Lycée

aidherbe

Cours d'informatique : option

Liste des chapitres

Chapitre I

De Python à Caml

1 Introduction

1-1 Où en sommes nous ?

Voici la définition de l'informatique proposée par la Société Informatique de France (SIF).

Informatique

L'informatique est la science et la technique de la représentation de l'information d'origine artificielle ou naturelle, ainsi que des processus algorithmiques de collecte, stockage, analyse, transformation et communication de cette information, exprimés dans des langages formels ou des langues naturelles et effectués par des machines ou des êtres humains, seuls ou collectivement.

On peut noter que le premier thème mis en évidence est la représentation de l'information.

Le langage Python

L'étude de **Python** proposée en informatique commune est centrée sur une représentation des données par les listes. Cette structure de donnée est très riche et permet de traiter de nombreux problèmes. **Python** permet, de plus, d'utiliser des structures de données plus finement adaptée à un problème, en particulier à l'aide de la programmation par objets.
La programmation en **Python** est la continuation des langages développés à partir de la structure des ordinateur : on utilise un ensemble de fonctions et procédures qui modifient un environnement composés de variables dont les valeurs sont modifiées.
Cette notion d'environnement de travail est la source de l'extraordinaire souplesse de ces langages mais induit des limitations dans le cas de programmes complexes. En particulier il est difficile d'étudier formellement les programmes : leur richesse rend ardue leur traduction dans une forme logique rigoureuse.

1-2 Un autre type de programmation

Dès la fin des années 50 d'autres types de langages ont été proposés.
Ils sont structurés autours de concepts indépendants des ordinateurs ; pour les langages qui nous intéressent ils s'appuient sur une formulation de la logique imaginée vers 1930, le **lambda-calcul**.

On peut citer LISP (1958), ML (1974) dont CAML est une évolution, MIRANDA, `Haskell` …
Le principe de ces langages fonctionnels est de considérer la programmation comme une suite de fonctions appliquées à des éléments persistants (dont la valeur ne change pas) qui ont un **type** déterminé.
Une fonction est, dans ce cadre, un objet comme un autre dont le type est composé.
Par exemple, si `int` désigne le type des entiers, la fonction `successeur` qui envoie n en $n+1$ sera un élément de type `int -> int` : elle transforme un entier en entier.
Cette méthode de programmation s'est avérée très efficace pour développer des programmes surs : les langages modernes contiennent souvent une partie fonctionnelle.
C'est cette méthode de programmation que nous allons étudier avec le langage **`OCaml`**.

Propriétés de `OCaml`

1. **Partie impérative**
   `OCaml` n'est **pas** un langage fonctionnel pur ; il contient toutes les fonctions impératives classiques. On peut donc l'utiliser avec la souplesse des langages classiques ; on perd alors la rigueur des langages fonctionnels.
   On peut écrire beaucoup de programmes comme en **`Python`** sans d'autres modifications que les différences de syntaxe.

2. **Typage fort**
   `OCaml` est un langage typé. Les éléments ont un type fixé.
   En particulier les fonctions seront définies pour un type de données et donneront un résultat de type homogène.
   Contrairement à beaucoup de langages le typage est très contraignant : si on veut définir l'entier 2 comme un flottant (2.0) on doit le convertir.
   On ne peut donc pas calculer 2 + 3.5.
   Le langage va jusqu'à donner deux noms différents aux opérations dans $\mathbb{Z}$ et dans $\mathbb{R}$, "+ "et "+."

3. **Typage dynamique**
   Cette contrainte forte de typage permet que le type des variables soit déterminé par le langage : on n'a pas besoin de déclarer de quels types sont les éléments.

   Il se peut alors que le type ne soit pas unique, on parle alors de **polymorphisme**. L'exemple élémentaire est la fonction identité $x \mapsto x$; son type sera noté `'a -> 'a`.

4. **Récursivité**
   Comme la plupart des langages modernes **`OCaml`** permet d'utiliser la récursivité : ce sera un outil très utilisé dans cet enseignement.

2 Utilisation de OCaml

2-1 Installation

- On peut utiliser **OCaml** en ligne sur le site https://try.ocamlpro.com/.
  Le site contient quelques leçons d'initiation.

- Un collègue, Jean Mouric, a créé des outils qui permettent d'installer et d'utiliser simplement un environnement de travail.
  http://jean.mouric.pagesperso-orange.fr/
  Suivre les instructions correspondant à votre machine (il faut descendre dans la page, après la liste des mises-à-jour).

- Pour des installation plus poussées, on se référera au site officiel http://ocaml.org/.
  Il propose, sous Linux ou Mac OS, un gestionnaire de paquets : OPAM.

- Dans les machines (sous Linux) du lycée, le langage est installé dans la système.
  Il est accessible depuis un terminal par `ocaml` ou, plus confortablement, depuis un éditeur de texte. Ceux qui sont proposés sont
  ↪ `emacs` et le plug-in `tuareg` ou
  ↪ `gedit` avec le plug-in `external tools`.

2-2 Utilisation de la boucle interactive

OCaml peut être employé de manière rudimentaire dans une boucle interactive.
C'est ce qui est proposé dans le site https://try.ocamlpro.com/.

```
ericd13@eric-bureau ~ $ ocaml
        OCaml version 4.02.3

#
```

Le symbole # signifie que la boucle attend votre instruction. On peut alors entre les commandes.
On écrit des instructions les unes après les autres de manière accumulative.
Le programme les analyse, les interprète puis affiche les résultats éventuels et le type. On a la réponse après chaque instruction.

```
# 1+1;;
- : int = 2
#
```

Les double point-virgule sont la marque de fin d'une instruction.
La réponse est composée
↪ du nom de la variable définie (ici aucune d'où le tiret),
↪ du type du résultat du calcul (ici un entier),
↪ et de la valeur de ce calcul.

Les environnements ci-dessus utilisent la boucle interactive.

2-3 Utilisation d'un éditeur

De manière générale on écrit le texte dans un éditeur qui proposer propose plusieurs outils :
↪ coloration syntaxique,
↪ proposition de nom de variables à partir des premières lettres,
↪ numéros de lignes,
↪ ...

On peut alors envoyer le texte écrit à exécuter sous **`OCaml`**.
↪ Le plus souvent (`WinCaml`, `MacCaml`, `emacs`) les instructions sont ajoutées dans une boucle interactive.
↪ `emacs` permet d'écrire directement dans la console.
↪ Au contraire `Gedit` impose d'envoyer le fichier entier dans la boucle interactive, celle-ci est en fait ré-initialisée à chaque exécution (ce n'est plus vraiment une boucle interactive). Bien que contraignante, cette méthode permet un meilleur contrôle.

3 Syntaxe de OCaml

Nous allons ici donner les syntaxes des instructions de **OCaml**, il s'agit de traduire ce que nous savons faire en **Python**. Le prochain chapitre montrera des comportements nouveaux.

3-1 Variables

Pour un usage plus élaboré qu'une simple calculatrice nous devrons définir les variable du langage : cela se fait avec l'instruction **let**.

```
# let a = 1;;
a : int = 1
```

↪ La valeur de la variable est donnée au moment de sa création.
↪ Le type de la variable n'est pas déclaré, c'est le logiciel qui le définit.
↪ La variable n'est pas très variable : a sera toujours associé à 1 sauf si on définit une nouvelle variable avec le même nom ; dans ce cas l'ancienne variable est perdue.

On peut définir une variable pour un usage local : elle n'existe plus en dehors de son domaine d'application.

```
# let x = 2 in x+1;;
- : int = 3
# x;;
Error: Unbound value x
#
```

Il sera souvent profitable de lire les messages d'erreurs qui peuvent être explicatifs et pertinents. Ici le programme nous dit que x n'est pas lié à une valeur.
Si une variable globale existe avec le même nom elle est oubliée dans le domaine d'application de la variable locale puis reprend son existence.

```
# let u = 3;;
val u : int = 3
# let u = 4 in u+1;;
- : int = 5
# u;;
- : int = 3
```

Il est cependant possible de définir des variables dont on peut modifier la valeur.
Dans **OCaml** on les appelle variables référencées.

Une variable référencée n'est plus liée (bound en anglais) directement à la valeur associée mais à un conteneur dont le contenu est modifiable. C'est le même comportement que pour les éléments des listes en **Python**.

```
# let valeur = ref 0;;
val valeur : int ref = {content = 0}
```

`valeur` ne vaut pas 0, elle réfère à 0.
La valeur référencée est accessible par `!valeur`.
On peut alors changer la valeur référencée par une affectation de valeur :

```
valeur := !valeur + 1;;
```

Cette instruction ne produit rien : elle ne fait que modifier le contenu de la variable `valeur`.

3-2 Types simples

Entiers

Le principal type numérique que nous emploierons sera le type entier, `int` en **OCaml**.
↪ Les calculs seront exacts, contrairement aux flottants, à condition de rester dans les limites des nombres exprimables dans la machines : `min_int <= n <= max_int`
↪ Les opérations sont les opérations classiques : +, -, *, /, `mod`
↪ / désigne la division entière : 7/2 renvoie 3
↪ `mod` est le reste de la division ; `7 mod 2` renvoie 1.
↪ Il n'y a pas de fonction puissance.
↪ On imprime une valeur entière avec `print_int`.

```
# print_int 4;;
4- : unit = ()
```

On peut remarquer que cette instruction a un type, `unit`.

Flottants

OCaml peut utiliser des flottants : `float`
↪ Les calculs seront arrondis, la représentation en mémoire est la même que pour les flottants en **Python**.
↪ Les opérations sont classiques mais elles sont marquées d'un point : +. -. *. /.
↪ On ne peut pas mélanger entiers et flottants : 7.5 +. 2 renvoie une erreur.
↪ On imprime une valeur flottante avec `print_float`.

Il existe la fonction puissance notée ** et les fonctions usuelles, `sin`, `atan`, `log` …
Une particularité de **OCaml** est que les parenthèses ne sont pas nécessaires

```
# log 10.0;;
- : float = 2.30258509299
```

Pour les conversion on a les fonctions `int_of_float` et `float_of_int` dont les noms sont explicites.

