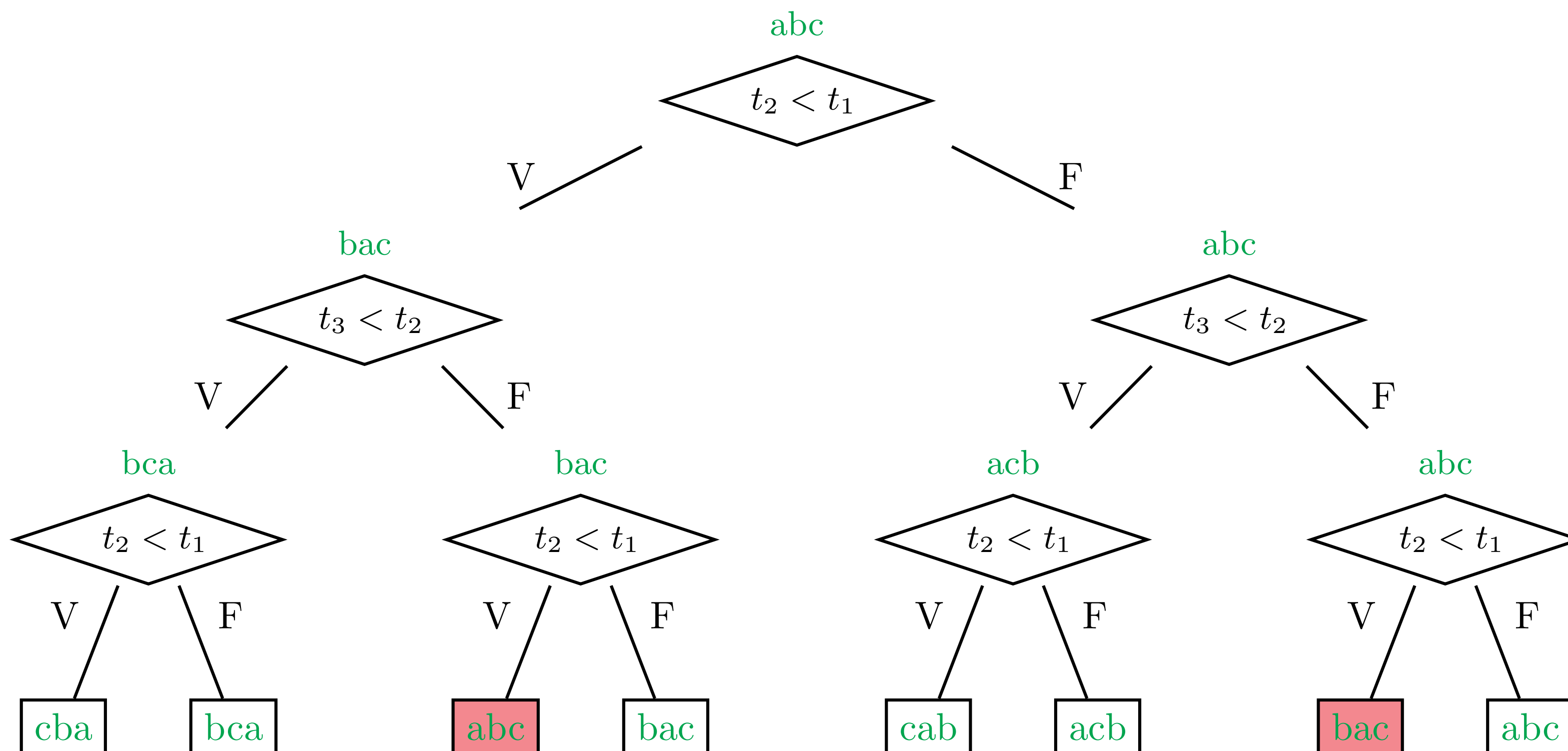
- Le nombre de comparaisons du tri par insertion vaut :
  - Dans le meilleur cas, $C_{min}(n) = n - 1 = \Theta(n)$,
  - Dans le cas moyen, $C_{moy}(n) = (n-1)(n+4)/4 = \Theta(n^2)$,
  - Dans le cas le pire, $C_{max}(n) = (n-1)(n+2)/2 = \Theta(n^2)$;
- Le nombre de transferts du tri par insertion vaut :
  - Dans le meilleur des cas, $T_{min}(n) = 2(n-1) = \Theta(n)$,
  - Dans le cas moyen, $T_{moy}(n) = (n-1)(n+8)/4 = \Theta(n^2)$,
  - Dans le cas le pire, $T_{max}(n) = (n-1)(n+4)/2 = \Theta(n^2)$;
- Le tri par insertion est stable.

Comparaison des tris élémentaires

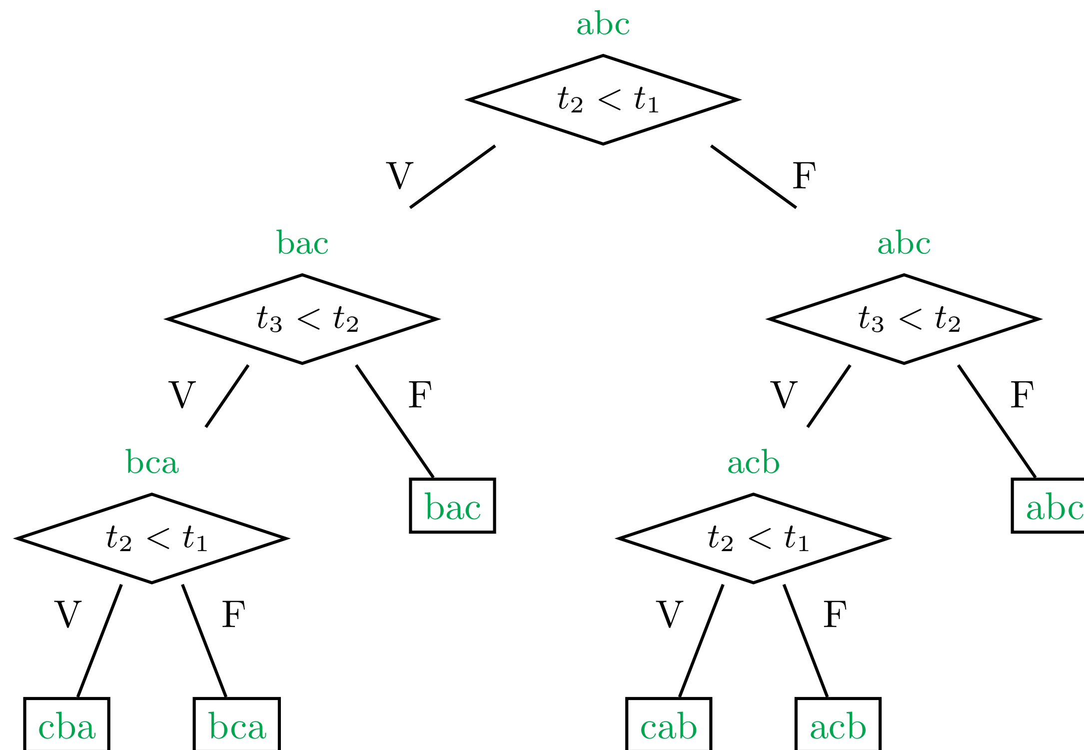
	Comparaisons			Transferts			Stabilité
Tri	min	moy	max	min	moy	max	
Sélection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	non
Bulle	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	0	$\Theta(n^2)$	$\Theta(n^2)$	oui
Insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	oui

Une borne inférieure pour les algorithmes de tris par comparaisons

Arbre de décision pour le tri bulle $t = [|a;\ b;\ c|]$



Arbre de décision pour un algorithme optimal qui trie 3 éléments



Théorème 6. ***Borne inférieure d'un algorithme de tri***

Tout algorithme de tri qui n'utilise que les comparaisons entre éléments d'un tableau de taille n effectue dans le cas le pire au moins $\log_2(n!)$ *comparaisons.*

Comme $log_2(n!) \sim n \log_2 n$, le nombre de comparaisons d'un tel algorithme de tri dans le cas le pire est en $\Omega(n. \log n)$.

- Il n'existe pas d'algorithme de tri linéaire qui n'utilise que les comparaisons entre clés ;
- Le résultat précédent ne vaut que pour les tris qui se basent sur les comparaisons des clés. Si l'on fait des hypothèses sur les clés, il existe des algorithmes linéaires ;
- Aucun algorithme de tri élémentaire n'a cette complexité asymptotique optimale ;
- Nous verrons deux algorithmes de tris qui ont une complexité en $O(n \log n)$: ils seront en un certain sens “optimaux”.

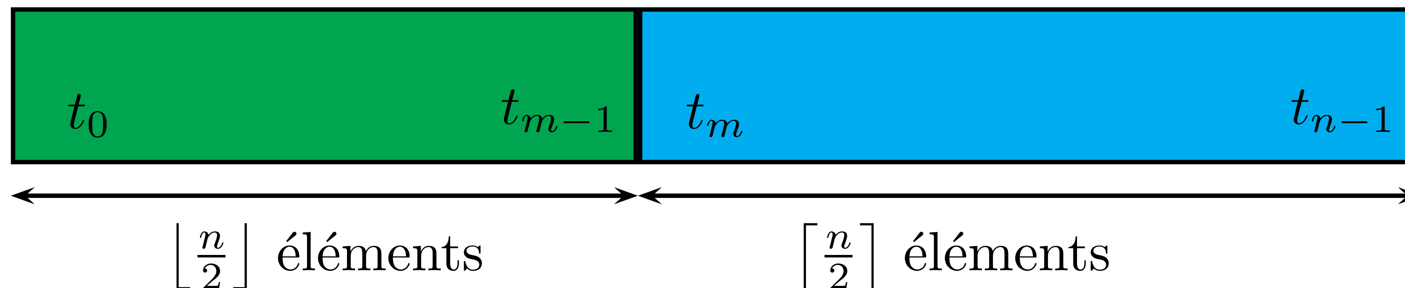
Chapitre 4

Diviser pour régner

Idée de la méthode

Pour traiter un problème de taille n :

1. Partager les données en deux parties de taille approximativement égales ($m = n/2 = \lfloor \frac{n}{2} \rfloor$) ;



2. Traiter récursivement les deux parties ;

3. Fusionner les résultats des deux parties.

Un exemple modèle

1. Partager le problème de taille n en deux sous-problèmes de taille $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$. Coût du partage : $C_1 n$;
2. Traiter récursivement les deux sous-problèmes ;
3. Fusionner les résultats. Coût de la fusion : $C_2 n$.

a) Ecrire la relation de récurrence vérifiée par le coût $T(n)$ de l'algorithme.

b) On suppose que $n = 2^p$ est une puissance de 2. En notant $u_p = T(2^p)$, déterminer la relation de récurrence vérifiée par la suite (u_p).

c) En déduire l'expression de $T(n)$ en fonction de n lorsque n est une puissance de 2.

d) Montrer que $T(n)$ est une suite croissante.

e) Montrer que $T(n) = \Theta(n \log n)$.

Complexité générale des récurrences diviser pour régner

1. *Diviser* le problème en deux sous-problèmes de taille $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$. Coût du partage : $f_1(n)$;
2. *Résoudre récursivement* les deux sous-problèmes ;
3. *Fusionner* les résultats. Coût de la fusion : $f_2(n)$.

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + f_1(n) + f_2(n)$$

Coût total de la division-fusion : $f(n) = f_1(n) + f_2(n)$.

- Si $f(n) = O(n^\beta)$ avec $\beta < 1$, alors $T(n) = \Theta(n)$;
- Si $f(n) = \Theta(n)$, alors $T(n) = \Theta(n \log n)$;
- Si $f(n) = \Theta(n^\beta)$ avec $\beta > 1$, alors $T(n) = \Theta(n^\beta)$.

Théorème 7. ***Complexité générale d'un algorithme diviser pour régner***

Soit $(a, b) \in \mathbb{N}^2$ *avec* $a + b \geq 1$. *Posons*

$$\alpha = \log_2(a + b)$$

On considère une suite $T(n)$ *vérifiant la relation de récurrence :*

$$T(n) = aT\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + bT\left(\left\lceil \frac{n}{2} \right\rceil\right) + f(n)$$

- *Si* $f(n) = O(n^\beta)$ *avec* $\beta < \alpha$, *alors* $T(n) = \Theta(n^\alpha)$ *;*
- *Si* $f(n) = \Theta(n^\alpha)$ *alors* $T(n) = \Theta(n^\alpha \log n)$ *;*
- *Si* $f(n) = \Theta(n^\beta)$ *avec* $\beta > \alpha$ *alors* $T(n) = \Theta(n^\beta)$.