Booléens

Les booléens (type `bool`) sont des variables qui peuvent prendre les valeurs `true` et `false` avec les opérateurs
↪ `et` noté && ou &
↪ `ou` noté || ou `or`
↪ `négation` notée `not`

Le résultat d'une comparaison (`=`, `<>`, `<`, `>`, `<=`, `>=`) est un booléen

```
# 4 < 2;;
- : bool = false
```

Caractères

Les caractères (type `char`) sont notés entre 2 guillemets simples, ce type comprend les lettres, les chiffres et les symboles affichables sur l'écran plus quelques autres (le caractère de fin de ligne `\n` par exemple).
Le code (`Ascii`) d'un caractère est accessible par `int_of_char` ; l'opération inverse est `char_of_int`

```
# char_of_int 54;;
- : char = `6`
# int_of_char `t`;;
- : int = 116
```

Chaînes de caractères

Les chaînes de caractères (type `string`) sont encadrées par des guillemets doubles, avec comme opérateur la concaténation de 2 chaînes :

```
# "Deux plus deux"^" font quatre";;
- : string = "Deux plus deux font quatre"
```

↪ La longueur d'une chaîne est donnée par `String.length`.
↪ On peut convertir les nombres en chaînes et réciproquement :
`string_of_int`, `string_of_float`,
`int_of_string`, `float_of_string`.
↪ On imprime une chaîne de caractère avec `print_string`.
↪ `print_newline ()` permet d'aller à la ligne.

Le caractère d'indice i d'une chaîne (le premier a 0 pour indice) est accessible par `ch.[i]` ;

```
# let ch = "Le langage camL";;
val ch : string = "Le langage camL"
# string_length ch;;
- : int = 15
# ch.[5];;
- : char = 'n'
```

Les chaînes de caractères sont modifiables mais cela est indiqué comme obsolète ; il vaut mieux éviter cette possibilité.

Le rien

Il existe un type qui exprime l'absence de valeur, le type `unit`.
Il a une valeur unique, notée `()` : c'est le type de retour des fonctions qui ne renvoient pas de résultat et des instructions de modifications

```
# print_int 4;;
4- : unit = ()
# somme := !somme + 1;;
- : unit = ()
```

3-3 Types composés

Couples

Un couple est un assemblage de 2 données. Il a les propriétés suivantes :
↪ les données ne sont pas forcement de même type
↪ les données ne sont pas modifiables.

On peut décomposer un couple avec `fst` et `snd`.

```
# let couple = (1, 1.0);;
val couple : int * float = 1, 1.0
# fst couple;;
- : int = 1
# snd couple;;
- : float = 1.0
```

Tuples

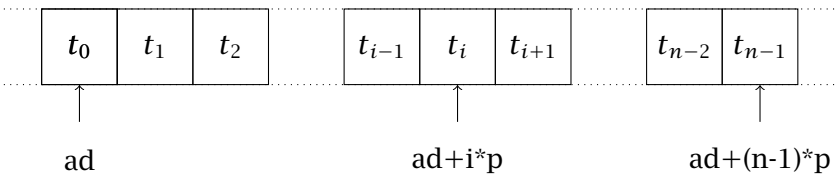
Un tuple est un assemblage de plusieurs données qui ne sont pas forcement de même type et ne sont pas modifiables.
Pour accéder aux élément on doit déconstruire le tuple

```
# let tr = (1, 1.2, 3);;
val tr : int * float * int = 1, 1.2, 3
# let (a,b,c) = tr;;
val a : int = 1
val b : float = 1.2
val c : int = 3
```

Tableaux

Un tableau ou vecteur (type `vect`) est un assemblage de données qui vérifie
↪ la longueur est fixée
↪ les données sont homogènes, elles ont toutes le même type
↪ les données sont modifiables (mutables)

Les limitations (taille et type fixés) sont les conséquences du grand avantage de ce type de données : on accède à chaque élément en temps constant (et court) indépendant de la longueur.
L'implémentation en mémoire consiste à réserver les n emplacement contigus pour les éléments d'un tableau de longueur n.
Il suffit de connaître l'adresse du premier élément, `ad`, l'élément d'indice i sera alors a l'adresse `ad + i*p` où p est la longueur de stockage d'un élément.



↪ La taille de stockage doit être constante : cela impose le type fixe.
↪ La mémoire est réservée à l'avance : cela impose la taille fixe.

1. On définit un tableau
   ↪ en l'écrivant en extension

```
# let a = [|4; 2; 6; 3|];;
a : int vect = [|4; 2; 6; 3|]
```

   ↪ en créant un tableau avec une valeur initiale

```
# let b = Array.make 100 1;;
val b : int vect =
 [|1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   ...|]
```

   ↪ On peut créer un tableau de taille n dont les éléments sont les valeurs de $f(i)$ pour i variant de 0 à $n-1$ par `Array.init n f`.
   Dans l'exemple suivant f est la fonction $x \mapsto x^2$.

```
# let c = Array.init 10 f;;
# val c : int array = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

2. On accède aux éléments par leur indice

```
# a.(1);;
- : int = 2
```

3. On peut modifier un élément, c'est une instruction qui ne renvoie rien, de résultat `none`.

```
# a.(1) <- 3;;
- : unit = ()
# a;;
- : int vect = [|4; 3; 6; 3|]
```

   On notera que le signe d'affectation est `<-`, comme pour les chaînes de caractères ; il est différent de l'affectation d'une variable mutable qui est `:=`.
4. La longueur d'un tableau est accessible par `Array.length`.
5. On peut copier un tableau avec `Array.copy`.
   L'instruction `let tab2 = tab1` ne copie pas le tableau `t1`, elle donne un autre nom à ce tableau, les deux seront modifiés en même temps.

Listes

Nous verrons au chapitre suivant un type d'assemblage de données différent ; les **listes**.

3-4 Fonctions

La définition des fonctions en **OCaml** est calquée sur la formulation mathématique usuelle.

```
# let f = function x -> x+1;;
val f : int -> int = <fun>
```

ou

```
# let f = fun x -> x+1;;
val f : int -> int = <fun>
```

OCaml sait reconnaître que la fonction est définie pour `x` entier et renvoie un entier.

On peut abréger l'écriture

```
# let f x = x + 1;;
val f : int -> int = <fun>
```

Les fonction sont des objets comme les autres :
↪ on les définit de la même manière que les variables,
↪ elles ont un type.

Pour définir une fonction avec plusieurs paramètres on peut les rassembler :

```
# let somme (x,y) = x+y;;
val somme : int * int -> int = <fun>
```

Pour **OCaml** on a affaire à une fonction qui n'a qu'un paramètre qui est un couple d'entier, c'est-à-dire un élément du produit de l'ensemble des entiers par lui-même.

Nous écriront autrement les fonctions à plusieurs variables.

```
# let somme x y = x + y;;
val somme : int -> int -> int = <fun>
```

Le type est différent : `int -> int -> int` se lit `int -> (int -> int)`.
La fonction reçoit donc un entier en paramètre et renvoie une fonction.

Cette fonction reçoit elle aussi un entier en paramètre et renverra un entier.
On peut alors n'appliquer que le premier paramètre :

```
# let plus2 = somme 2;;
val plus2 : int -> int = <fun>
# plus2 3;;
- : int = 5
```

Parfois **OCaml** ne pourra pas déterminer le type de la fonction car plusieurs types sont possibles : on dit que la fonction est **polymorphe**.

```
#let compare a b = a < b;;
val compare : 'a -> 'a -> bool = <fun>
```

3-5 Structuration des programmes

Les fonctions que nous allons écrire sont en résumé l'évaluation d'une expression dépendante des variables d'entrée. C'est la valeur calculée qui sera le retour de la fonction : il n'y a pas d'instruction `return`
Cette expression pourra être précédée d'un ensemble de définitions locales qui créent les variables ou les fonctions utilisées dans l'expression finale.
Il est recommandé d'indenter le texte par rapport à l'en-tête.

1. **Suites d'instructions**
   On peut faire précéder l'évaluation finale d'autres instructions mais celles-ci ne peuvent pas produire (renvoyer) de résultat ; ce sont des évaluation qui renvoient `()`, la valeur du type `unit`. Ces instructions modifient l'environnement ou dit qu'elles ont *un effet de bord*.
   Ce pourra être des instructions d'impression ou de modification de variables mutables : variables référencées, valeur de tableaux par exemple.
   Elles seront séparées par ";"
   Il est fortement recommandé de faire suivre le caractère ";" par un passage à la ligne.

```
#let afficherPlus c =
  let (a,b) = c in
  print_int a;
  print_newline();
  print_int b;
  print_newline();
  a + b;;
afficher : int * int -> int = <fun>
#afficherPlus (4,2);;
4
2
- : int = 6
```

2. **Branchement conditionnel**
   Les instructions classiques de branchement sont présentes
   `if p then e`
   `if p then e else e'`
   ↪ `p` doit être une expression de résultat booléen
   ↪ si `e` (ou `e'`) est une suite d'instructions il faut les borner par `begin` …`end` ou par des parenthèses
   ↪ `e` et `e'` (ou leur dernière instruction) doivent produire des résultats de même type.
   En particulier s'il n'y a pas de traitement avec `else`, `e` ne doit rien évaluer.

```
let max a b =
  if a > b then a else b;;
```

3. **Répétitions inconditionnelles**
   La répétition d'un nombre fixé d'opérations se fait à l'aide de

```
for i = a to b do
  instruction1;
  instruction2;
  ...
  instructionp done;
```

   Chaque instruction doit avoir un résultat de type `unit` car elle sera suivie d'autres instructions ; même la dernière qui sera suivie de la première lors du passage suivant dans la boucle.
   La suite des instruction est encadrée par `do` et `done` : il n'y a pas besoin de parenthèses ni de `begin` et `end`.

```
let maxTab tab =
  let n = Array.length tab in
  let maxi = ref tab.(0) in
  for i = i to (n-1) do
    if tab.(i) > !maxi
    then maxi := tab.(i) done;
  !maxi;;
```

4. **Répétitions conditionnelles**
   La répétition d'opérations en attente de la réalisation d'une condition se fait à l'aide de

```
while p_bool do
  instruction1;
  instruction2;
  ...
  instructionp done;
```

   `p_bool` est une expression de résultat booléen.
   Chaque instruction doit avoir un résultat de type `unit`.
   Il est important de prouver que la condition va devenir fausse après un nombre fini de passages.

3-6 Un exemple

Une fonction que l'on utilise souvent est la recherche d'un élément dans un ensemble.
Nous allons ici supposer que l'ensemble est représenté par un tableau. On cherche donc une fonction `cherche x t` qui renvoie `true` ou `false` selon que x est ou non un élément du tableau `t`.

1. Le premier algorithme auquel on pense est la recherche terme-à-terme

```
let recherche x tableau =
  let n = Array.length tableau in
  let reponse = ref false in
  for i = 0 to (n-1) do
    if tableau.(i) = x then reponse := true done;
  !reponse;;
```

2. Dans le programme précédent on continue à chercher même après avoir trouvé. On peut éviter ces tests inutiles à l'aide d'une boucle **while**.

```
let recherche x tableau =
  let n = Array.length tableau in
  let reponse = ref false in
  let i = ref 0 in
  while !i < n && not !reponse do
    if tableau.(!i) = x then reponse := true;
    i:= !i + 1 done;
  !reponse;;
```

On notera que l'évaluation de **et** (&&) est paresseuse, si la première expression est fausse, la seconde n'est pas évaluée. Ici elle pourrait renvoyer une erreur si `i` prenait la valeur `n`.

4 Exercices

4-1 Traductions

Ces premiers exercices demandent d'écrire des fonctions qui ont déjà été étudiées en **Python**.

Ex. 1 **Suite récurrente**
Écrire une fonction `calculer_u n u0 f` qui calcule n-ième terme de la suite (u_p) définie par $u_0 =$ `u0` et $u_{p+1} = f(u_p)$.
`f` est une fonction `CaML` telle que `f x` calcule $f(x)$.

Ex. 2 **Suite de Collatz**
La suite de Collatz de valeur initiale a est définie par $u_0 = a$ et, pour $n \in \mathbb{N}$, $u_{n+1} = 3u_n + 1$ si u_n est impair et $u_{n+1} = u_n/2$ si u_n est pair.

1. Écrire un programme `longueurCollatz a` qui détermine le premier entier n tel que $u_n = 1$ pour la suite de Collatz de terme initial a.
2. Écrire un programme `hauteurCollatz a` qui détermine l'entier maximal atteint par la suite de Collatz de terme initial a.

Ex. 3 **Somme** Écrire une fonction `somme t` qui renvoie la somme des termes (entiers) d'un tableau.

Ex. 4 **Maximum** Écrire une fonction `maximum t` qui renvoie la valeur maximale parmi des termes (entiers) d'un tableau. On étudiera avec soin le cas d'une liste vide.

Ex. 5 **Occurrences**
Un tableau ne contient que des chiffres, des entiers compris entre 0 et 9.
Écrire une fonction qui renvoie le tableau du nombre d'apparitions de chaque chiffre.

4-2 Autres exercices

Ex. 6 **Parenthésage** Ajouter les parenthèses nécessaires pour que le code ci-dessous compile.

```
let somme x y = x + y;;
somme somme somme 2 3 4 somme 2 somme 3 4;;
```

Ex. 7 **Tableau unimodal** Écrire une fonction `unimodal t` qui reçoit un tableau d'entiers `t` et qui retourne `true` ou `false` selon que `t` représente ou non une suite unimodale c'est-à-dire croissante puis décroissante.

```
unimodal [|2; 5; 12; 16; 14; 9; 5|]
#- : bool = true
unimodal [|2; 5; 12; 16; 11; 13; 5|]
#- : bool = false
```

Ex. 8 **Composition**
Écrire une fonction `comp f g` qui reçoit deux fonctions et qui retourne la composée $f \circ g$.
Quelle est sa signature ?

Ex. 9 **Fonction mystère** Que fait le programme suivant ?

```
let itere f n =
  let g x =
    let y = ref x in
      for i = 1 to n do y := f !y done;
      !y in
  g;;
```

Quelle est sa signature ?

Ex. 10 **Nombres de Hamming**
Un nombre de Hamming est un nombre qui est un produit de facteurs 2, 3 et 5 et uniquement ceux-ci : $n = 2^p 3^q 5^r$ avec p, q et r entiers positifs.
Les 20 premiers nombres de Hamming sont
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36.

1. Écrire une fonction qui teste si un nombre est de Hamming.
2. Écrire une fonction qui détermine la liste des n premiers nombres de Hamming.
3. Déterminer le 1259-ème nombre de Hamming (le premier est 1).

4-3 Permutation suivante

On considère les permutations des n premiers entiers strictement positifs.
Les permutations sont représentées par le tableau de leurs valeurs.
Par exemple, pour $n = 3$, on obtient `[|1; 2; 3|]`, `[|1; 3; 2|]`, `[|2; 1; 3|]`, `[|2; 3; 1|]`, `[|3; 1; 2|]`, `[|3; 2; 1|]`.
On veut classer ces permutations dans l'ordre croissant.
Nous allons ici étudier un algorithme connu depuis longtemps[1].
Pour en comprendre le fonctionnement on peut considérer la permutation `[|2; 1; 6; 5; 8; 7; 4; 3|]` et en chercher le successeur.
On cherche donc une permutation qui garde le maximum de termes au début.
Les 4 derniers termes forment une suite décroissante donc on ne peut les ré-ordonner pour obtenir une suite plus petite. On va donc déterminer le successeur de `[|5; 8; 7; 4; 3|]`.
On doit augmenter le 5 et la plus petite valeur possible est 7 : en échangeant on aboutit à `[|7; 8; 5; 4; 3|]`. Il suffit alors de "retourner" les derniers termes pour trouver le successeur `[|2; 1; 6; 7; 3; 4; 5; 8|]`

L'algorithme pour déterminer l'élément suivant d'un vecteur `perm` peut s'énoncer.
1. On cherche le dernier indice j tel que `perm.(j) < perm.(j+1)`.
2. S'il n'existe pas on est arrivé au dernier terme.
3. S'il existe on a `perm.(j) < perm.(j+1) >= perm.(j+2) >= ... >= perm.(n-1)`.
   On cherche le dernier indice $k > j$ tel que `perm.(j) < perm.(k)`.
4. On échange les termes d'indices k et j dans `perm`.
5. On retourne les termes de `perm` entre $j+1$ et $n-1$.

Ex. 11 **Permutation** Écrire la fonction définie ci-dessus.

[1] La première trace écrite date du XIV-ième siècle, en Inde

Chapitre II

Récursivité

Nous allons étudier dans ce chapitre l'analogue des *et ainsi de suite* ou des pointillés qui laissent à l'interlocuteur ou au lecteur le soin de continuer lui-même ce que le locuteur ébauche. Ici cela signifie que l'ordinateur va prendre en charge la répétition des instructions ou de la structure de données.
Nous introduirons ensuite un type d'assemblage de données très bien adapté à la récursivité, les listes. Une écriture simple de l'étude des cas de liste, le pattern-matching est introduit qui permet l'écriture de programmes courts et lisibles.
Dans la fin nous verrons comment **OCaml** permet de définir simplement des types de données et de les UTILISER.

1 Fonctions récursives

1-1 Définition

Fonctions récursives

Une fonction est récursive si, dans sa définition, elle fait référence à elle-même.
En **OCaml**, une fonction récursive sera déclarée par le préfixe `rec`.

Les fonctions mathématiques définies par récurrence s'expriment naturellement avec un algorithme récursif.

Exemples

1. $n!$ est peut être défini par $n! = \prod_{k=1}^{n} k$ avec la convention qu'un produit vide vaut 1 pour le calcul de 0!. On peut alors écrire

```
let fact n =
    let f = ref 1 in
    for k = 1 to n do
        f := !f*k done;
    !f;;
```

Mais on peut aussi définir $n! = \begin{cases} 1 & \text{si n=0} \\ n.(n-1)! & \text{si n>0} \end{cases}$

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

Code II.1 Expression récursive de la factorielle

2. La recherche du maximum dans un tableau peut aussi être pensée récursivement. Le maximum d'un tableau de taille n est le plus grand élément entre le maximum des $n-1$ premiers éléments et le dernier élément. On voit qu'ici aussi on doit faire intervenir un paramètre de plus qui est le dernier indice où l'on cherche.

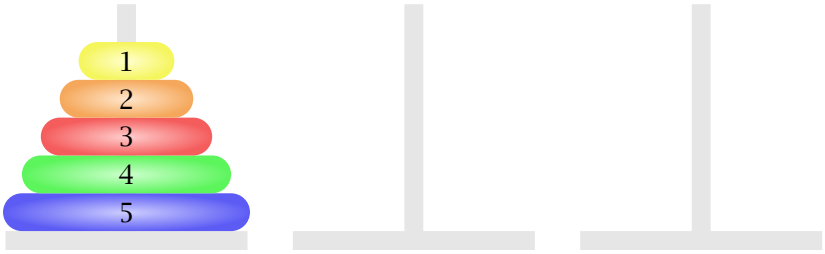
```
let maximum tableau =
    let n = Array.length tableau in
    let rec aux dernier =
       if dernier = 0
       then tableau.(0)
       else let a = aux (dernier - 1) in
              max a tableau.(dernier) in
   aux (n-1);;
```

Code II.2 Recherche récursive du maximum dans un tableau non vide

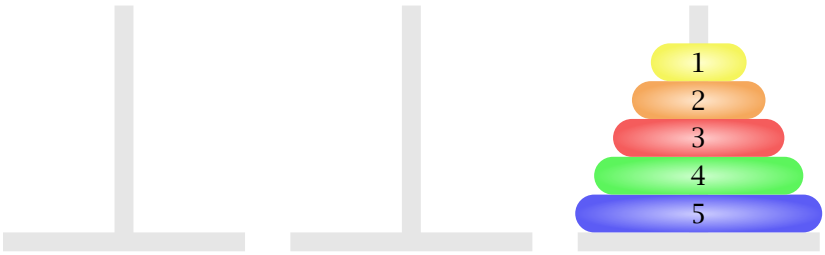
   Cependant ce dernier code est moins facile à écrire que celui qui utilise une boucle `for`. De manière générale, il vaut mieux réserver l'usage des fonctions récursives à des situations où elles apportent de la lisibilité, ce sera le cas lorsque le type de données est lui-même défini de manière récursive. Dans le cas d'un tableau il vaut mieux utiliser une boucle `for`, parfaitement adaptée à ce type de données.
3. Le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas consistant à déplacer des disques de diamètres différents d'une tour de départ à une tour d'arrivée en utilisant une tour intermédiaire tout en respectant les règles suivantes :
   ↪ on ne peut déplacer plus d'un disque à la fois,
   ↪ on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

   Au départ les disques sont placés en respectant la seconde règle sur la tour de départ.

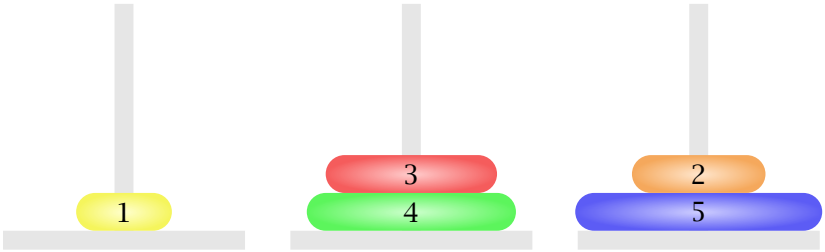
Le point de départ est



On veut arriver à



Une position intermédiaire possible serait



La résolution est en fait simple à énoncer.
Pour déplacer n disque de la tour A la tour B il suffit
↪ de ne rien faire si $n = 0$
↪ de déplacer les $n - 1$ disques supérieurs de la tour A vers la tour C,
puis de déplacer le disque restant vers la tour B
puis enfin de déplacer les $n - 1$ disques supérieurs de la tour C vers la tour B.
On voit encore apparaître un algorithme récursif.