Exponentiation rapide

But : calculer x^n ($x \in \mathbb{R}$, $n \in \mathbb{N}$).

Algorithme naïf

```
let exponentiation x n =
  let p = ref(1.) in
    for i = 1 to n do
      p := !p *. x
    done;
    !p;;
```

Il nécessite n multiplications.

Exponentiation rapide

$$x^n = \begin{cases} \left(x^{n/2}\right) * \left(x^{n/2}\right) & \text{si n est pair} \\ x * \left(x^{n/2}\right) * \left(x^{n/2}\right) & \text{si n est impair} \end{cases}$$

```
let rec expo_rapide x n =
  match n with
    | 0 -> 1.;
    | 1 -> x;
    | n when n mod 2 = 0 ->
        let y = expo_rapide x (n / 2) in y *. y
    | n ->
        let y = expo_rapide x (n / 2) in x *. y *. y;;
```

Théorème 8. ***Complexité de l'exponentiation rapide***

1. *Lorsque* $n = 2^p$*, l'algorithme d'exponentiation rapide nécessite* $\log_2(n)$ *multiplications.*

2. *Lorsque* n *est quelconque, l'algorithme d'exponentiation rapide nécessite* $\Theta(\log_2 n)$ *multiplications dans le pire des cas.*

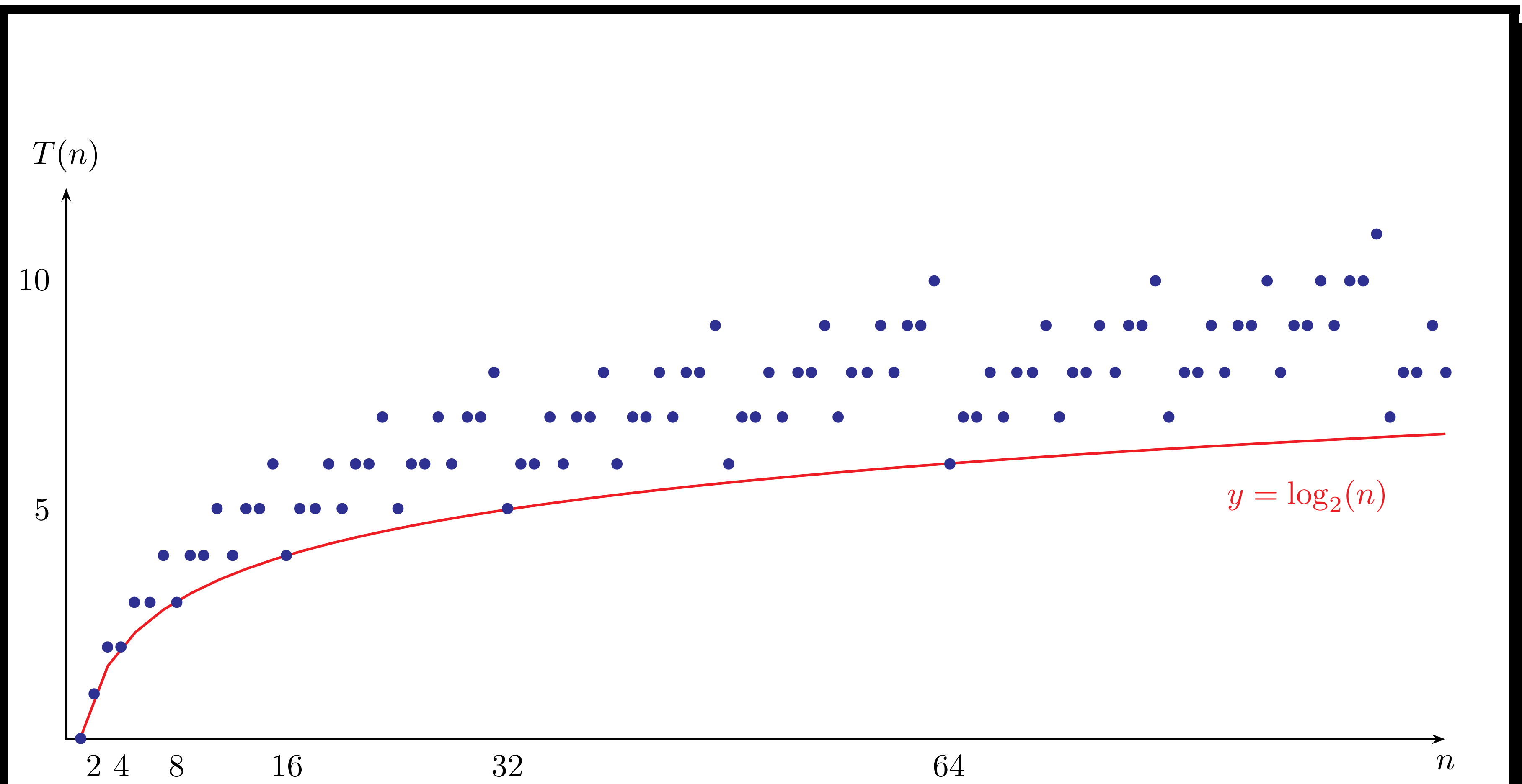


Figure 5: Nombre de multiplications pour l'exponentiation rapide

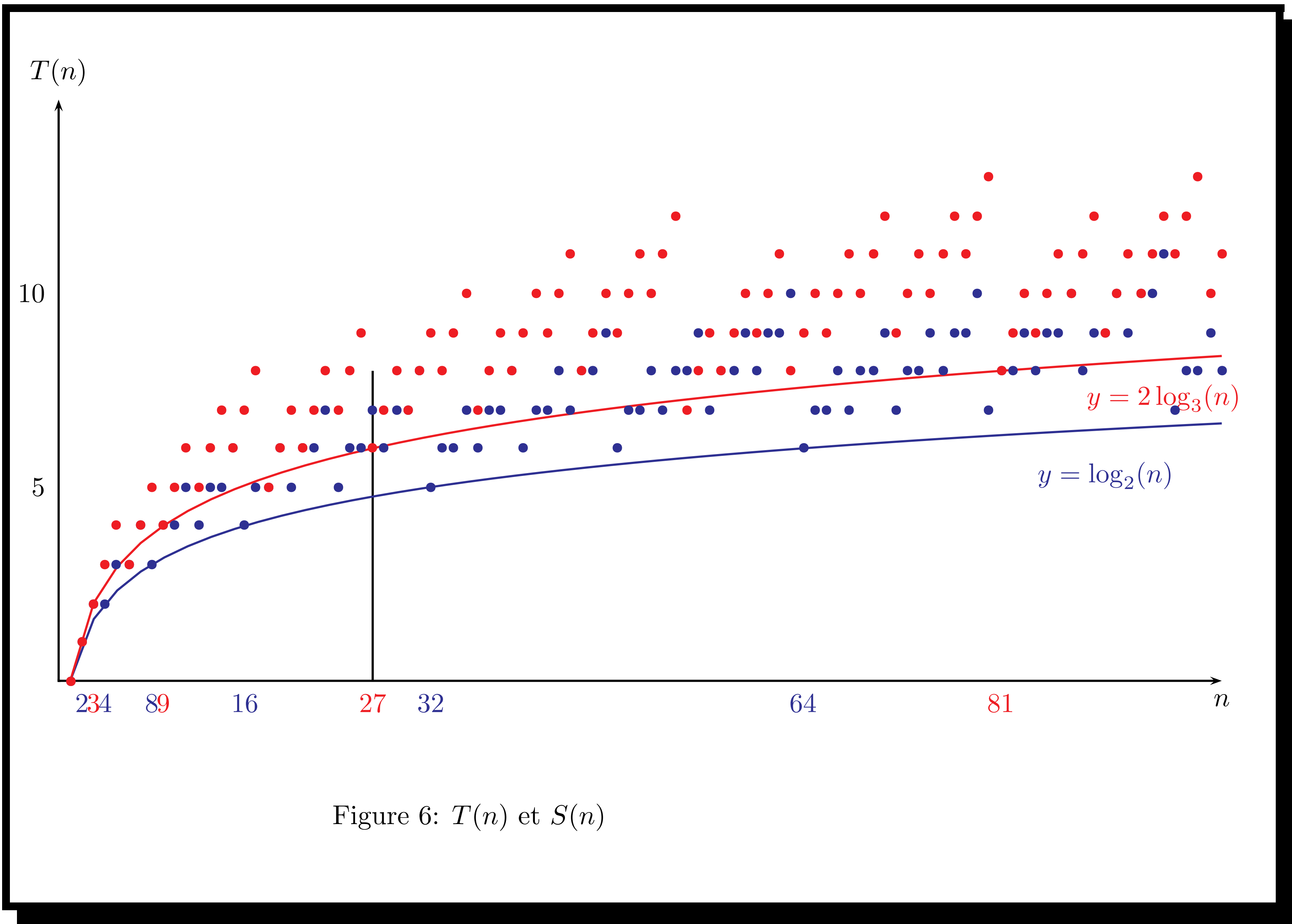
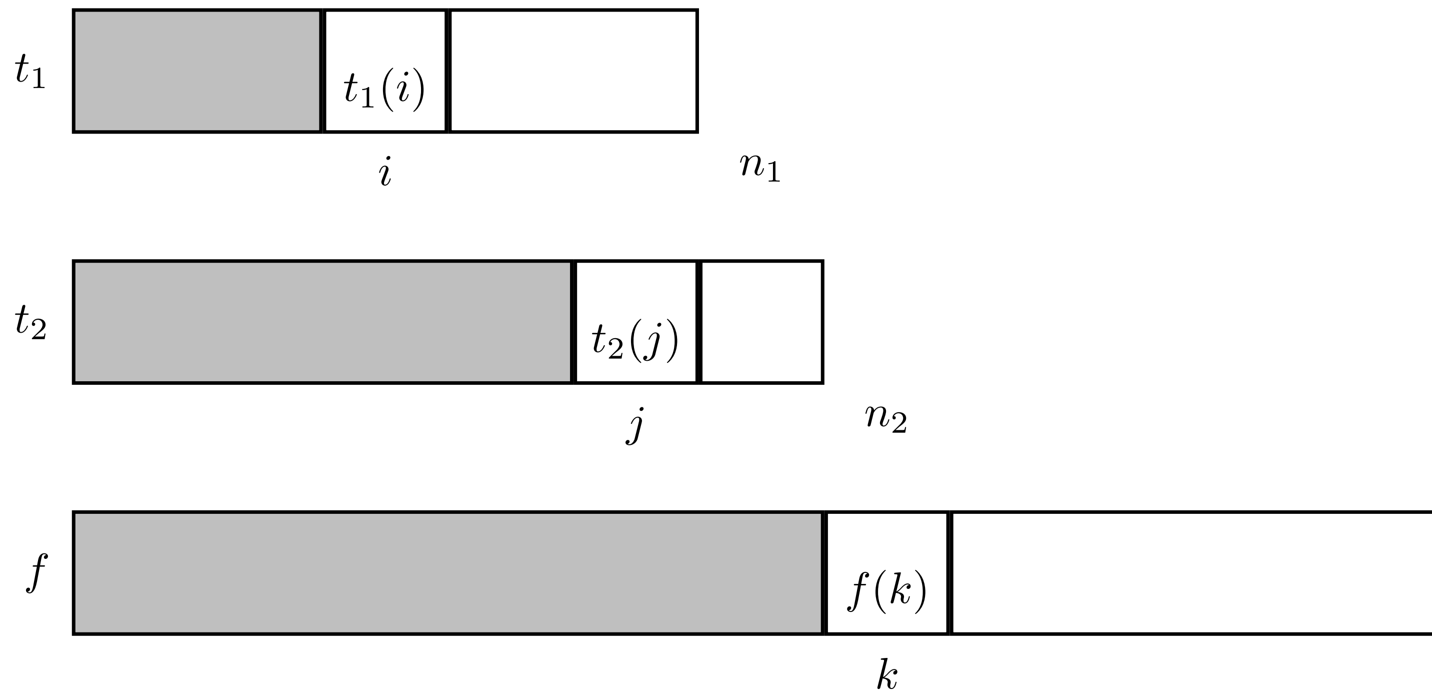


Figure 6: $T(n)$ et $S(n)$

Tri fusion (mergesort)