```
let rec hanoi n tour1 tour2 tour3
    if n <> 0
    then begin hanoi (n-1) tour1 tour3 tour2;
            print_string (
                        "Déplacer le disque supérieur de "
                       ^tour1^" vers "^tour2);
          hanoi (n-1) tour3 tour2 tour1 end ;;
```

Code II.3 Résolution des tours de Hanoï

`tour1`, `tour2` et `tour3` sont des chaînes de caractères qui representent les noms donnés au 3 tours ; le premier est le nom de la tour de départ, le deuxième celui de la tour d'arrivée et le troisième nom est celui de la tour en plus
`hanoi 4 "A" "B" "C"` donne alors un mode d'emploi.

```
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de B vers A
Déplacer le disque supérieur de B vers C
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de C vers A
Déplacer le disque supérieur de B vers A
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
```

On voit ici le caractère un peu magique de la récursivité : on dit très simplement les choses et le programme produit un résultat compliqué.

1-2 Propriétés

1. Dans chacun des 3 exemples il y a eu un test pour lequel dans un des cas la fonction récursive ne s'appelait pas elle-même, on parle de **cas terminal**.
   Dans le cas des tours de Hanoï, ce cas terminal consistait à ne rien faire.
   Cette condition est indispensable car il faut que la suite des appels à la fonction stoppe.

Cas d'arrêt

Toute fonction récursive doit avoir un (ou des) **cas d'arrêt** : c'est un cas particulier des variables dont le traitement ne fait pas appel à la fonction récursive.

Un des problèmes que nous rencontrerons avec les fonctions récursives sera de s'assurer que toute variable initiale aboutit à un cas d'arrêt après un nombre fini d'appels récursifs. C'est le problème de la **terminaison** semblable à celui rencontré dans les boucles `while`. Il est très facile d'écrire des programmes récursifs qui ne terminent pas.

2. On peut suivre les appels récursifs grâce à la directive `trace` :

```
trace "fact";;
fact 3;;

fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
- : int = 24
```

3. La récursivité permet parfois d'écrire plus rapidement les algorithmes quand on veut traduire des définitions récurrentes.

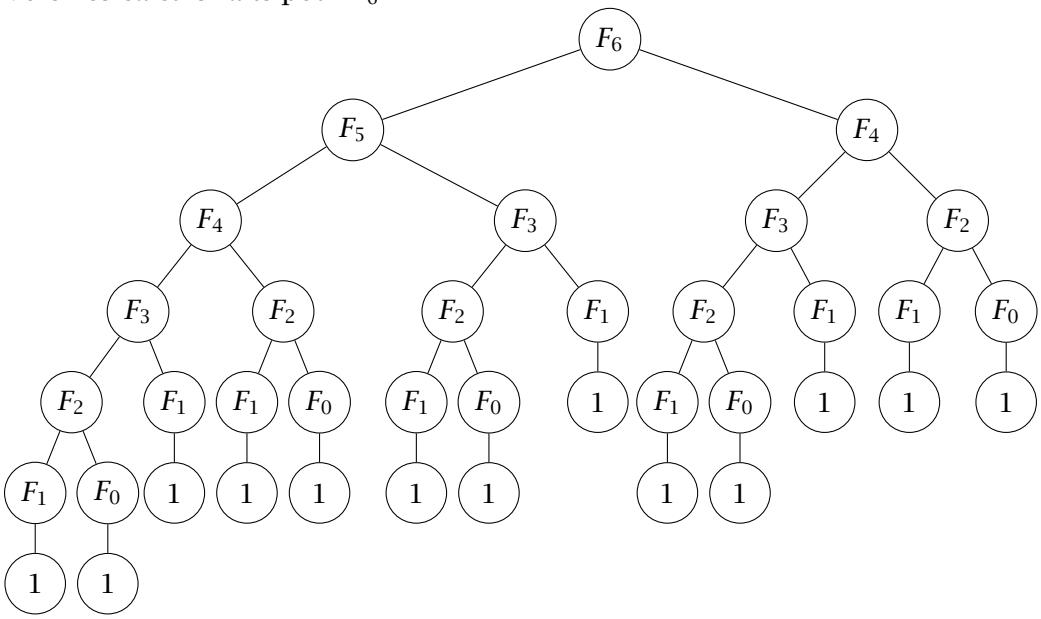
4. Les programmes récursifs sont souvent plus courts car on délègue à l'ordinateur la tâche d'écrire les boucles (on diminue aussi le nombre de variables référencées).

5. La méthode naturelle pour prouver la validité d'une fonction récursive est une démonstration par récurrence.

6. L'écriture directe de fonctions récursives peut aboutir à des temps de calcul prohibitifs. Par exemple la suite de Fibonacci définie par $F_0 = F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$ pour $n \in \mathbb{N}$ peut être implémentée sous la forme

```
let rec fibo n =
  if n <= 1
  then 1
  else fibo (n-1) + fibo (n-2);;
```

Le calcul de F_{43} demande plus d'une minute.
Voici les calculs faits pour F_6.



On remarque que le programme fait le même calcul un grand nombre de fois.

7. Le principe de la récursivité est de mettre les calculs en attente dans une pile.
Les piles ont une taille limitée et il se peut qu'elle arrive à saturation (stack overflow).

```
let rec add n =
  if n = 0 then 1
           else n + add(n-1);;
add 200000;;
#add : int -> int = <fun>
#Uncaught exception: Out_of_memory
```

Pour éviter cela on peut écrire les fonctions sous forme récursive **terminale** : un algorithme récursif est terminal si l'appel récursif n'apparaît pas comme un argument d'une autre fonction ni comme opérande d'un opérateur. Les langages évolués (dont **OCaml**) reconnaissent alors qu'ils n'ont pas besoin d'empiler les appels.
Cependant un algorithme récursif terminal sera en général moins facile à écrire car il demandera souvent d'ajouter une variable qui servira de résultat partiel ou de compteur.

```
let add_term n =
  let rec aux_add n accu =
    if n = 0  then accu
              else aux_add (n-1) (accu+n) in
  aux_add n 0;;
add_term 200000;;
#- : int = 20000100001
```

2 Listes

2-1 Tableaux

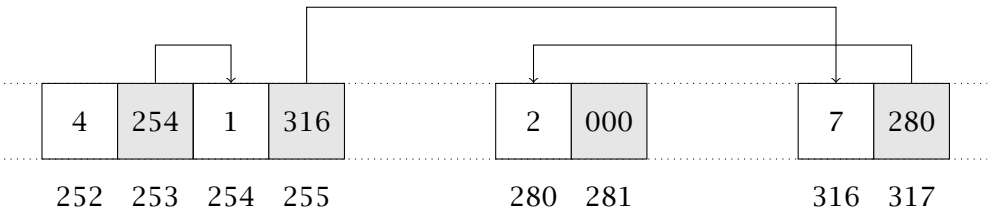
Nous avons défini les tableaux dans le chapitre précédent.
Les avantages des tableaux sont nombreux.
↪ L'encombrement en mémoire est limité : on ne conserve que la valeur stockée.
↪ L'accès aux éléments est rapide : la valeur est lue sans intermédiaire. De plus la contiguïté des valeurs permet le déplacement des valeurs suivantes dans les mémoires-cache et accélère la lecture.
↪ La modification d'une valeur est immédiate, l'ancienne valeur disparaît.

Cependant les tableaux ont aussi des inconvénients.
↪ Le premier inconvénient est le nombre d'élément qui doit être choisi à l'avance, il n'est pas possible d'ajouter un nouvel emplacement[2].
↪ La modification d'une valeur élimine la valeur précédente : il n'y a pas persistance des données.
↪ L'adressage des tableaux se fait par des entiers consécutifs : pour obtenir d'autres adressages il faut créer des structures de données plus sophistiquées, nous en verrons quelques unes dans les chapitres suivant ou en TP.

[2] Les tableaux Python permettent la modification de la taille mais cela se fait au prix d'un encombrement en mémoire et d'un temps d'accès augmentés.

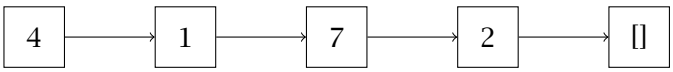
2-2 Définition

Les listes donnent une solution aux deux premiers inconvénient ci-dessus mais sont moins efficaces que les tableaux en termes d'occupation mémoire et de vitesse d'accès.
Une liste est implémentée dans un ordinateur sous la forme d'une liste chaînée : chaque valeur est couplée à une adresse qui indique où est la valeur suivante.
Par exemple une liste formée de 4, 1, 7 et 2 peut être implémentée sous la forme



L'adresse 000 ne correspond pas à une adresse physique ; elle signifie qu'il n'y a plus de terme ensuite.
La liste sera associée à l'adresse 252 ; on y lit la première valeur, 4.
Pour connaître la valeur suivante on doit lire l'adresse que l'on lit en 252 + 1, c'est 254. Le troisième terme sera trouvé à l'adresse trouvée en 254 + 1 donc en 316 …

Une représentation abstraite est possible en n'indiquant pas les adresses physiques mais en liant les valeurs.



C'est cette abstraction qui sera utilisée dans les langages de haut niveau.
On peut remarquer que ce que l'on sait de la liste est la première valeur et le reste de la liste qui est aussi une liste (éventuellement vide).

Listes Caml

Une α-liste est
↪ soit la liste vide, notée `[]`,
↪ soit l'assemblage, noté `t::q`, d'un élément t, de type α, et d'une α-liste q. t est la **tête** de la liste, q en est la **queue**.

La liste ci dessus peut donc être définie par `let liste = 4::1::7::2::[];;`.
Elle sera affichée sous la forme `[4; 1; 7; 2]`.
On pouvait la définir directement sous cette forme `let liste = [4; 1; 7; 2];;`.

Il faut noter que les éléments d'une liste ont un type qui doit être constant : une liste est un assemblage homogène.

2-3 Utilisation

1. **Déconstruction**
   Pour voir les termes d'une liste il faut déconstruire l'assemblage.
   La tête et la queue d'une liste sont obtenus par les fonctions `List.hd` (pour ***head***) et `List.tl` (pour ***tail***). Ces fonctions renvoient une erreur si la liste est vide.
   Comme la construction d'une liste est récursive le moyen naturel pour les utiliser est une fonction récursive. Le cas terminal sera souvent le cas d'une liste vide.
   Par exemple la somme des termes d'une liste d'entiers est calculée par

```
let rec somme liste =
  if liste = []
  then 0
  else (hd liste) + (somme (tl liste));;
```

2. **Pattern-matching**
   Caml permet une construction simplifiée du test ci-dessus : le **pattern-matching**.
   Les listes ont deux structures possibles, vide ou un assemblage, on parle de **motifs** (patterns en anglais).
   On peut filtrer les différents motifs d'une variable par la construction

```
...
  match variable with
   motif1 -> instructions1
  |motif2 -> instructions2
  |motif3 -> instructions3
  ...
```

   La barre verticale symbolise un "ou". Il est possible (et peut être plus lisible) de mettre ce symbole aussi devant le premier motif.
   Dans le cas d'une liste les motifs seront souvent `[]` et `t::q` mais d'autres motifs sont possibles : `a::b::q`, `0::q`, ...

```
let rec somme liste =
  match liste with
  |[] -> 0
  |t::q -> t + (somme q);;
```

Code II.4 Somme des termes d'une liste

↪ Un motif peut contenir une valeur constante mais pas la valeur d'une variable.
↪ Quand on veut filtrer en utilisant une variable on peut écrire une condition `if` mais on peut aussi ajouter une condition après le motif avec le mot-clé `when`. On doit reprendre le même motif ensuite pour gérer les cas où la condition n'est pas vérifiée.

```
match variable with
|motif1 -> instructions1
|motif2 when condition -> instructions2
|motif2 -> instructions3
...
```

↪ Les filtres sont appliqués du haut vers le bas, la fonction est appliquée au premier filtre satisfait.
↪ Les cas restant peuvent être filtrés par le motif générique : _.

2-4 Fonctions usuelles

Dans cette section nous allons écrire des fonctions qui existent dans le module `List`. Il est plus important de savoir les écrire que de se souvenir de leur nom.

1. **Tête et queue (`List.hd` et `List.hd`)**
   À l'aide du pattern-matching on peut écrire les déconstructions. On remarquera que l'on peut ne pas nommer les composantes inutilisées.

```
let tete liste =
  match liste with
  |[] -> failwith "Liste vide dans tete"
  |t::_ -> t;;
```

```
let queue liste =
  match liste with
  |[] -> failwith "Liste vide dans queue"
  |_::q -> q;;
```

2. **Longueur (`List.length`)**
   La longueur d'une liste s'écrit simplement aussi.

```
let longueur liste =
  match liste with
  |[] -> 0
  |t::q -> 1 + (longueur q);;
```

Code II.5 Longueur d'une liste

La fonction **OCaml** utilise la récursivité terminale pour pouvoir gérer les longues listes. La fonction `succ` calcule le successeur d'un entier.

```
let rec length_aux n = function
     []  -> n
  | _::l -> length_aux (succ n) l;;

let list_length l = length_aux 0 l;;
```

On remarquera aussi la définition sans nommer la variable.

3. **Retournement (`List.rev`)**
   Pour retourner une liste on place les élément de la liste en les lisant de la gauche vers la droite) dans une liste, au départ vide, ils seront donc écrits de la droite vers la gauche. On écrit naturellement une fonction récursive terminale.

```
let retourne liste =
  let rec aux aFaire fait =
    match aFaire with
    |[] -> fait
    |t::q -> aux q (t::fait) in
  aux liste [];;
```

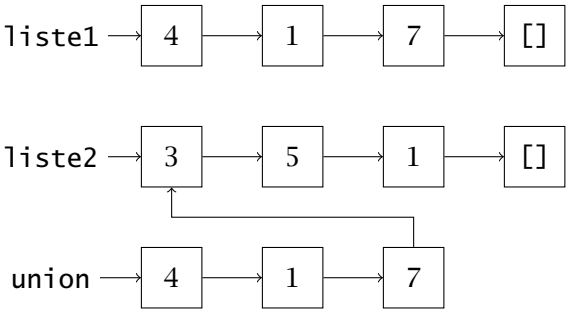
4. **Appartenance (`List.mem`)**

```
let rec appartient a liste =
  match liste with
  |[] -> false
  |t::q when t = a -> true
  |t::q -> appartient a q;;
```

5. **Concaténation de listes (`List.append`)**
   Pour concaténer deux listes on va ajouter les éléments de la première à la seconde.

```
let rec union liste1 liste2 =
  match liste1 with
  |[] -> liste2
  |t::q -> t::(union q liste2);;
```

Code II.6 Concaténation de deux listes



Le nombre d'opérations d'assemblage (`::`) est égal au nombre d'éléments de la première liste ; tous ces éléments sont copiés. Par contre les éléments de la seconde liste restent tels quels, ils sont communs à `liste2` et à `union liste1 liste2`.
La concaténation est aussi définie en **OCaml** par l'opérateur `@` : `liste1 @ liste2`.

3 Création de types

Caml permet de créer facilement des types nouveaux.
C'est un outil qui augmente la lisibilité des programmes en permet un contrôle fin des variables. En particulier le pattern matching, dans le cas des types sommes, permet une structuration plus simple des études des différents cas.

3-1 Types produits

Un type produit (ou enregistrement) est un ensemble de rubriques nommées (ou champs) qui ont chacune un type fixé.

```
type personne = {nom : string; age : int};;
#Type personne defined.
```

↪ Les différents champs peuvent ne pas être du même type.

Les champs ne sont par ordonnés, contrairement au couples et tuples.

↪ On définit un enregistrement en renseignant chacune des champs :

```
let ed = {age = 60; nom = "Détrez"};;
```

↪ On atteint la valeur du champs `label` de l'enregistrement `nom` par `nom.label`.

```
ed.nom;;
#- : string = "Détrez"
```

↪ Si on fait précéder le type d'un champs du mot clé `mutable` alors la valeur de la rubrique est modifiable. On modifie un champs par `nom.label <- nouveau`. La modification d'un champs mutable est une instruction de résultat `unit` sans retour de valeur. C'est un effet de bord.

```
type etudiant = {nom : string;
                 mutable classe : string};;
let ed = {nom = "Detrez"; classe = "HX4"};;
ed.classe <- "M'1";;
ed;;
#- : etudiant = {nom = "Detrez"; classe = " M'1"}
```

↪ Le comportement des enregistrements ressemble fortement à celui des tableaux, en particulier la modification d'un champs mutable fait perdre l'ancienne valeur.

↪ La définition d'une variable référencée (avec le mot-clé `ref`) crée en fait un enregistrement mutable dont le seul champs est `content`.

↪ On peut aussi définir soi-même un type d'entier mutable.

```
type intRef = {mutable valeur : int};;
# type intRef = { mutable valeur : int; }

let n = {valeur = 1};; (* remplace let n = ref 1;; *)
# val n : intRef = {valeur = 1}

let incremente n = n.valeur <- n.valeur + 1;;
# val incremente : intRef -> unit = <fun>
```

3-2 Types sommes

Un type somme est formé d'une énumération de cas possibles pour une valeur de ce type, chaque cas comporte un nom de cas, le **constructeur**, et un (éventuel) type associée.

Les cas sont séparés par |.
Il faut écrire en majuscule la première lettre du nom d'un constructeur de valeur.

```
type couleur = Pique | Coeur | Carreau | Trefle;;
type carteTarot = |Roi of couleur
                  |Dame of couleur
                  |Cavalier of couleur
                  |Valet of couleur
                  |Numero of int*couleur
                  |Atout of int
                  |Excuse;;
```

Lorsque l'on veut définir une fonction sur un type somme il faut en général donner un comportement distinct selon les différents cas.
On peut, comme dans le cas des listes, utiliser le **pattern matching**.

Un type peut être défini de manière récursive.
Par exemple on peut définir les listes avec

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

Pour gérer la possibilité de l'absence de réponse il existe le type **optionnel** :

```
type 'a opt = None |Some of 'a;;
```

4 Exercices

Ex. 1 **Suite**
Écrire une fonction **récursive** `suite u0 n` qui calcule u_n en fonction u_0 avec $u_{n+1} = \frac{2u_n+1}{3+u_n}$. On notera que les termes sont des réels.

Ex. 2 **Puissance**
Il n'y a pas de fonction puissance en Caml. Écrire une fonction **récursive** `puissance a n` qui calcule a^n pour a entier et n entier positif.

Ex. 3 Fonction d'Ackermann

La fonction d'Ackermann, A, est définie de $\mathbb{N} \times \mathbb{N}$ dans $\mathbb{N}$ par

$$\begin{cases} \forall n \in \mathbb{N} & A(0,n) = n+1 \\ \forall m \in \mathbb{N}^* & A(m,0) = A(m-1,1) \\ \forall (m,n) \in \mathbb{N}^* \times \mathbb{N}^* & A(m,n) = A(m-1, A(m,n-1)) \end{cases}$$

Écrire une fonction `ackermann m n` qui renvoie $A(m,n)$.

Ex. 4 Coefficients binomiaux

Écrire une fonction récursive `binomial n p` qui renvoie $\binom{n}{p}$ pour $0 \leqslant p \leqslant n$.
On rappelle que, pour $1 \leqslant p \leqslant n$, $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$ et on prendra soin de n'effectuer que des opérations exactes sur les entiers.

Ex. 5 PGCD

1. En remarquant que le PGCD de a et b avec $b \neq 0$ est égal à celui de b et c où c est le reste de la division de a par b (`a mod b`) écrire une fonction **récursive** `pgcd a b` qui calcule le PGCD de a et de b.
2. Montrer que si on a $0 < b < a$ et si `pgcd a b` effectue $p \geqslant 1$ appels récursifs alors on a $b \geqslant F_p$ et $a \geqslant F_{p+1}$ où (F_n) est la suite de Fibonacci ($F_0 = F_1 = 1$).
3. En déduire que si on a $0 \leqslant a \leqslant F_n$ et $0 \leqslant b \leqslant F_n$ avec $n \geqslant 1$ alors `pgcd a b` effectue au plus n appels à la fonction `mod`.

Ex. 6 Nombres de Catalan

Les nombres de Catalan sont définis par $C_0 = 1$ et $C_{n+1} = \sum_{k=0}^{n} C_k.C_{n-k}$.

1. Écrire une fonction récursive qui calcule le nombre de Catalan d'indice n.
2. Pourquoi le calcul de C_{17} demande-t-il plusieurs secondes ?
3. Écrire une fonction plus rapide qui calcule le nombre de Catalan d'indice n : on pourra utiliser un tableau.

Ex. 7 Maximum

Écrire une fonction `maximum liste` qui calcule le maximum d'une liste d'entiers ou de flottants. La fonction devra lever une erreur si la liste est vide.

Ex. 8 Transformation d'une liste

Écrire une fonction `carre liste` qui calcule une liste dont les termes sont les carrés des éléments de la liste passée en argument.
De manière générale écrire une fonction `applique f liste` qui calcule une liste dont les termes sont les images par la fonction f des éléments de `liste`.
`applique : ('a -> 'b) -> 'a list -> 'b list = <fun>`

Cette dernière construction est très souvent utilisée.
Caml contient une fonction déjà écrite : `map`.
On n'est pas obligé d'écrire la fonction à appliquer on peut utiliser directement une fonction non nommée (anonyme). Par exemple la fonction `carre` pouvait s'écrire

```
let carre  = map (fun x-> x*x);;
```

Comme on n'a pas écrit le dernier argument de `map` on obtient bien une fonction qui doit recevoir une liste (d'entiers) et qui renvoie une liste.

Ex. 9 Séparation du maximum

Écrire une fonction `selectionMax` telle que `selectionMax liste`, appliquée à une liste d'entiers distincts, renvoie un couple formé de l'élément maximum de la liste et de la liste des éléments restants (l'ordre des éléments peut changer).
`selectionMax [4]` devra renvoyer `4, []`,
`selectionMax [4; 7; 1; 5]` pourra renvoyer `7, [4; 5; 1]`,
`selectionMax []` devra lever une erreur par l'instruction `failwith "liste vide"`.

Ex. 10 Retournement

On peut utiliser la concaténation de listes pour écrire le retournement d'une liste sous la forme.