Fusion linéaire de deux tableaux triés



Tri fusion

1. *diviser* le tableau t en deux tableaux t_1, t_2 de tailles $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$
2. *trier récursivement* les deux tableaux t_1 et t_2.
3. *fusionner* les deux tableaux triés.

```
let tri_fusion t =
  let n = vect_length t in
  let fusion debut milieu fin = ...
  in
    (* tri du sous-tableau [|t.(i) ... t.(j) |] *)
  let rec tri_rec i j =
    if j > i  then  (* au moins 2 éléments *)
      let m = (i + j) /2 in
        tri_rec i m;
        tri_rec (m + 1)  j;
        fusion i m j
  in
    tri_rec 0 (n-1);;
```

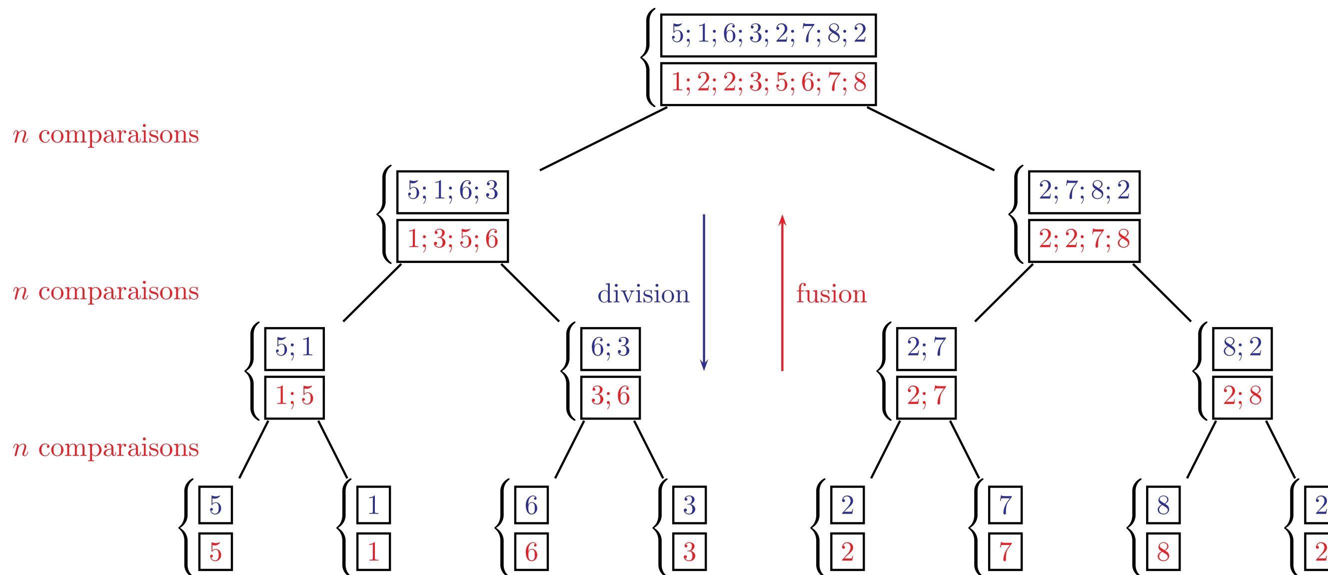


Figure 7: Tri fusion. Hauteur de l'arbre : $\log_2(n)$, nombre de comparaisons : $n \log_2(n)$

Théorème 9. ***Complexité du tri fusion***

1. *Si le tableau t est de taille $n = 2^p$, le tri fusion nécessite $n \log_2 n$ comparaisons.*
2. *Si le tableau t est de taille n quelconque, le tri fusion nécessite $\Theta(n \log_2 n)$ comparaisons.*

L'inconvénient majeur du tri fusion est qu'il nécessite un tableau auxiliaire de taille n dans l'étape de fusion ; il n'est pas utilisé en pratique.

Tri rapide (quicksort)

Idée du tri rapide

- Choisir un *pivot* (ici le premier élément) ;

4	1	5	8	2	0	3	7

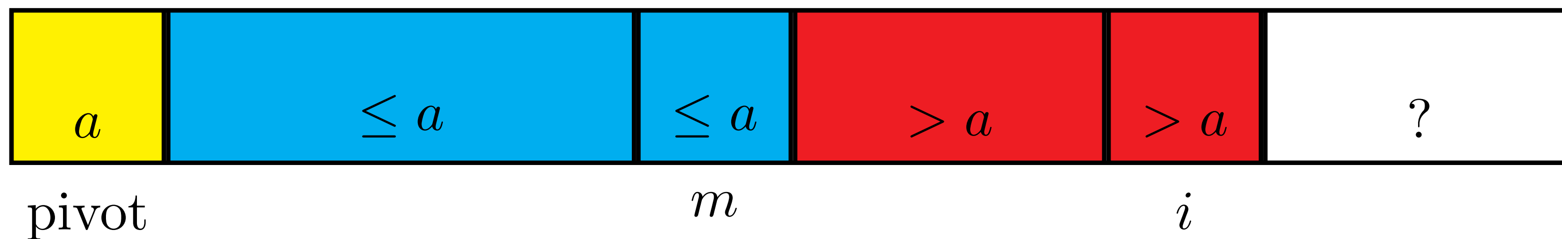
- Partager le tableau en deux selon le pivot ;

1	2	0	3	4	5	8	7

- Trier récursivement les sous-tableaux gauche et droit.

0	1	2	3	4	5	7	8

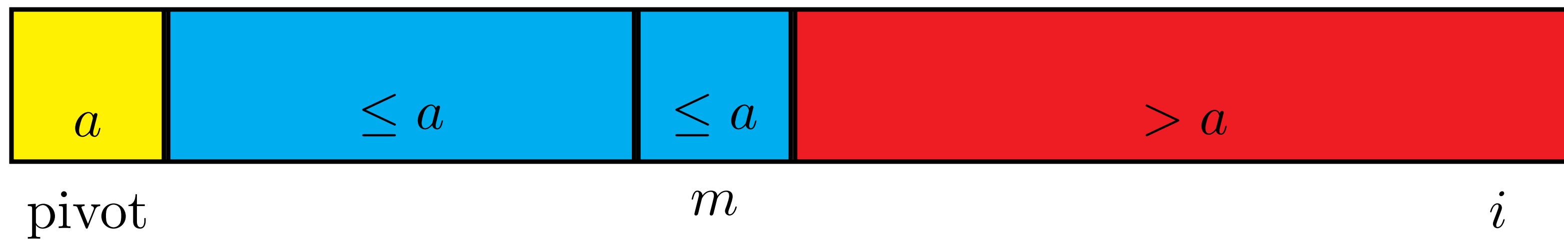
Algorithme de partition



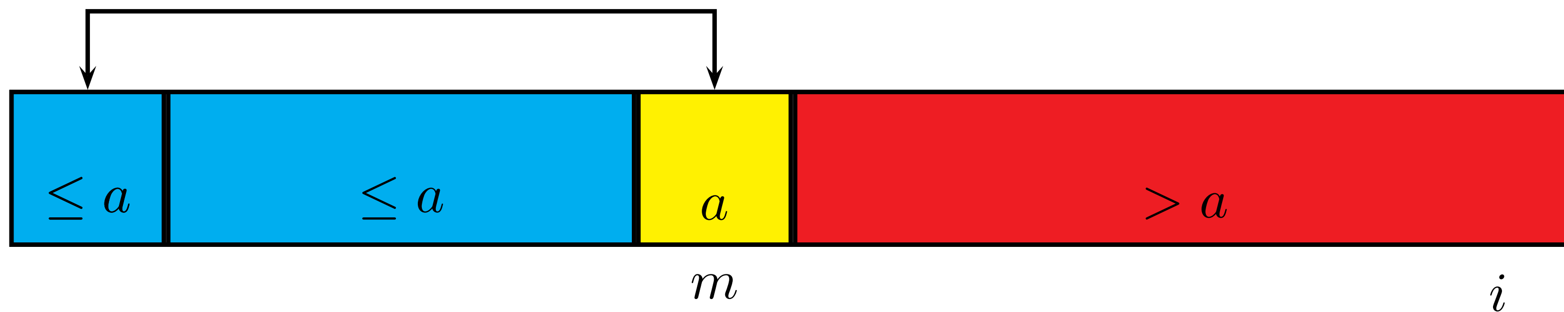
Invariant :

- $1 \leq m \leq i < n$;
- $\forall k \in [1, m],\ t_k \leq a$;
- $\forall k \in [m+1, i],\ t_k > a$.

A la sortie de boucle



Echanger t_m avec t_0



Analyse du tri rapide

Théorème 10. *Le tri rapide nécessite :*

- *Dans le pire des cas :* $\Theta(n^2)$ *comparaisons ;*
- *En moyenne :* $\Theta(n \log n)$ *comparaisons.*
- Intérêt du tri rapide :
  - pas de tableau auxiliaire,
  - bonne complexité en moyenne ;
- Inconvénients :
  - Mauvaise complexité dans le cas le pire —lorsque le tableau est déjà trié !

Multiplication de Knuth des polynômes

$$P = a_0 + a_1 X + \cdots + a_n X^p, \quad Q = b_0 + b_1 X + \cdots + b_q X^q$$

$$R = PQ = c_0 + c_1 X + \cdots + c_{p+q} X^{p+q}$$

```
let multiplie p q =
  let dp = vect_length p - 1 (* degré de p*)
  and dq = vect_length q - 1 in (* degré de q*)
  let r = make_vect (dp + dq + 1) 0 in
    (* remplit les coefficients du produit*)
    for i = 0 to dp do
      for j = 0 to dq do
        r.(i + j) <- r.(i + j) + p.(i) * q.(j)
      done;
    done;
    r;;
```

Algorithme de Knuth

$$
\begin{aligned}
P &= \left(a_0 + a_1 X + \cdots + a_{n-1} X^{n-1}\right) \\
&\quad + X^n \left(a_n + a_{n+1} X + \cdots + a_{2n-1} X^{n-1}\right) \\
&= P_0 + X^n P_1
\end{aligned}
$$

$$
\begin{aligned}
Q &= \left(b_0 + b_1 X + \cdots + b_{n-1} X^{n-1}\right) \\
&\quad + X^n \left(b_n + b_{n+1} X + \cdots + b_{2n-1} X^{n-1}\right) \\
&= Q_0 + X^n Q_1
\end{aligned}
$$

Alors le produit s'écrit :

$$
\begin{aligned}
R &= (P_0 + P_1 X^n) * (Q_0 + Q_1 X^n) \\
&= P_0 * Q_0 + (P_0 * Q_1 + P_1 * Q_0)\, X^n + P_1 * Q_1 X^{2n} \\
&= P_0 * Q_0 + ((P_0 + P_1) * (Q_0 + Q_1) - P_0 * Q_0 - P_1 * Q_1)\, X^n \\
&+ P_1 * Q_1 X^{2n} \\
&= \alpha_0 + (\alpha_1 - \alpha_0 - \alpha_2)\, X^n + \alpha_2 X^{2n}
\end{aligned}
$$

$$
\alpha_0 = P_0 * Q_0, \quad \alpha_1 = (P_0 + P_1) * (Q_0 + Q_1), \quad \alpha_2 = P_1 * Q_1
$$

Analyse de l'algorithme de Knuth

Théorème 11. *On suppose que les deux polynômes P et Q sont de degré $2^k - 1$. (Les polynômes P et Q ont donc $n = 2^k$ coefficients). En notant $M(k)$ le nombre de multiplications scalaires, on a la formule de récurrence :*

$$M(0) = 1$$

$$\forall k \geq 1 \quad M(k) = 3M(k-1)$$

et on obtient alors

$$M(k) = 3^k$$

d'où le nombre total de multiplications qui vaut

$$n^{\log_2 3} \approx n^{1.58}$$

Remarque : il existe un algorithme qui permet de multiplier 2 polynômes en $\mathcal{O}(n \log n)$ multiplications (transformée de Fourier

rapide).

Multiplication de matrices : l'algorithme de Strassen

Multiplication de matrices

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix}$$

$C = ((c_{ij}))_{1 \leq i,j \leq n} = A \times B$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

Algorithme naïf : n^3 multiplications.

Idée de Strassen

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix}$$

et alors

$$C = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix}$$

L'idée de Strassen est qu'il suffit de 7 multiplications matricielles pour calculer C à la place de 8.

Complexité de l'algorithme de Strassen

`Exercice 4.` Déterminer le nombre de multiplications scalaires nécessaires dans l'algorithme de Strassen.