```
let rec renverse liste =
  match liste with
  |[] -> []
  |t::q -> (renverse q)@[t];;
```

On notera qu'il n'est pas possible d'utiliser la construction `::` pour ajouter un élément "à droite" d'une liste.
En comparant le nombre d'opération d'assemblage (`::`) justifier pourquoi la fonction donnée dans la chapitre est préférable.

Ex. 11 Filtrage

1. Écrire une fonction `positifs liste` qui renvoie la listes des termes positifs appartenant à la liste donnée en paramètre.
2. Plus généralement, écrire une fonction `filtrer f liste` qui renvoie les termes `x` de la liste donnée en paramètre tels que `f x` est évalué en `true`.
   Dans la question ci-dessus, la fonction était `let positif x = x >= 0;;`
   et on pourrait écrire `let positifs = filtrer positif;;`
3. Quelle est la signature de `filtrer` ?

La fonction existe sous le nom `List.filter`.

Ex. 12 Condition sur une liste

Comme ci-dessus on se donne une fonction à résultat booléen, `f`.

1. Écrire une fonction `verifie f liste` qui renvoie le premier élément de la liste vérifiant `f`, sous la forme du type optionnel `Some x`.
   S'il n'existe aucun élément vérifiant la condition la fonction doit renvoyer `None`.
   La fonction existe sous le nom `List.find_opt`.
2. Écrire une fonction `il_existe f liste` qui renvoie `true` si et seulement si au moins un élément de la liste vérifie `f`.
   La fonction existe sous le nom `List.exists`.
3. Écrire une fonction `pour_tous f liste` qui renvoie `true` si et seulement si tous les éléments de la liste vérifient `f`.
   La fonction existe sous le nom `List.for_all`.

Ex. 13 Séparation

Écrire une fonction `prendre n liste` renvoie deux listes : la première est formée par les n premiers éléments de la liste, dans le même ordre, la seconde formée par les éléments restants. Si la liste comporte moins de n éléments le résultat sera formé de la liste et d'une liste vide.

Ex. 14 Entrelacement

Écrire une fonction `shuffle` qui reçoit deux listes en paramètres et qui renvoie une liste formée des éléments des deux listes entrelacés en finissant pas les élément restants de la liste plus longue quand les deux listes n'ont pas le même nombre d'éléments.
`shuffle [13; 4; 11; 8] [9; 7; 13; 5; 12; 2]` devra renvoyer
`[13; 9; 4; 7; 11; 13; 8; 5; 12; 2]`.

Ex. 15 Parties

On considère une liste de taille n comme un ensemble à n éléments même si des éléments peuvent être égaux.
Écrire une fonction `parties` qui renvoie une liste des sous-ensembles associées.
Par exemple `parties [1; 2; 1]` pourra renvoyer
`[[]; [1]; [2]; [2; 1]; [1]; [1; 1]; [1; 2]; [1; 2; 1]]`.

Ex. 16 Logique floue

On veut travailler avec une logique laissant la place à l'indétermination.
Une proposition peut donc être vraie, fausse ou indéterminée.
Définir un type correspondant à ces possibilités et écrire les fonctions `ou`, `et` et `non` correspondantes.

Ex. 17 Calcul avec l'infini

Définir un type somme composé des entiers et d'un objet supplémentaire représentant l'infini. Écrire les fonction d'addition et de multiplication pour ce type.

Ex. 18 Rationnels

Définir un type produit permettant de représenter les fractions rationnelles.
Écrire les fonction de simplification, d'addition et de multiplication.

4-1 Crible d'Érathostène

Le crible d'Érathostène permet de calculer les nombres premiers entre 2 et n.
Son principe est le suivant.

1. On écrit, dans l'ordre croissant, la suite des entiers de 2 à n.
2. On enlève de la liste tous les multiples entiers (à partir de $2k$) de chaque terme k (non enlevé) que l'on lit dans la liste.
3. Il ne reste alors que les nombres premiers.

Par exemple, pour $n = 20$, on calcule successivement
↪ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
↪ <u>2</u>, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19, ~~20~~
↪ <u>2</u>, <u>3</u>, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
↪ <u>2</u>, <u>3</u>, ~~4~~, <u>5</u>, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
↪ <u>2</u>, <u>3</u>, ~~4~~, <u>5</u>, ~~6~~, <u>7</u>, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
↪ <u>2</u>, <u>3</u>, ~~4~~, <u>5</u>, ~~6~~, <u>7</u>, ~~8~~, ~~9~~, ~~10~~, <u>11</u>, ~~12~~, 13, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
↪ <u>2</u>, <u>3</u>, ~~4~~, <u>5</u>, ~~6~~, <u>7</u>, ~~8~~, ~~9~~, ~~10~~, <u>11</u>, ~~12~~, <u>13</u>, ~~14~~, ~~15~~, ~~16~~, 17, ~~18~~, 19, ~~20~~
↪ <u>2</u>, <u>3</u>, ~~4~~, <u>5</u>, ~~6~~, <u>7</u>, ~~8~~, ~~9~~, ~~10~~, <u>11</u>, ~~12~~, <u>13</u>, ~~14~~, ~~15~~, ~~16~~, <u>17</u>, ~~18~~, 19, ~~20~~
↪ <u>2</u>, <u>3</u>, ~~4~~, <u>5</u>, ~~6~~, <u>7</u>, ~~8~~, ~~9~~, ~~10~~, <u>11</u>, ~~12~~, <u>13</u>, ~~14~~, ~~15~~, ~~16~~, <u>17</u>, ~~18~~, <u>19</u>, ~~20~~

Les nombres premiers inférieurs à 20 sont 2, 3, 5, 7, 11, 13, 17, 19.

Ex. 19 **Énumération** Écrire une fonction `enum a b` qui renvoie la liste des entiers consécutifs de `a` à `b` dans l'ordre croissant, bornes comprises.

Ex. 20 **Enlever les multiples** Écrire une fonction `oterMult k liste` qui renvoie la liste des entiers de `liste` qui ne sont pas multiples de `k` dans le même ordre que dans la liste initiale.

Ex. 21 **Crible** Déduire des questions précédentes une fonction `premiers n` qui retourne la liste des nombres premiers inférieurs à n.
Combien d'opérations effectue-t-elle ?

Dans l'exemple on voit qu'il n'y a plus de termes à éliminer à partir de 5.
Ce phénomène est général : si $p^2 \geqslant n$, tous les multiples de p ont été retirés.

Ex. 22 **Amélioration** En déduire une modification de `premiers n`.
Combien d'opérations effectue-t-elle ?

4-2 Codes de Gray

Un code de Gray d'ordre n est une façon d'énumérer les entiers de 0 à 2^n-1 tel que le passage de l'écriture binaire du k-ème à celle du $(k+1)$-ème nombre se fait en ne changeant qu'un seul bit.

Ex. 23 Déterminer un code de Gray d'ordre 3.

On suppose que Γ est un code de Gray d'ordre n considéré comme une liste ℓ de 2^n chaînes de caractères de même longueur n dont les lettres sont 0 ou 1.

Ex. 24 Montrez que la liste obtenue en concaténant les mots de ℓ précédés de 0 puis les mots de la liste inverse de ℓ précédés de 1 est un code de Gray d'ordre $n+1$.

Ex. 25 En déduire une fonction construisant un code de Gray d'ordre n.

Les codes de Gray obtenus avec la construction qui vient d'être décrite sont dits réfléchis : on les note Γ_n. $g_n(k)$ est l'entier représenté par le terme à la position k de Γ_n pour $0 \leqslant k < 2^n$.

Ex. 26 Déterminer $g_n(0)$, $g_n(2^n-1)$, $g_n(2^{n-1}-1)$ et $g_n(2^{n-1})$.
Comparer $g_n(k)$ et 2^{n-1} pour $k < 2^{n-1}$ et $k \geqslant 2^{n-1}$.

Ex. 27 Comment passer du k-ième terme de Γ_n au $(k+1)$-ième ?

4-3 Division egyptienne

Pour effectuer la division de a par b la méthode utilisée dans l'Egypte antique consiste à calculer les termes b, $2b$, $4b$ ...jusqu'au dernier terme inférieur à a. On soustrait ensuite les termes de la forme $2^p b$ à partir du plus grand quand cela est possible : le terme restant est le reste. Le quotient est obtenu en ajoutant les puissance de 2 correspondant aux produits soustraits.
Exemple : pour diviser 2874 par 53 on calcule les valeurs

2^p	1	2	4	8	16	32
53.2^p	53	106	212	424	848	1696

$2874 - 1696 = 1178$, $1178 - 848 = 330$, on ne soustrait pas 424, $330 - 212 = 118$ puis $118 - 106 = 12$: le reste est 12. Le quotient est $32 + 16 + 4 + 2 = 54$.

Ex. 28 Déterminer la liste (décroissante) des $2^p.b$ inférieurs à a.

Ex. 29 Écrire une fonction `reste a b` qui calcule le reste de la division de a par b en utilisant cette méthode.

Ex. 30 Quel est le nombre de multiplications et de soustractions effectué ?

4-4 Sous-listes

Une sous-liste de la liste $m = [x_0; \ldots; x_{n-1}]$ est une liste obtenue en supprimant certains éléments de m (et en conservant l'ordre), c'est-à-dire une liste (éventuellement vide) de la forme $[x_{\sigma(0)}; \ldots; x_{\sigma(k-1)}]$ avec $0 \leq \sigma(0) < \cdots < \sigma(k-1) \leqslant n-1$.
Par exemple, $[0;1;2;2;3;3]$ est une sous-liste de $[4;0;1;2;2;2;2;3;4;3]$.

Ex. 31 Écrire une fonction `est_sous_liste` qui prend pour argument deux listes m_1 et m_2 et teste si m_1 est une sous-liste de m_2.

Étant donnée une liste m, on note $\Sigma(m)$ l'ensemble des sous-listes de la liste m.

Ex. 32 Soit `m = t::q` une liste non vide. Montrer que $\Sigma(m)$ est l'union de $\Sigma(q)$ et de l'ensemble des listes `t::r` où `r` décrit $\Sigma(q)$.

Ex. 33 En déduire une fonction `sous_listes` qui prend pour argument une liste m et calcul l'ensemble $\Sigma(m)$ (cet ensemble sera retourné sous forme d'une liste de listes, la même sous-liste pouvant éventuellement être répétée plusieurs fois).

4-5 Décomposition de Fibonacci

(F_n) est la suite de Fibonacci définie par $F_0 = 1$, $F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$ pour $n \geqslant 0$.
Pour k et m entiers, on note $k \gg m$ quand $k \geqslant m + 2$.
Une décomposition de $n \in \mathbb{N}$ dans la base de Fibonacci est la donnée de $r \in \mathbb{N}$ et de $(r + 1)$ nombres de Fibonacci $(F_{i_r}, \ldots, F_{i_1}, F_{i_0})$ tels que
$n = \sum_{k=0}^{r} F_{i_k}$ avec $i_r \gg i_{r-1} \gg \cdots \gg i_1 \gg i_0 \geqslant 1$.

Ex. 34 Prouver que si $(F_{i_r}, \ldots, F_{i_1}, F_{i_0})$ est une décomposition de n alors $F_{i_r} \leqslant n < F_{i_r+1}$.

Ex. 35 Montrer que, pour tout $n \in \mathbb{N}^*$, il existe une unique décomposition de Fibonacci de n.

Ex. 36 Déterminer cette décomposition pour $n = 67$.

Ex. 37 Écrire une fonction `listeFibo : int -> int * int list` qui prend en argument un entier $n \geqslant 1$ et qui renvoie un couple formé de l'indice p et de la liste $(F_p, F_{p-1}, \ldots, F_1, F_0)$ avec p tel que $F_p \leqslant n < F_{p+1}$.

Ex. 38 Écrire une fonction `decompose : int -> int list` qui renvoie l'unique décomposition de Fibonacci de n.

Ex. 39 Écrire une fonction `somme1 : int -> int list -> int list` qui ajoute un nombre de Fibonacci F_p représenté par $p \geqslant 1$ a une décomposition de Fibonacci. Le résultat doit être sous la forme d'une décomposition de Fibonacci ne doit pas calculer l'entier associé à la liste.

Ex. 40 Écrire une fonction `somme : int list -> int list -> int list` qui calcule la somme de deux entiers représentés par leur décomposition de Fibonacci. Le résultat doit être sous la forme d'une décomposition de Fibonacci et on ne passera pas par les entiers

Chapitre III

Analyse des algorithmes

Nous allons étudier dans ce chapitre les méthodes qui vont permettre d'analyser les programmes écrits et de les valider avant même leur exécution par un ordinateur.
Nous introduirons des algorithmes élémentaires pour trier des collections de valeurs et en donnerons leur analyse.

1 Analyse des algorithmes

Lorsque l'on écrit un programme il est nécessaire que celui-ci soit correct.
Le premier outil qui vient à l'esprit est de le tester avec des entrées pour lesquelles on sait, à l'avance, le résultat attendu et de vérifier qu'il est bien obtenu. Cependant cela comporte des incertitudes.

↪ Si le programme ne passe pas les tests, on sait que quelque chose n'a pas fonctionné mais il est difficile de savoir ce pose problème. Est-ce l'algorithme, le système, le compilateur du langage …?
↪ Si le programme passe les tests on ne sait qu'il fonctionne que dans les cas où on n'a pas vraiment besoin de lui (on a déjà la réponse). Qu'en est-il des nombreux cas non testés ?

Il existe un moyen plus sûr qui ne concerne que l'algorithme : c'est une analyse a-priori de celui-ci. Cette analyse est un travail théorique antérieur à l'exécution sur une machine. Cette démarche est très exigeante mais elle permet d'éviter de nombreux problèmes. C'est cette étude que nous allons ébaucher.
Nous allons analyser un programme en trois étapes.

1. Le programme fournit-il un résultat ? C'est le problème de la **terminaison**
2. Le programme fournit-il le bon résultat ? On doit en donner la **preuve**.
3. Le programme fournit-il le bon résultat dans un temps raisonnable ?
   On en étudie la **complexité**.

Ces trois question sont de plus en plus précises et seront souvent étudiées dans cet ordre. Cependant il y aura souvent des ressemblances entre la première et la troisième question : on pourrait reformuler le problème de la terminaison sous la forme "*Le programme fournit-il un résultat dans un temps fini ?*". Un temps fini peut être considéré comme la définition la plus élémentaire d'un temps raisonnable.

Dans cette étude on va appliquer les méthodes aux fonctions suivantes.

A.

```
let somme_tableau tab =
  let n = Array.length tab in
  let som = ref 0 in
  for i = 0 to (n-1) do som := !som + tab.(i) done;
  !som;;
```

B.

```
let estPuissance2 n =
  let p = ref n in
  while !p mod 2 = 0 do p := !p/2 done;
  !p = 1;;
```

C.

```
let rec appartient x liste =
  match liste with
  |[] -> false
  |t::q when t = x -> true
  |t::q -> appartient x q;;
```

1-1 Terminaison

Le programme fournit-il un résultat ? C'est-à-dire finit-il ?
On emploie aussi le verbe **terminer** sous une forme intransitive et on parle de terminaison.
Il y a des limites à cette interrogation.

↪ On peut montrer qu'il n'existe pas d'algorithme permettant de répondre à cette question. En termes techniques le problème de l'arrêt est indécidable.

↪ Certains programmes n'ont pas besoin de terminer : on peut penser à certains jeux, à la console python, à l'interface windows, mac ou linux

Une réponse simple serait "*on le lance et on voit bien s'il s'arrête*"
Bien que facile à mettre en œuvre cette méthode n'est pas satisfaisante.

↪ Il se peut que le programme boucle indéfiniment pour certaines valeurs, pas toutes. Un test positif ne prouve pas que le programme est correct.

↪ Il arrive aussi que le temps de calcul soit plus long que le temps d'attente supportable. Un test négatif ne prouve pas que le programme est incorrect.

Il faut donc essayer de prouver la terminaison par une étude a-priori.

Un programme est composé d'une suite d'instructions ; elles sont séparées par un point-virgule dans **OCaml**.
Une instruction peut être

1. une instruction simple faisant appel à des fonctions de bases ou à des fonctions écrites par l'utilisateur,

2. une instruction conditionnelle qui fait appel

   a. à une évaluation simple à résultat booléen
   b. une ou deux suites d'instructions selon qu'il y a ou non une clause `else`

3. une suite d'instructions répétée par une boucle `for`,

4. une suite d'instructions répétée par une boucle `while`.

Les instructions de base terminent si les fonctions qu'elles appellent terminent.
La suite d'instructions d'une boucle `for` est répétée une nombre de fois prévu à l'avance ; la boucle termine à condition que les instructions du corps terminent.
On remarquera que cette dernière propriété est mise en défaut si l'indice de la boucle est modifié dans les évaluations : **c'est une écriture à proscrire absolument !**

Dans un algorithme on doit donc prouver la terminaison des fonctions appelées, ce qui demande une étude particulière dans le cas des fonctions récursives, et celle des boucles `while`.

On peut remarquer que la preuve de terminaison peut être impossible.
La suite de Syracuse de terme initial a est la suite définie par

$$u_0 = a \quad u_{n+1} = \begin{cases} u_n/2 & \text{si } n \text{ est pair} \\ 3u_n + 1 & \text{si } n \text{ est pair} \end{cases} \qquad \text{pour tout } n \geqslant 1$$