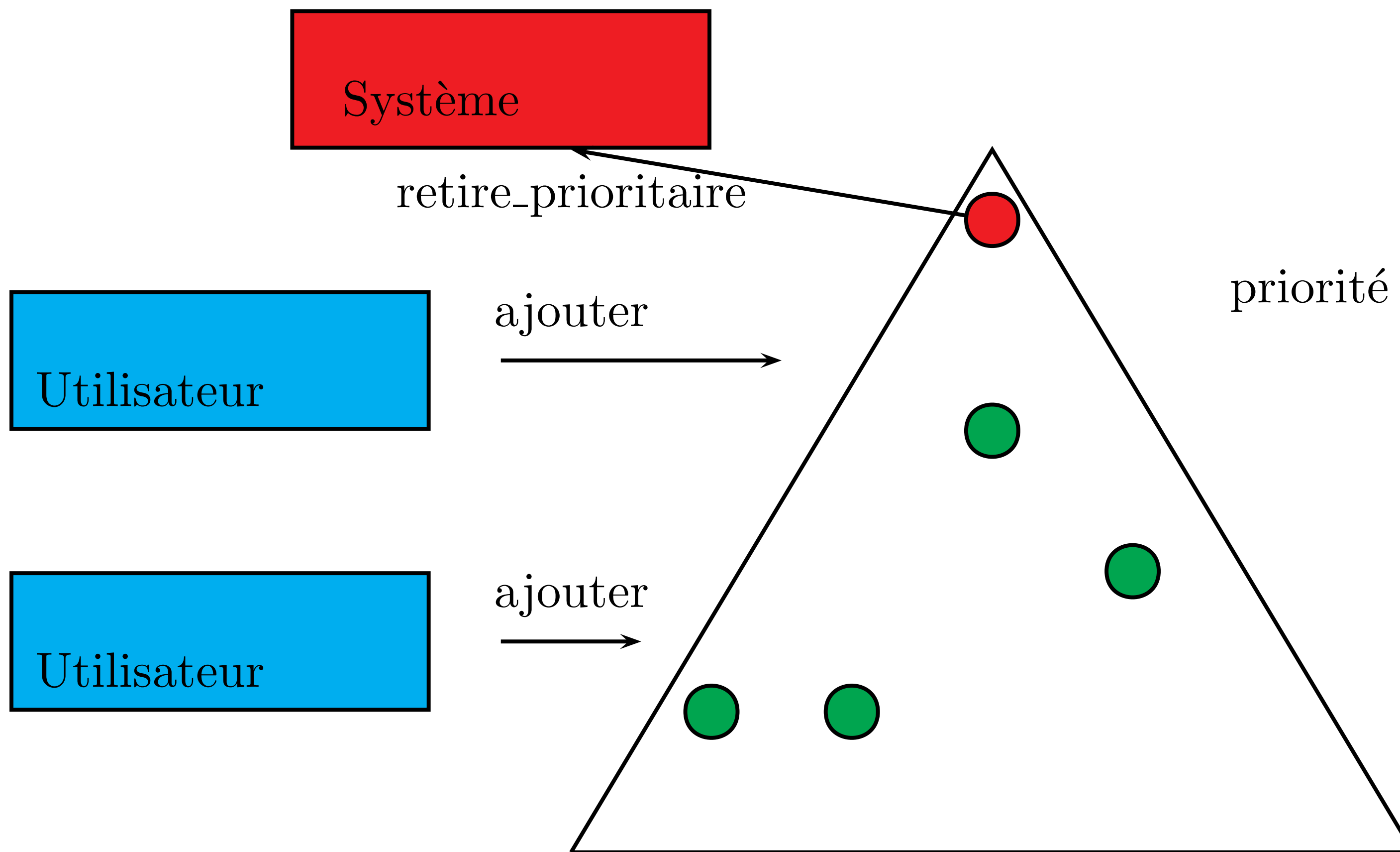
`Exercice 5.` Si dans l'algorithme précédent, on utilise 8 multiplications au lieu de 7, déterminer le nombre de multiplications nécessaires.

Chapitre 5

Structures de données abstraites, piles, expressions algébriques

Structures de données abstraites

File de priorité



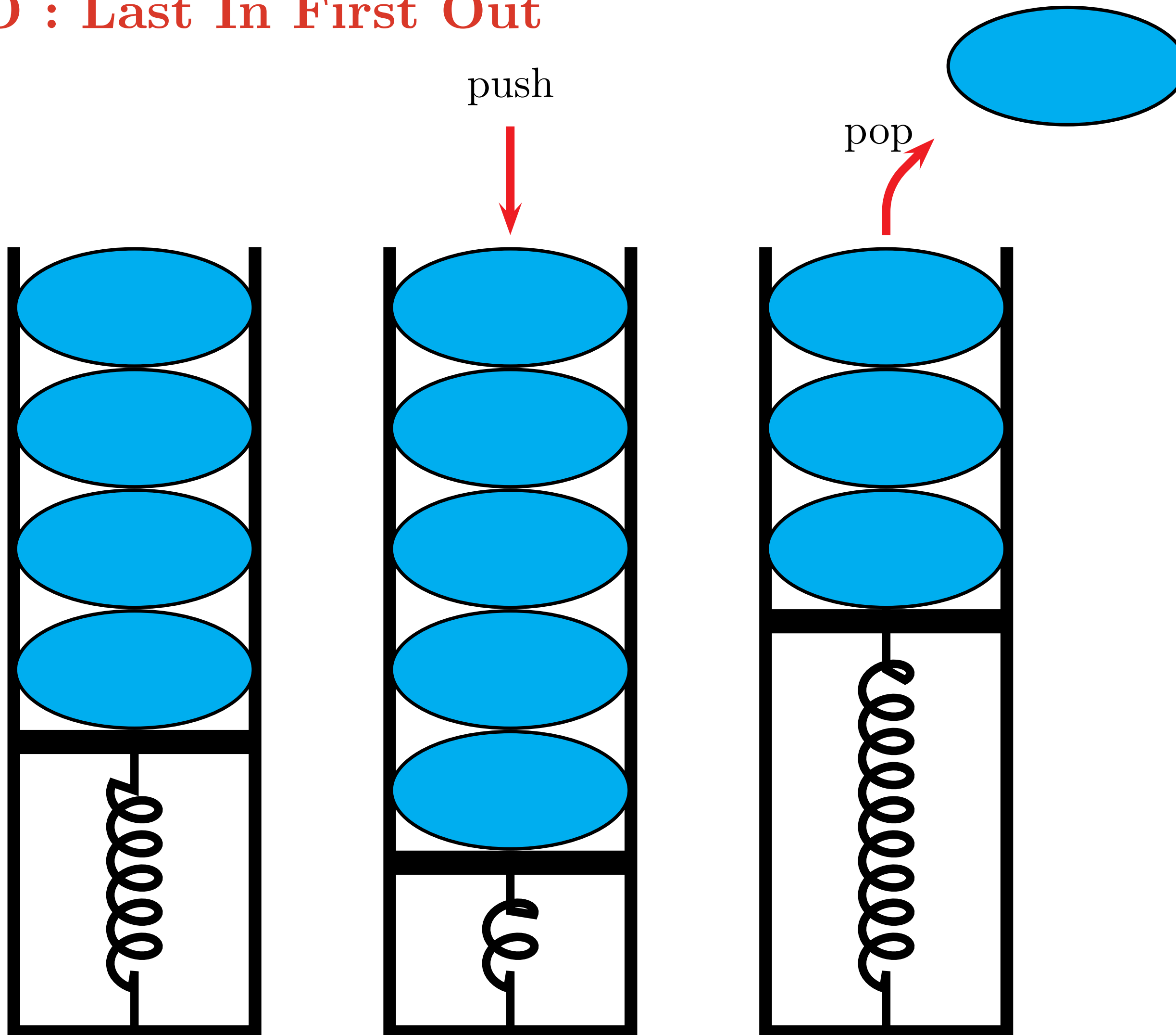
Opérations sur une file de priorité

- `new_file : unit -> 'a file` (nouvelle file vide) ;
- `ajouter : 'a -> 'a file -> unit` (ajout d'un élément à la file) ;
- `retire_prioritaire : 'a file -> 'a` (retire l'élément de plus haute priorité de la file et le retourne).

Spécifications formelles d'une structure de données.

Piles

Pile LIFO : Last In First Out



Opérations sur une pile

- `new : unit -> 'a pile` : crée une nouvelle pile vide ;
- `push : 'a -> 'a pile -> unit` : ajout d'un élément en sommet de pile ;
- `pop : 'a pile -> 'a` : retire l'élément en sommet de pile et le retourne ;
- `is_empty : 'a pile -> bool` : un prédicat disant si une pile est vide.

Exceptions :

- `pile vide` (lorsque `pop` est appelé sur une pile vide) ;
- `stack overflow` (dépassement de pile).

Exemples d'utilisation d'une pile

- Évaluation des fonctions récursives ; Lorsqu'une fonction s'appelle récursivement, son environnement est empilé, en attente du résultat. Une fois le résultat calculé, l'environnement est dépilé : pile d'exécution ;
- Vérification d'un parenthésage : voir TD.

Types produit en CAML

Représentation des points du plan

```
#type point = {x : float; y : float};;
Le type point est défini.

#let p1 = {x = 0.5; y = 4.1};;
p1 : point = {x = 0.5; y = 4.1}


#p1.x;;
- : float = 0.5
#p1.y;;
- : float = 4.1
```

```
#type vecteur = {dx : float; dy : float};;
Le type vecteur est défini.

#let translate p v  =
    {x = p.x +. v.dx; y = p.y +. v.dy};;
translate : point -> vecteur -> point = <fun>
```

Champs mutables

```
#type point = {mutable x : float; y : float};;

#let p = {x = 0.3; y = 2.};;

#p.x <- 2.;;
- : unit = ()

#p.y <- 5.;;
Entrée interactive:
>p.y <- 5.;;<EOF>
>^^^^^^^^^^
L'étiquette y n'est pas mutable.
```

Une implémentation des piles

```
type 'a pile = {mutable contenu : 'a list};;

let new () =  {contenu = []};;

let push x p =  p.contenu <- x :: p.contenu;;

let pop p =  match p.contenu with
    | [] -> failwith "pile vide"
    | x :: q -> p.contenu <- q; x;;

let is_empty p =  match p.contenu with
    | [] -> true
    | _ -> false;;
```

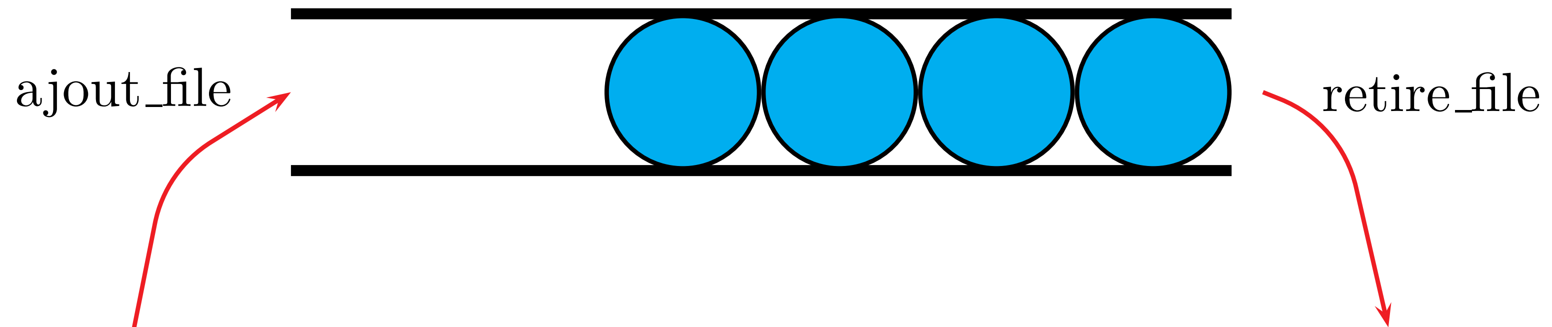
Exercice Ecrire une fonction

`rotation` de type : `'a pile -> unit`

qui effectue une “ rotation d'un cran ” de la pile : l'élément qui était au fond de la pile se retrouve au sommet et les autres éléments sont décalés vers le bas.

Le type abstrait File

File FIFO : First In First Out



Opérations sur les files

- `new_file : unit -> 'a file` : crée une nouvelle file vide ;
- `ajout_file : ‘a -> 'a file -> unit` : ajoute un élément à une file ;
- `retire_file : 'a file -> 'a` : retire le premier élément de la file et le retourne.
- `file_vide : 'a file -> bool` : un prédicat qui teste si une file est vide ;

Utilisation de files d’attente

Tampons d’entrée-sortie d’un système d’exploitation :

- Caractères frappés au clavier ;
- Écritures sur le disque.

Type somme en CAML

type énuméré

```
type figure = Valet | Dame | Roi | As;;

let valeur_figure f =
  match f with
    | Valet -> 2;
    | Dame -> 3;
    | Roi -> 4;
    | As -> 5;;

valeur_figure : figure -> int = <fun>
```

Constructeurs paramétrés

```
type nombre = Entier of int | Reel of float;;

let addition_nombres x y =
  match (x, y) with
    | Entier p, Entier q -> Entier (p + q)
    | Reel p, Reel q -> Reel (p +. q)
    | Entier p, Reel q -> Reel (float_of_int p +. q)
    | Reel p, Entier q -> Reel (p +. float_of_int q);;

addition_nombres : nombre -> nombre -> nombre = <fun>
```

Types somme récursifs

```
#type liste = NIL | Cons of int * liste;;

let rec longueur l =
  match l with
    | NIL -> 0
    | Cons (x, q) -> 1 + longueur q;;

longueur : liste -> int = <fun>
```

Type somme paramétré par un type générique

```
type 'a liste = NIL | Cons of 'a * 'a liste;;
```

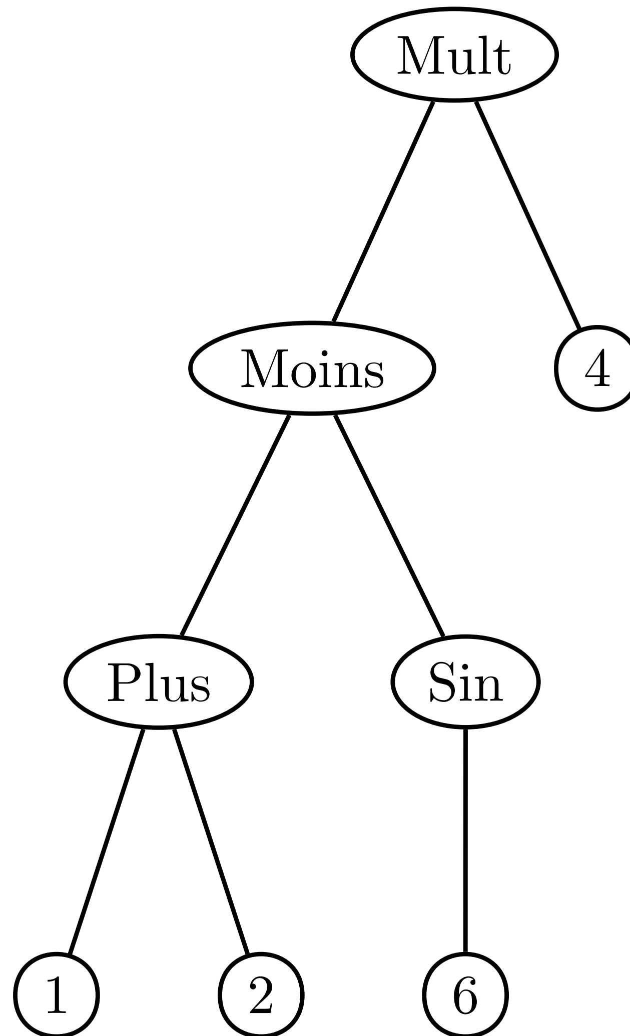
```
#let l = Cons (1.2, Cons(2.3, Cons(3.4, NIL)));;
l : liste = Cons (1.2, Cons (2.3, Cons (3.4, NIL)))
```

Expressions algébriques

Définition inductive d'une expression algébrique

1. des variables atomiques (de type float) ;
2. deux constructeurs d'arité 1 (fonctions) : `Sin`, `Exp` ;
3. quatre constructeurs d'arité 2 (opérateurs) : `Plus`, `Moins`, `Mult`, `Div`.

Un exemple $e = \big((1+2) - sin(6)\big) * 4$



Définition du type exp_alg en CAML

```
type exp_alg =
    Variable of float             (* arité 0*)
  | Sin of exp_alg                (* Fonctions : arité 1*)
  | Exp of exp_alg
  | Plus of exp_alg * exp_alg     (* Opérateurs : arité 2*)
  | Moins of exp_alg * exp_alg
  | Divise of exp_alg * exp_alg
  | Mult of exp_alg * exp_alg;;
```

Représentation abstraite d'une expression algébrique

$e = \big((1+2) - sin(6)\big) * 4$

```
let e = Mult(
  Moins(
    Plus(Variable 1., Variable 2.),
    Sin (Variable 6.)),
  Variable 4.);;
```

Sémantique : valeur d'une expression algébrique

```
let rec evalue e =
  match e with
    | Variable x -> x
    | Sin e1 -> sin (evalue e1)
    | Exp e1 -> exp (evalue e1)
    | Plus (e1, e2) -> (evalue e1) +. (evalue e2)
    | Moins (e1, e2) -> (evalue e1) -. (evalue e2)
    | Divise (e1, e2) -> (evalue e1) /. (evalue e2)
    | Mult (e1, e2) ->  (evalue e1) *. (evalue e2);;
```

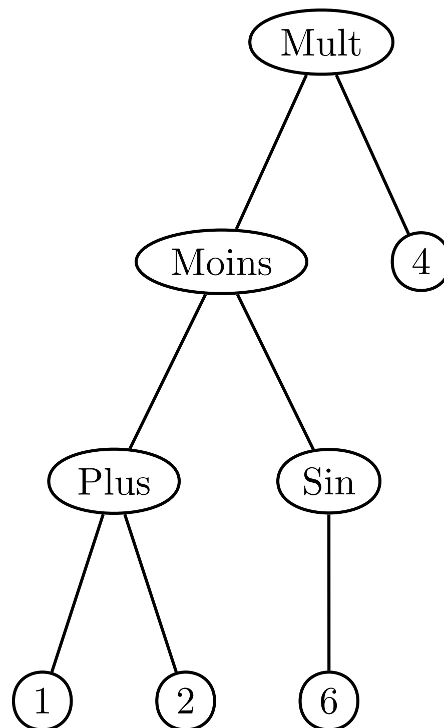
Syntaxe concrète - Syntaxe abstraite

- Syntaxe concrète : $e = \big((1+2) - sin(6)\big) * 4$. Facile à entrer pour un utilisateur, difficile à manipuler par un programme ;
- Syntaxe abstraite :

```
#let e = Mult(
  Moins(
    Plus(Variable 1., Variable 2.),
    Sin (Variable 6.)),
  Variable 4.);;
```

  Facile à manipuler pour un programme, difficile à entrer pour un utilisateur.

Notations linéaires d'une expression algébrique.



- Notation infixe : $1 \ + \ 2 \ - \ \sin \ 6 \ * \ 4$;
- Notation préfixe : $* \ - \ + 1 \ 2 \ \sin \ 6 \ 4$;
- Notation postfixe : $1 \ 2 \ + \ 6 \ \sin \ - \ 4 \ *$.

La notation infixe est ambigüe :

$$3 + 5 * 2 = (3 + 5) * 2 \text{ ou } 3 + (5 * 2)?$$

utilisation de parenthèses ou d'une priorité sur les constructeurs.

La notation postfixe n'est pas ambigüe.

Transformation en notation postfixe d'une expression

```
let rec postfixe e =  match e with
    | Variable x -> print_float x;
      print_string " "
    | Plus (e1, e2) ->  postfixe e1; postfixe e2;
      print_string "+ "
    | Moins (e1, e2) -> postfixe e1; postfixe e2;
      print_string "- "
    | Divise (e1, e2) -> postfixe e1; postfixe e2;
      print_string "/ "
    | Mult (e1, e2) ->  postfixe e1; postfixe e2;
      print_string "* "
    | Sin e1 -> postfixe e1; print_string "sin "
    | Exp e1 -> postfixe e1;  print_string "exp ";;
```

Évaluation d'une Expression Algébrique Postfixée à l'aide d'une pile

Représentation d'une expression algébrique

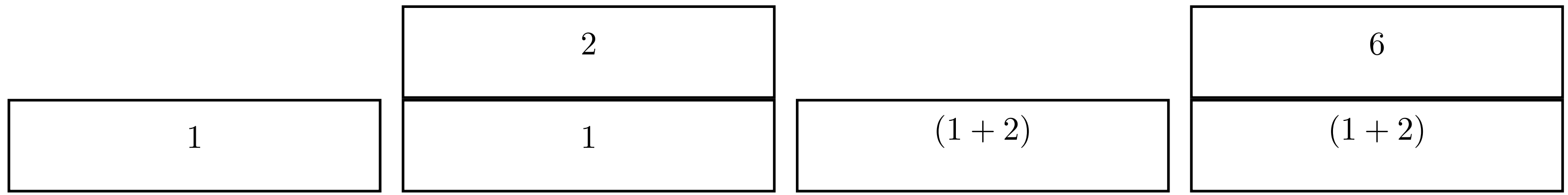
```
type opérateurs = L_plus | L_moins | L_mult | L_div;;
type fonctions = L_sin | L_exp;;
type lexème = L_var of float | L_fonc of fonctions
  | L_op of  opérateurs;;
```

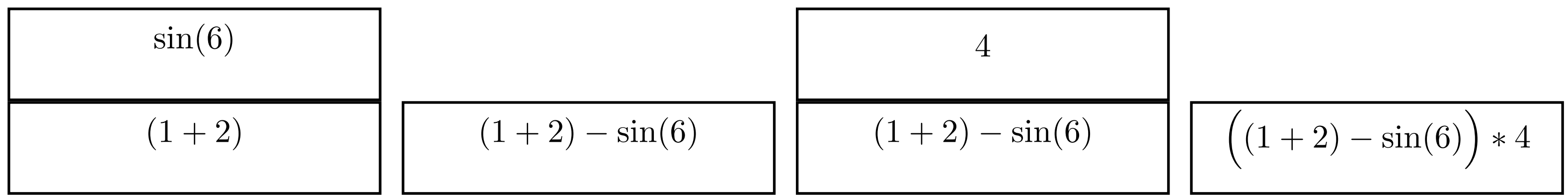
$e = 1\ 2\ +\ 6\ \sin\ -\ 4\ *$

```
let l = [L_var 1.; L_var 2.; L_op L_plus; L_var 6.;
L_fonc L_sin;L_op L_moins; L_var 4.; L_op L_mult];;
```

Algorithme d'évaluation

- Si on lit un lexème représentant une variable, on empile la valeur de la variable ;
- Si on lit un lexème représentant une fonction, on dépile la valeur au sommet de pile, on applique la fonction correspondante à cette valeur et on empile le résultat ;
- Si on lit un lexème représentant un opérateur, on dépile deux valeurs de la pile, on applique l'opérateur à ces deux valeurs et on empile le résultat.


2
6
1
1
$(1+2)$
$(1+2)$
Var 1
Var 2
Op Plus
Var 6


$\sin(6)$
4
$(1+2)$
$(1+2)-\sin(6)$
$(1+2)-\sin(6)$
$\Big((1+2)-\sin(6)\Big)*4$
Fonc Sin
Op Moins
Var 4
Op Mult

Théorème 12. ***Preuve de l'algorithme***

1. *Si l'expression est une EAP syntaxiquement correcte, lors du parcours gauche-droite, la pile contient toujours au moins un élément et à la fin du parcours elle contient un seul élément.*

2. *La valeur correspondant à l'évaluation de l'EAP est l'unique élément qui se trouve dans la pile à la fin du parcours.*

```
let evalue expression =
  let p = new() in
  let traite_var v =
    push v p
  and traite_fonc f =
    match f with
      | L_sin -> let x = pop p in push (sin x) p
      | L_exp -> let x = pop p in push (exp x) p
  and traite_op o =
    let x = pop p in
    let y = pop p in
      match o with
        | L_plus -> push (x +. y) p
        | L_moins -> push (x -. y) p
        | L_mult -> push (x *. y) p
```

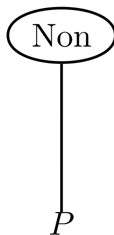
```
      | L_div -> push (x /. y) p
in
let traite_lexème lex =
  match lex with
    | L_var v -> traite_var v
    | L_fonc f -> traite_fonc f
    | L_op o -> traite_op o
in
let rec parse l =
  match l with
    | [] -> pop p
    | lex :: q -> traite_lexème lex; parse q
in
  parse expression;;
```

Chapitre 6

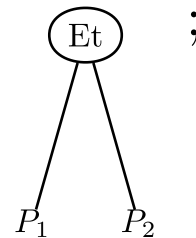
Logique

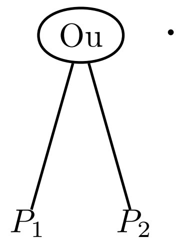
Syntaxe des propositions logiques

Définition inductive d'une proposition

- Les éléments de base sont les variables propositionnelles v (un certain ensemble extérieur) et les deux constantes V et F, représentés par une feuille d'un arbre (v);
- Si P est une proposition, Non(P) est une proposition, représentée par l'arbre 

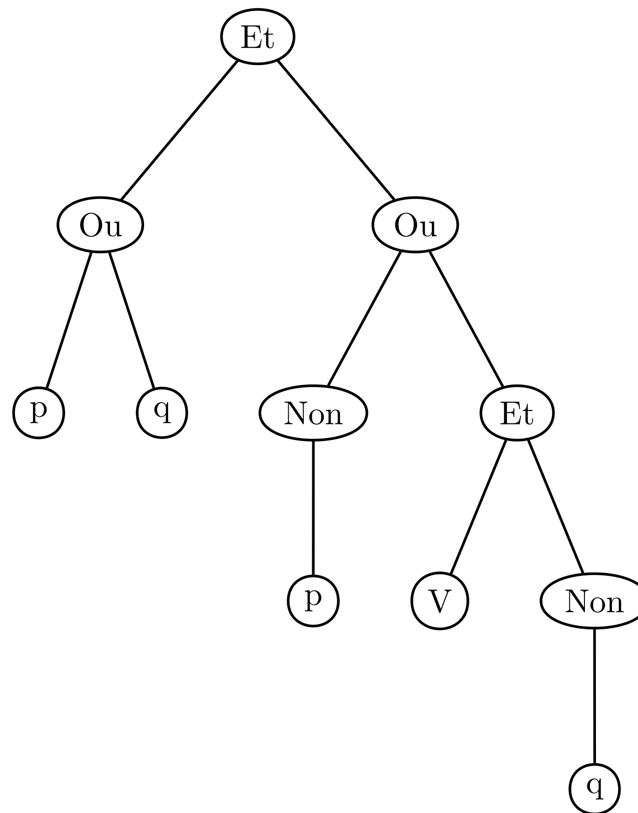
;

- Si P_1 et P_2 sont deux propositions, $\mathrm{Et}(P_1, P_2)$ est une proposition, représentée par l'arbre  ;

- Si a et b sont deux propositions, $\mathrm{Ou}(a, b)$ est une proposition, représentée par l'arbre  .

Un exemple

```
A = Et(  Ou(p, q), Ou( Non(p), Et(V, Non(q)) ) )
```



En CAML

```
type proposition =
    V
  | F
  | Var of char
  | Non of  proposition
  | Et of proposition * proposition
  | Ou of proposition * proposition;;
```

```
let p = Et(
  Ou(Var `p`, Var `q`),
  Ou(
    Non(Var `p`),
    Et(V, Non(Var `q`))
  )
);;

p : proposition =
 Et (Ou (Var `p`, Var `q`), Ou (Non (Var `p`),
Et (V, Non (Var `q`))))
```

Notation linéaire infixe parenthésée

- Si $P = v$ où v est une variable, $v = V$ ou $v = F$, $\tilde{P} = v$, (on identifie v avec le caractère correspondant) ;
- Si $P = \text{Non}(p_1)$, $\tilde{P} = \neg\tilde{p_1}$;
- Si $P = \text{Et}(p_1, p_2)$, $\tilde{P} = (\tilde{p_1} \wedge \tilde{p_2})$;
- Si $P = \text{Ou}(p_1, p_2)$, $\tilde{P} = (\tilde{p_1} \vee \tilde{p_2})$.

En CAML

```
let rec linéaire p =
  match p with
    | V -> print_char `V`
    | F -> print_char `F`
    | Var c ->  print_char c
    | Non p1 -> print_string " non "; linéaire p1
```

```
| Et (p1, p2) ->
    print_char ‘(‘;
    linéaire p1;
    print_string " et ";
    linéaire p2;
    print_char ‘)‘
| Ou (p1, p2) ->
    print_char ‘(‘;
    linéaire p1;
    print_string " ou ";
    linéaire p2;
    print_char ‘)‘;;
```

```
# let p = Et(
  Ou(Var ‘p‘, Var ‘q‘),
  Ou(
    Non(Var ‘p‘),
    Et(V, Non(Var ‘q‘))
  )
);;

# linéaire p;;
((p ou q) et ( non p ou (V et  non q)))
```

PREUVES : Induction structurelle

Sémantique

Valeur d'une proposition logique

Distribution de vérité : σ : Vars $\mapsto \{0,1\}$.

Définition inductive de $V_\sigma : Prop \mapsto \{0,1\}$:

- Si $P = V$, $V_\sigma(P) = 1$;
- Si $P = F$, $V_\sigma(P) = 0$;
- Si $P = v$, $V_\sigma(P) = \sigma(v)$;
- Si $P = \text{Non}(P_1)$, $V_\sigma(P) = 1 - V_\sigma(p_1)$;
- Si $P = \text{Et}(P_1, P_2)$, $V_\sigma(P) = \min\big(V_\sigma(P_1), V_\sigma(P_2)\big)$;
- Si $P = \text{Ou}(P_1, P_2)$, $V_\sigma(P) = \max\big(V_\sigma(P_1), V_\sigma(P_2)\big)$.

Propositions logiquement équivalentes

Soient deux propositions logiques P et Q définies à partir d'un ensemble de variables Vars. On dit que les deux propositions sont *logiquement équivalentes*, et on note $P \equiv Q$ lorsque pour toute distribution de vérité $\sigma : \text{Vars} \mapsto \{0, 1\}$, $V_\sigma(P) = V_\sigma(Q)$.

Table de vérité

Deux propositions peuvent être logiquement équivalentes, mais syntaxiquement très différentes. Par exemple, pour deux variables a, b,

$$P = a \vee \neg b, \quad Q = \neg(\neg a \wedge b)$$

Pour vérifier que $P \equiv Q$, il suffit de regrouper toutes les valeurs possibles des variables dans une *table de vérité* :

a	b	P	Q
1	1	1	1
1	0	1	1
0	1	0	0
0	0	1	1

Nombre de lignes d'une table de vérité

Si le nombre de variables vaut n, il y a 2^n distributions de vérité $\sigma : \text{Vars} \mapsto \{0, 1\}$ différentes.

Chaque distribution de vérité correspond à une ligne de la table de vérité.

Obtention de propositions logiquement équivalentes

Soient P et Q sont deux propositions formées à partir des variables $(v_1, \ldots, v_n)$. Si P et Q sont logiquement équivalentes, on en déduit de nouvelles proposisions logiquement équivalentes en remplaçant les variables par des propositions quelconques. Par exemple :

$$(p \vee q) \wedge \neg(p \wedge q) \equiv (p \wedge \neg q) \vee (q \wedge \neg p)$$

$$\begin{aligned} & \big((a \wedge b) \vee (\neg a \wedge c)\big) \wedge \neg\big((a \wedge b) \wedge (\neg a \wedge c)\big) \\ \equiv\ & \big((a \wedge b) \wedge \neg(\neg a \wedge c)\big) \vee \big((\neg a \wedge c) \wedge \neg(a \wedge b)\big) \end{aligned}$$

Définition de nouveaux connecteurs

p	q	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \Rightarrow q$	$p \Longleftrightarrow q$	p NAND q	p NOR q
1	1	1	1	0	1	1	0	0
1	0	0	1	1	0	0	1	0
0	1	0	1	1	1	0	1	0
0	0	0	0	0	1	1	1	1

On définit alors par induction la sémantique de ces nouveaux connecteurs. Pour une distribution de vérité σ, et deux propositions P et Q, on pose :

$$V_\sigma(P \Rightarrow Q) = \begin{cases} 0 & \text{si } V_\sigma(P) = 1 \text{ et } V_\sigma(Q) = 0 \\ 1 & \text{sinon} \end{cases}$$

De même pour $\oplus$, $\Longleftrightarrow$, NAND et NOR .

Implication

Montrer que $P \Rightarrow Q \equiv \neg P \vee Q$.

De même, on montre que $P \iff Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$.

Système de connecteurs complet

Définition 3. On dit qu'un système S de connecteurs est *complet* lorsque toute proposition est logiquement équivalente à une proposition formée uniquement à l'aide des connecteurs de S.

p	q	p NAND q
1	1	0
1	0	1
0	1	1
0	0	1

Montrer que $S = \{NAND\}$ est un système complet.

Conversion d'une proposition en NAND

```
type proposition =
    Var of char
  | Non of  proposition
  | Et of proposition * proposition
  | Ou of proposition * proposition
  | Nand of proposition * proposition;;
```

Écrire une fonction `convertit : proposition -> proposition` qui convertit une proposition p en une proposition logiquement équivalente n'utilisant que le constructeur `Nand`.

Calcul propositionnel

Notations allégées

- $\neg p$ noté $\overline{p}$;
- $p \wedge q$ noté $p.q$;
- $p \vee q$ noté $p + q$.

Lois de De Morgan Pour toutes propositions P, Q :

- $\neg(P \wedge Q) \equiv \neg P \vee \neg Q, \quad \overline{P.Q} \equiv \overline{P} + \overline{Q}$;
- $\neg(P \vee Q) \equiv \neg P \wedge \neg Q, \quad \overline{P + Q} \equiv \overline{P}.\overline{Q}$.

Équivalences logiques classiques

- $P.P \equiv P$;
- $P + P \equiv P$ (idempotence) ;
- $\overline{\overline{P}} \equiv P$;
- $(P.Q).R \equiv P.(Q.R)$;
- $(P + Q) + R \equiv P + (Q + R)$ (associativité) ;
- $P + Q \equiv Q + P$;
- $P.Q \equiv Q.P$ (commutativité) ;
- $(P.Q) + R \equiv (P + R).(Q + R)$;
- $(P + Q).R \equiv (P.R) + (Q.R)$ (distributivité) ;

- $P + \overline{P} \equiv V$ (tiers exclu) ;
- $P.\overline{P} \equiv F$;
- $P + F \equiv P$;
- $P + V \equiv V$;
- $P.V \equiv P$;
- $P.F \equiv F$;
- $P + (P.Q) \equiv P$;
- $P.(P + Q) \equiv P$;
- $P \Rightarrow Q \equiv \overline{Q} \Rightarrow \overline{P}$ (raisonnement par contraposition) ;
- $P \Longleftrightarrow Q \equiv (P \Rightarrow Q).(Q \Rightarrow P)$;
- $P \Rightarrow Q \equiv (\overline{P} + Q)$ (expression de $\Rightarrow$).

Formes disjonctives, conjonctives

Littéral

Définition 4. Une proposition P est un *littéral* lorsqu'elle est de la forme $P = v$ ou $P = \overline{v}$, où v est une variable propositionnelle.

Définition 5. **Forme conjonctive, disjonctive**

Soit une proposition logique P formée à partir de variables propositionnelles $v_1, \ldots, v_k$. On dit que P est une *forme conjonctive* si P s'écrit

$$P = P_1 \wedge P_2 \wedge \ldots \wedge P_n$$

où les P_i sont des *clauses*, c'est à dire des disjonctions de littéraux (par exemple $P_i = v_1 \vee \overline{v_3} \vee v_5$).

On dit que P est une *forme disjonctive* si P s'écrit

$$P = P_1 \vee P_2 \vee \ldots \vee P_n$$

où chaque P_i est une conjonction de littéraux (par exemple $P_i = \overline{v_2} \wedge v_3 \wedge \overline{v_4}$).

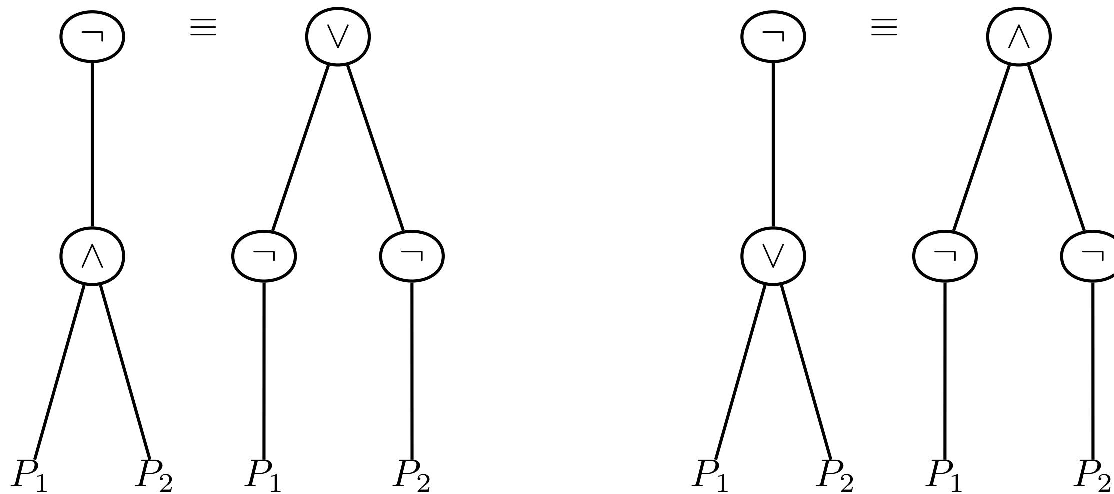
Algorithme de mise sous forme conjonctive (disjonctive)

Étape 1

Remplacer tous les connecteurs apparaissant dans P en fonction des connecteurs basiques $\neg$, $\wedge$ et $\vee$;

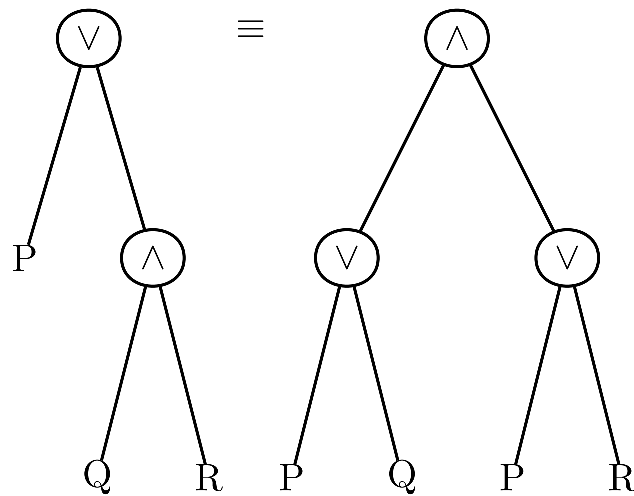
Étape 2

Utiliser les lois de De Morgan pour faire descendre au maximum les connecteurs $\neg$: $\neg(P_1 \wedge P_2) \equiv \neg P_1 \vee \neg P_2$ et $\neg(P_1 \vee P_2) \equiv \neg P_1 \wedge \neg P_2$.



Étape 3

Utiliser la distributivité de $\vee$ par rapport à $\wedge$ pour faire descendre dans l'arbre les $\vee$: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$.



Exemple

$$P = (a \Rightarrow b) \Rightarrow (\neg c \wedge b)$$

1. $P \equiv \neg(\neg a \vee b) \vee (\neg c \wedge b)$
2. $P \equiv (a \wedge \neg b) \vee (\neg c \wedge b)$
3. $P \equiv [a \vee (\neg c \wedge b)] \wedge [\neg b \vee (\neg c \wedge b)]$
4. $P \equiv [(a \vee \neg c) \wedge (a \vee b)] \wedge [(\neg b \vee \neg c) \wedge (\neg b \vee b)]$
5. $P \equiv (a \vee \neg c) \wedge (a \vee b) \wedge (\neg b \vee \neg c) \wedge (\neg b \vee b)$

Exercice

Mettre sous une forme disjonctive puis conjonctive la proposition

$$P = (p \Longleftrightarrow q) \text{ NAND } (q \Rightarrow r)$$

Exercice Un coffre-fort est muni de n serrures et peut être ouvert uniquement lorsque ces n serrures sont simultanément ouvertes. Cinq personnes : a, b, c, d et e doivent recevoir des clés correspondant à certaines serrures. Chaque clé peut être disponible en autant d'exemplaires qu'on le souhaite. On cherche n et une répartition des clés entre les cinq personnes, de telle manière que le coffre puisse être ouvert si et seulement si on se trouve dans une au moins des situations suivantes :

- présence simultanée de a et b ;
- présence simultanée de a, c et d ;
- présence simultanée de b, d et e.

On considère cinq variables propositionnelles notées a, b, c, d et e. La variable x vaut 1 si et seulement si x est présent.

1. Exprimer la proposition “ le coffre-fort peut-être ouvert ” à l’aide d’une proposition logique A sous forme disjonctive.
2. Transformer A en une forme conjonctive équivalente.
3. Déterminer une valeur de n et une répartition possible.

Formes normales

Définition 6. **Mintermes, maxtermes**

- On appelle *minterme*, constitué de k variables propositionnelles $(v_1, \ldots, v_k)$, une conjonction de littéraux dans laquelle figure exactement une fois chacune des k variables. Par exemple, $p.q$, $\overline{p}.q$, $p.\overline{q}$ et $\overline{p}.\overline{q}$ sont tous les mintermes que l'on peut former en les deux variables (p, q).
- On appelle *maxterme* une disjonction de littéraux dans laquelle figure exactement une fois chacune des k variables. Les maxtermes formés à l'aide de deux variables sont $p + q$, $\overline{p} + q$, $p + \overline{q}$ et $\overline{p} + \overline{q}$.

Définition 7. **Formes normales**

Soit P une proposition logique formée à l'aide de k variables $v_1, \ldots, v_k$. On appelle :

- *Forme normale conjonctive* de P, toute proposition logiquement équivalente à P qui s'écrit comme une conjonction de maxtermes.
- *Forme normale disjonctive* de P, toute proposition logiquement équivalente à P qui s'écrit comme une disjonction de mintermes.

Exemple

Variables : p, q, r.

- Une forme normale conjonctive :

$$P = (\overline{p} + q + r).(p + \overline{q} + \overline{r}).(p + q + r)$$

- Une forme normale disjonctive :

$$Q = (p.\overline{q}.r) + (\overline{p}.q.\overline{r})$$

Théorème 13. ***Mise sous forme normale d'une proposition***

Toute proposition logique formée à l'aide de k variables propositionnelles est logiquement équivalente à une proposition sous forme normale disjonctive (respectivement conjonctive).

De plus, cette forme normale est unique à l'ordre près.

Utilisation des tables de vérité

p	q	r	P
1	1	1	0
1	1	0	1
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	1

$$P \equiv (p.q.\overline{r}) + (p.\overline{q}.r) + (\overline{p}.q.r) + (\overline{p}.\overline{q}.\overline{r})$$

Forme normale conjonctive

p	q	r	P
1	1	1	0
1	1	0	1
1	0	1	1
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	0
0	0	0	1

$$\overline{P} \equiv (p.q.r) + (p.\overline{q}.\overline{r}) + (\overline{p}.q.\overline{r}) + (\overline{p}.\overline{q}.r)$$

$$\overline{P} \equiv (p.q.r) + (p.\overline{q}.\overline{r}) + (\overline{p}.q.\overline{r}) + (\overline{p}.\overline{q}.r)$$

Lois de De Morgan :

$$P \equiv (\overline{p} + \overline{q} + \overline{r}).(\overline{p} + q + r).(p + \overline{q} + r).(p + q + \overline{r})$$

Exercice

1. Montrer qu'il existe un unique connecteur logique ϕ à trois variables $\phi(p, q, r)$ tel que pour toute variable s :

$$\phi(s, \overline{s}, s) \equiv \phi(s, F, F) \equiv V \text{ et } \phi(s, s, \overline{s}) \equiv \phi(s, V, V) \equiv F$$

2. Donner la forme normale disjonctive de la proposition $A = \phi(p, q, r)$, puis sa forme normale conjonctive ;

3. Existe-t-il une forme disjonctive plus simple équivalente à A ?

Tautologies

Définition 8. **Satisfiabilité**

On dit qu'une proposition P est *satisfiable* s'il existe une distribution de vérité σ telle que $V_\sigma(P) = 1$.

On dit alors que σ *satisfait* P.

SAT : problème NP-complet.

Définition 9. **Tautologie**

Soit P une proposition. Si pour *toute* distribution de vérité σ, la valeur $V_\sigma(P)$ vaut 1, on dit que A est une *tautologie*.

Exemple de tautologies

- $(p \wedge q) \Rightarrow p$;
- $p \Rightarrow (p \vee q)$;
- $\big(p \wedge (p \Rightarrow q)\big) \Rightarrow q$;

Obtention de nouvelles tautologies

Si A est une tautologie, on peut remplacer les variables propositionnelles de A par des propositions quelconques, et obtenir de nouvelles tautologies.

$$(p \wedge q) \Rightarrow p$$

$$(P \wedge Q) \Rightarrow P$$

$$((a \wedge b) \wedge (a + c)) \Rightarrow (a \wedge b)$$

$$((a \wedge (b \vee c)) \wedge (c \Rightarrow a)) \Rightarrow (a \wedge (b \vee c))$$

Preuves logiques

Conséquence logique

Une proposition Q est une conséquence logique d'une proposition P ssi pour toute distribution de vérité telle que $V_\sigma(P) = 1$, on a $V_\sigma(Q) = 1$. En d'autres termes, si et seulement si

$$P \Rightarrow Q$$

est une tautologie.

Preuve

Hypothèses : $H_1, \ldots, H_n$ des propositions dont la valeur logique est supposée vraie ;

Conclusion : C une proposition.

Preuve : Montrer que C est une conséquence logique de $H_1 \wedge \ldots \wedge H_n$.

Méthode par les tables de vérité

Écrire la table de vérité de la proposition

$$(H_1 \wedge \ldots \wedge H_n) \Rightarrow C$$

pour montrer que c'est une tautologie.

Inconvénient : complexité exponentielle dans tous les cas.

Autre méthode

Des hypothèses $H_1, \ldots, H_n$, on dérive de proche en proche des conséquences logiques $E_1, \ldots, E_k$ avec $E_k = C$.

Règle d'élimination en utilisant la tautologie :

$$\big((p \vee q) \wedge (\neg p \vee r)\big) \Rightarrow (q \vee r) \tag{1}$$

Élimination de variables

$$\big((p+q).(\overline{p}+r)\big) \Rightarrow (q+r)$$

p	q	r	$(p+q).(\overline{p}+r)$	$q+r$	$(p+q).(\overline{p}+r) \Rightarrow q+r$
1	1	1	1	1	1
1	1	0	0	1	1
1	0	1	1	1	1
1	0	0	0	0	1
0	1	1	1	1	1
0	1	0	1	1	1
0	0	1	0	1	1
0	0	0	0	0	1

Autre technique similaire

Par l'absurde, montrer que $Faux$ est une conséquence logique des hypothèses

$$(H_1 \wedge \ldots \wedge H_n) \wedge \neg C$$

Méthode de résolution

1. Mettre les hypothèses et $\neg C$ sous forme conjonctive (pas forcément normale) : $H_1 = C_{11} \wedge \ldots \wedge C_{1p_1}, \ldots,$ $H_n = C_{n1} \wedge \ldots \wedge C_{np_n}$;

2. On obtient alors $p_1 \times \cdots \times p_n$ hypothèses H'_i qui sont des *clauses* (disjonctions de littéraux) ;

3. Utiliser systématiquement la règle d'élimination

$$\big((p \vee q) \wedge (\neg p \vee r)\big) \Rightarrow (q \vee r)$$

   pour obtenir des étapes E_k, en éliminant des variables.

4. Aboutir à la conséquence logique F.

Exemple

- “ S’il pleut, je prends mon imperméable ” ;
- “ Si j’ai mon imperméable, je ne suis pas mouillé ” ;
- “ S’il ne pleut pas, je ne suis pas mouillé ”.

Montrons que je ne serai jamais mouillé. Considérons les trois variables propositionnelles

1. p : “ il pleut ” ;
2. i : “ je prends mon imperméable ” ;
3. m : “ je suis mouillé ”.

Exemple de raisonnement par contradiction

1. $Hy_1 = p \Rightarrow i$;
2. $Hy_2 = i \Rightarrow \overline{m}$;
3. $Hy_3 = \overline{p} \Rightarrow \overline{m}$;
4. $C = \overline{m}$. (conclusion)

1. $H_1 = \overline{p} + i$;
2. $H_2 = \overline{i} + \overline{m}$;
3. $H_3 = p + \overline{m}$;
4. $H_4 = m$; $(\overline{C})$

1. $E_1 = i + \overline{m}$ (élimination de H_1 et H_3) ;
2. $E_2 = \overline{m} + \overline{m} \equiv \overline{m}$ (élimination de H_2 et E_1) ;
3. $E_3 = \overline{m} \wedge m \equiv F$ (simplification de H_4 et E_2).

Exercice

1. Lorsque je résous un exercice sans grogner, je le comprends
2. Les exercices de logique ne sont pas présentés comme les exercices auquels je suis habitué
3. Les exercices faciles ne me donnent pas mal à la tête
4. Je ne comprends pas les exercices qui ne sont pas présentés comme ceux auquels je suis habitué
5. Je ne grogne jamais devant un exercice sauf s'il me donne mal à la tête

Montrer en utilisant la méthode de résolution que les exercices de logique ne sont pas faciles.

Circuits logiques

Fonctions booléennes

Définition 10. **Fonction booléenne** On appelle *fonction booléenne* de k variables, une fonction $f : \{0, 1\}^k \mapsto \{0, 1\}$.

Proposition logique $\mapsto$ fonction booléenne

On associe à une proposition logique formée de k variables, une fonction booléenne $f_P : \{0,1\}^k \mapsto \{0,1\}$ obtenue à partir de la table de vérité.

Fonction booléenne $\mapsto$ proposition logique

A toute fonction booléenne $f : \{0,1\}^k \mapsto \{0,1\}$, on peut trouver une proposition logique P formée de k variables propositionnelles telle que $f = f_P$: il suffit d'écrire la table de la fonction booléenne et d'en déduire la forme normale disjonctive de P.

Circuit logique

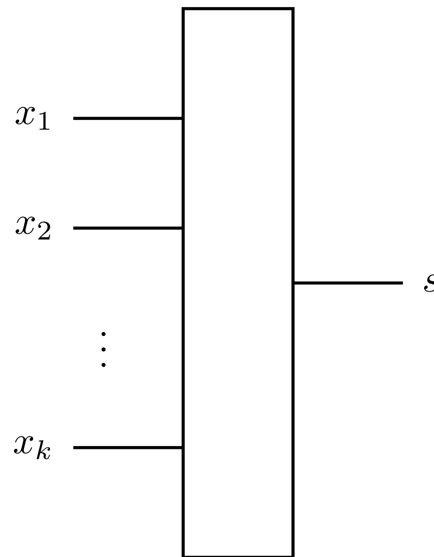
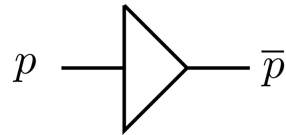


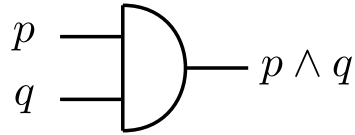
Figure 8: Un circuit électrique à k entrées et une sortie

Un circuit *câble* une fonction booléenne $f : \{0,1\}^k \mapsto \{0,1\}$ telle que $s = f(x_1, \ldots, x_k)$ représente la valeur logique de la sortie si les points d'entrée du circuit sont affectés des valeurs logiques $x_1, \ldots, x_k$.

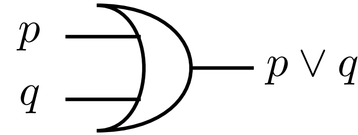
Portes logiques élémentaires (logigrammes)



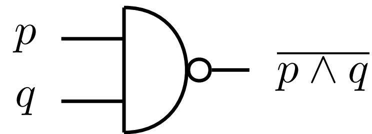
porte NON



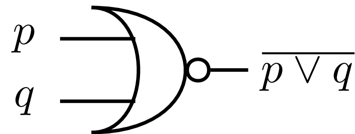
porte ET



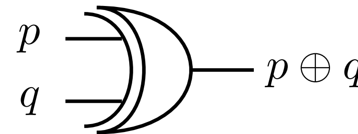
porte OU



porte NAND



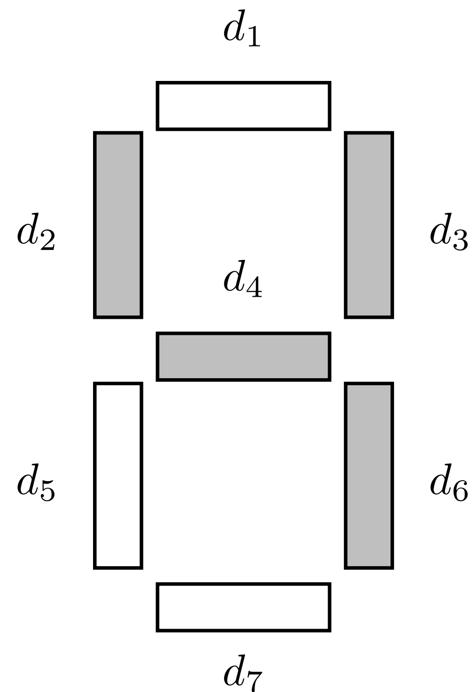
porte NOR



porte XOR

Figure 9: Portes logiques élémentaires

Un afficheur LCD



Câbler la fonction d'allumage $f_3(b_1, b_2, b_3)$ de la diode d_3, si l'entrée correspond à un entier $n \in [0, 7]$ codé sur 3 bits, $n = (b_1 b_2 b_3)_2$.

Conception d'un circuit

- Temps de réponse d'un circuit ;
- Minimiser le nombre de portes logiques utilisées.

Un additionneur n bits

Position du problème

- $2n$ points d'entrée correspondant à deux entiers x et y codés sur n bits :

$$x = (x_n \dots x_1)_2 \quad y = (y_n \dots y_1)_2$$

- n points de sortie correspondant à l'entier

$$z = x + y = (z_n \dots z_1)_2$$

Demi-additionneur 1 bit

p	q	a	r
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

Proposition associée : $a = p \oplus q$, $r = p \wedge q$.

Réalisation d'un demi-additionneur 1 bit

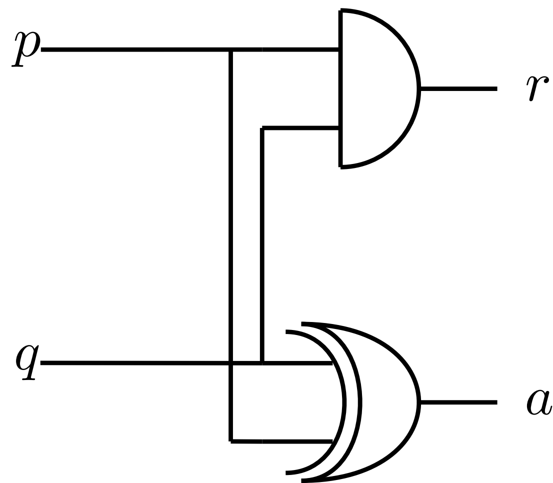
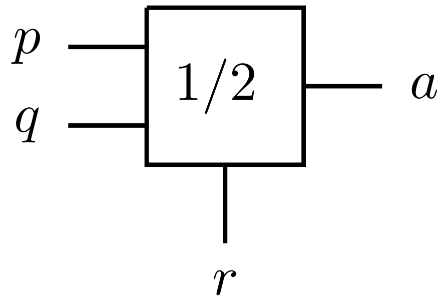


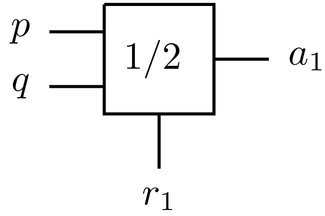
Schéma d'un demi-additionneur 1-bit

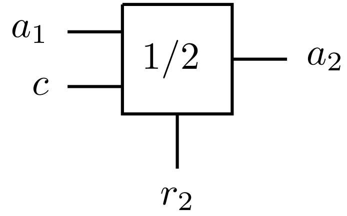


additionneur 1 bit avec retenue

Entrée : deux bits p, q et une retenue à prendre en compte c. Sortie : $a = p + q + c$ et r (nouvelle retenue).

Idée : Mettre en série deux demi-additionneurs 1 bits.

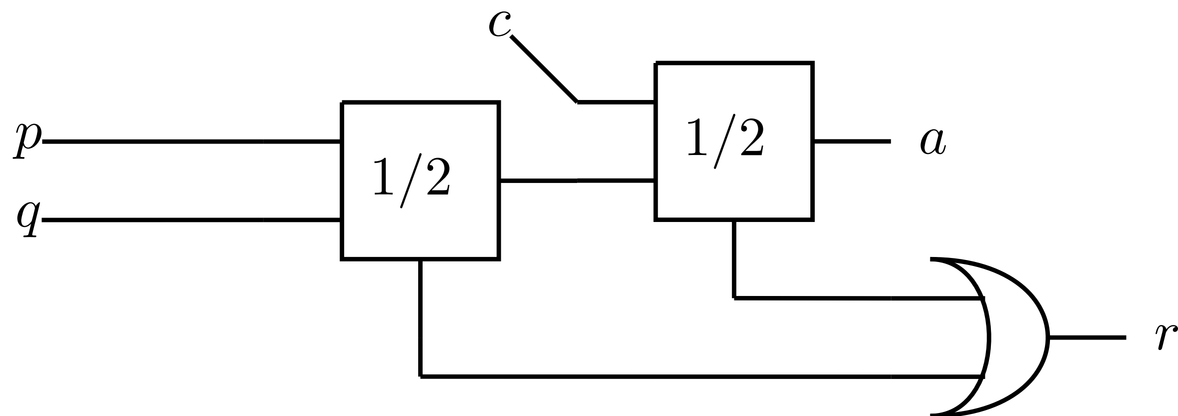




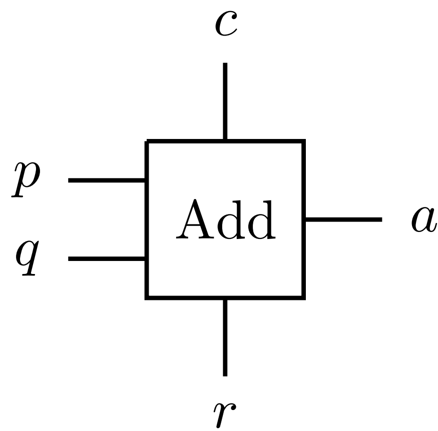
$$\begin{cases} a = (p + q + c),\ r \text{ retenue} \\ a = a_2 \\ r = r_1 \vee r_2 \end{cases}$$

p	q	c	a_1	r_1	a_2	r_2	a	r
1	1	1	0	1	1	0	1	1
1	1	0	0	1	0	0	0	1
1	0	1	1	0	0	1	0	1
1	0	0	1	0	1	0	1	0
0	1	1	1	0	0	1	0	1
0	1	0	1	0	1	0	1	0
0	0	1	0	0	1	0	1	0
0	0	0	0	0	0	0	0	0

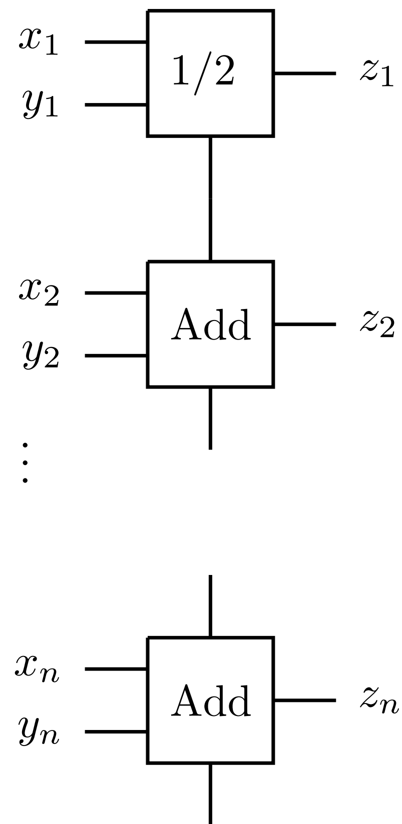
Câblage d'un additionneur 1-bit



Additionneur 1-bit



L'additionneur complet Mise en série d'un demi-additionneur et d'additionneurs 1-bit.



Exercice : comparateur

Soient x, y deux variables. On note $T(x, y)$ la fonction booléenne qui vaut 1 lorsque $x > y$ et 0 sinon. Donner sa table de vérité et exprimer T par une formule logique.

Dessiner un circuit logique TEST dont les entrées sont x et y et les sorties sont a et b définies par : $a = 1$ si et seulement si $x > y$, $b = 1$ si et seulement si $y > x$.

Concevoir, en utilisant la méthode diviser pour régner, un circuit logique dont les entrées sont $x_0, x_1, \ldots, x_{2^n-1}, y_0, y_1, \ldots, y_{2^n-1}$ et les sorties sont a et b définies par : $a = 1$ si et seulement si $\overline{x_{2^n-1} \ldots x_1 x_0} > \overline{y_{2^n-1} \ldots y_1 y_0}$, $b = 1$ si et seulement si $\overline{y_{2^n-1} \ldots y_1 y_0} > \overline{x_{2^n-1} \ldots x_1 x_0}$.