```
let rec syracuse a =
  if a = 1
  then 0
  else if a mod 2 = 0
       then 1 + (syracuse a/2)
       else 1 + (syracuse (3*a+1));;
```

On ne sait pas prouver que ce programme termine pour tout entier a ; ce serait un résultat mathématique (espéré) si on arrivait à le prouver.

Terminaison des boucles `while`

Pour prouver que les tests de sortie d'une boucle `while` finissent par être réalisés on peut utiliser un résultat mathématique simple :
toute suite strictement décroissante d'entiers tend vers moins l'infini
sous la forme :
toute toute suite strictement décroissante d'entiers positifs est finie.
Cela induit la définition suivante

Variant de boucle

Un variant de boucle est une valeur entière dépendant des variables du programme
↪ qui est positive quand la condition de la boucle est réalisée
↪ et qui décroît strictement à chaque passage dans la boucle.

Si une boucle admet un variant alors elle termine.

En effet s'il existe un variant de boucle alors il ne peut exister qu'un nombre fini de répétitions[3] donc la boucle termine.

Terminaison des fonctions récursives

Pour prouver la terminaison d'une fonction récursive on emploie la notion d'un variant en introduisant une mesure entière de la taille des paramètres. On prouve alors que la fonction termine par récurrence sur cette mesure.
La mesure d'une donnée peut être :
↪ la valeur de n (ou le nombre de bit dans sa représentation en base 2) dans le cas d'une variable entière,
↪ le nombre d'éléments des listes, tableaux, fichiers,
↪ la longueur d'un mot,
↪ le degré d'un polynômes,
↪ le nombre de lignes (ou de colonnes ou leur produit) dans le cas d'une matrice,
↪ …

[3] On retrouve le fait que les boucles `for i = a to b` terminent toujours car $b - i$ est un variant de boucle, **si i n'est pas modifié dans la boucle**.

Terminaison des exemples

A. `somme_tableau` fait appel à des fonctions simples, on admet qu'elles terminent, et à une boucle `for`. Ainsi le programme termine.

B. Dans le cas de la fonction `estPuissance2` la valeur de `!p` est un variant.

C. Dans la fonction `appartient`, on note n la longueur de la liste.

1. Pour $n = 0$, le programme termine directement.
2. On suppose que la fonction termine quand on l'appelle avec une liste de taille n.
   On considère une liste de taille $n + 1$: son premier terme est t.
   Si t est la valeur recherchée, le programme termine.
   Sinon on appelle la fonction avec le reste de la liste, de taille n, donc termine.
3. Ainsi la fonction termine dans tous les cas.

1-2 Preuve

Le programme fournit-il le bon résultat ?
Il existe des outils théoriques (la logique de Hoare) qui permettent une formalisation de l'analyse de ce problème. C'est une étape au delà de ce cours d'introduction mais elle est indispensable quand on veut prouver des programmes complexes.

Dans le cadre de ce cours nous allons donner des outils plus simples. Dans le cas d'une suite d'instructions il faut prouver que chaque instruction fait ce qui est attendu. Dans le cas d'instructions simples, il suffit de vérifier la concordance de l'écriture avec le projet.

Invariants des boucles `for`

Dans la fonction `somme_tableau` on voit facilement que la valeur de `som` au début de la boucle pour l'indice i est la somme des termes du tableau pour les indices 0 à $i - 1$.

1. Pour $i = 0$ la somme est vide, on lui associé la valeur 0.
2. Si on suppose que la propriété est vraie au début d'un passage de boucle pour i on ajoute `tab.(i)` à `som` donc la propriété est vraie au passage suivant, quand l'indice vaut $i + 1$.
3. Lors du passage de la dernière boucle la propriété est vraie en sortie pour $i = n$ ce qui prouve que la fonction renvoie la somme des termes.

On a donné un **invariant** pour la boucle.

Invariant de boucle

Un invariant d'une boucle `for i = a to b` est une propriété $P(i)$ dépendant des variables et de l'indice i de la boucle telle que
- ↪ $P(a)$ est vraie avant le premier passage de la boucle,
- ↪ si $P(i)$ est vraie avec $i \leqslant b$ alors $P(i+1)$ est vérifiée après l'exécution des instructions de la boucle pour l'indice i,
- ↪ $P(b+1)$ prouve le résultat attendu.

Si un boucle admet un invariant alors elle est prouvée.

On prouve par récurrence que $P(i)$ est vraie pour tout i entre a et $b+1$ d'où la preuve du résultat attendu.

Invariants des boucles `while`

On peut généraliser la définition pour les boucles `while`.

Invariant de boucle

Un invariant d'une boucle est une propriété P dépendant des variables telle que
- ↪ P est vraie à l'initialisation
- ↪ si P est vérifiée au début de la suite des instructions de la boucle alors elle est vraie au début de l'itération suivante
- ↪ P et la condition de fin prouvent le résultat attendu.

La principale difficulté sera souvent de trouver l'invariant : en fait rechercher un invariant peut être une étape utile lors de la recherche d'un algorithme.

Dans le cas de la fonction `estPuissance2` un invariant possible est

$$P \ : \quad \text{il existe un entier positif } k \text{ tel que } n = 2^k.p$$

1. P est valide à l'initialisation : $n = p = 2^0.p$.
2. Si P est vérifiée, $n = 2^k.p$ et si la condition de la boucle est vérifiée alors on divise p par 2, $p' = \frac{p}{2}$ et on a bien $n = 2^{k+1}.p'$.
3. Si P est vérifiée, $n = 2^k.p$ et si la condition de la boucle ne l'est pas alors p n'est pas divisible par 2 donc p vaut 1 si et seulement si n est une puissance de 2, ce qui est le résultat attendu.

Fonctions récursives

Dans le cas des fonctions récursives l'invariant consiste à prouver l'algorithme par récurrence sur la taille de l'entrée (ou autre variant).

Dans la fonction `appartient`, on note n la longueur de la liste.
1. Pour $n = 0$, la fonction renvoie `false`, ce qui est le bon résultat car x ne peut pas appartenir à une liste vide.
2. On suppose que la fonction renvoie le bon résultat quand on l'appelle avec une liste de taille n. Pour une liste de taille $n + 1$:
   ↪ si son premier terme est x elle doit renvoyer `true`, ce qui est le cas,
   ↪ sinon x appartient à la liste si et seulement s'il appartient au reste, ce qui assuré avec l'hypothèse de récurrence.
3. Ainsi la fonction renvoie le bon résultat dans tous les cas.

1-3 Complexité

Le programme fournit-il le bon résultat dans un temps raisonnable ?
Le but ici n'est pas de prévoir le temps exact que va demander la résolution d'un problème mais d'avoir une idée de l'accroissement du temps quand la taille des données d'entrée augmente.
Pour cela on peut compter le temps utilisé en nombre de cycles du processeur.
Cependant il est souvent difficile de descendre à un niveau aussi bas d'instructions : on compte plutôt le nombre d'opérations "élémentaires" dans le langage choisi.
Même cela sera souvent inutilement compliqué. Le plus souvent on choisit une instruction élémentaire signifiante et on compte le nombre d'exécutions de cette instruction. Ce pourra être le nombre de comparaisons, d'affectation, de produits, …
Ce que l'on cherche c'est prévoir comment évolue le temps d'exécution lors que les données d'entrée ont une mesure qui augmente.
Parfois l'entrée sera caractérisée par plusieurs mesures (lorsqu'il y a plusieurs listes en paramètres, dans le cas d'une matrice, …).

On aboutit à une fonction de complexité dont la variable est une mesure de la donnée d'entrée.
Cependant la complexité peut varier selon les différentes entrées possible d'une même taille.
Nous choisirons souvent de calculer la complexité maximale[4].
On cherche donc une fonction $C(n)$ telle que, pour toutes les données d'entrée de taille n, le nombre d'instruction effectuées est majoré par $C(n)$.

[4] Mais on peut aussi déterminer la complexité en moyenne.

A. Pour la fonction `somme_tableau` le nombre d'additions est la taille du tableau : on peut écrire $C(n) = n$.

B. Pour la fonction `estPuissance2` le nombre de divisions est l'exposant k de 2 dans la décomposition en facteurs premiers de n : on a $2^k \leqslant n$ donc $C(n) \leqslant \log_2(n)$.

C. Pour la fonction `appartient` on montre par récurrence sur la longueur n de la liste que le nombre de comparaisons vérifie $C(n) \leqslant n$.

Cette fonction $C(n)$ est en général inutilement précise.
La question qu'on se pose est, on y revient, l'évolution avec la taille n.

Que devient le temps de calcul si on multiplie la taille des données par 2 ?

On utilise la notation de Landau, $C(n) = \mathcal{O}(g(n))$ qui signifie qu'il existe une constante K telle que $C(n) \leqslant Kg(n)$ pour tout n (ou pour tout $n \geqslant n_0$).
Le plus souvent g sera une fonction très simple de la forme $g(n) = n^\alpha$ ou $g(n) = a^n$.
↪ On parle de complexité linéaire quand la complexité est un $\mathcal{O}(n)$,
↪ On parle de complexité quadratique quand elle est un $\mathcal{O}(n^2)$,
↪ On parle de complexité polynomiale quand elle est un $\mathcal{O}(n^p)$,
↪ On parle de complexité quasi-constante quand elle est un $\mathcal{O}(\log_2(n))$,
↪ On parle de complexité quasi-linéaire quand elle est un $\mathcal{O}(n\log_2(n))$.
↪ On parle de complexité exponentielle quand elle est un $\mathcal{O}(a^n)$,

2 Tris simples de listes

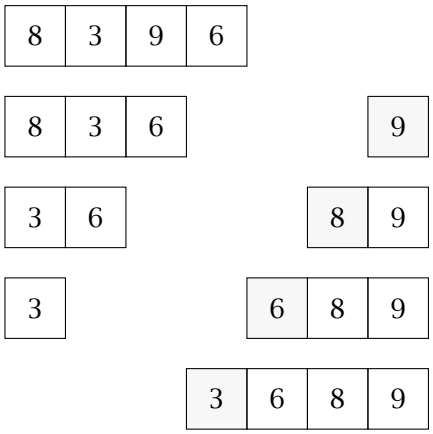
Le but de cette partie est d'écrire des fonctions qui trient une liste et d'en faire l'analyse.
On part d'une liste dont les éléments sont comparables : il existe une relation d'ordre total sur les éléments. Les cas simples sont ceux de listes d'entiers ou de flottants mais il arrivera souvent que les éléments soient plus compliqués.
On supposera donnée une fonction `plusPetit` telle que `plusPetit x y` renvoie `true` si et seulement si $x \leqslant y$.
Une liste est triée si la fonction suivante renvoie `true`.

```
let rec estTriee liste =
  match liste with
  |[] -> true
  |[a] -> true
  |s::t::q -> (plusPetit s t)&&(estTriee t::q);;
```

Les fonctions de tri devront vérifier que si `tri liste` renvoie `liste1` alors `liste` et `liste1` ont les mêmes éléments, comptés avec leur multiplicité, et `liste1` est triée.

2-1 Tri par sélection

Le tri par insertion consiste à construire la liste triée terme-à-terme en ajoutant les éléments dans l'ordre. Comme on construit une liste en ajoutant les éléments par la gauche on devra commencer par le plus grand. À chaque étape on retire le maximum de ce qui reste à trier et on l'ajoute dans la liste triée.
Exemple : on veut trier 8, 3, 9, 6




```
let rec selectionMax liste =
  match liste with
  |[] -> failwith "selectionMax d'une liste vide"
  |[x] -> x, []
  |t::q -> let max, reste = selectionMax q in
           if  pluspetit t max then max, t::reste
                               else t, q;;

let triSelection liste =
  let rec auxTriS aFaire fait =
    match aFaire with
    |[] -> fait
    |_ -> let max, reste = selectionMax aFaire in
          auxTriS reste max::fait in
  auxTriS liste [];;
```

Code III.1 Tri par sélection

On a écrit séparément une fonction qui sépare une liste (non vide) en un terme maximal et le reste.
La fonction de tri fait appel à une fonction auxiliaire qui gère la liste des termes restant à trier et la liste des termes sélectionnés .

Analyse

1. **Terminaison**
   Dans la fonction `selectionMax` appelée avec une liste non vide, on fait un appel récursif avec une liste de taille strictement inférieure : la taille est un variant donc la fonction termine.
   Pour la fonction `triSelection` la longueur de la liste est aussi un variant de boucle.

2. **Preuve de `selectionMax`**
   On va prouver, par récurrence; la propriété P_n :
   pour tout liste `l` de taille $n \geqslant 1$, `selectionMax l` renvoie la valeur maximale de la liste et une liste dont les éléments sont ceux de `l` privés d'une valeur maximale.
   ↪ P_1 est vraie car `x` est le maximum de `[x]` et il ne reste rien.
   ↪ On suppose P_n vraie. `l = t::q` est de taille $n+1$.
   `max, reste = selectionMax q` définit, par hypothèse de récurrence, le maximum de `q` et une liste comportant les éléments de `q` à l'exception d'une valeur maximale.
   ↪ Si `t <= max` alors `max` reste le maximum de `l`. De plus `t::reste` comporte tous les éléments de `l` sauf l'élément de `q` qui vaut le maximum.
   ↪ Si `t > max` alors `t` est un maximum de `l` et `q` `q` comporte tous les éléments de `l` à l'exception d'une valeur maximale.

3. **Preuve de `triSelection`**
   On va prouver, par récurrence; la propriété P_n :
   pour tout liste `aFaire` de taille n, `auxTriS aFaire fait` renvoie une liste dont les éléments sont ceux de `aFaire` triés suivis des éléments de `fait` dans leur ordre initial.
   ↪ P_0 est trivialement vraie.
   ↪ On suppose P_n vraie. `aFaire` est de taille $n+1$.
   On a prouvé que `max, reste = selectionMax aFaire` définissait un maximum de la liste et une liste de taille n contenant les éléments de `aFaire` inférieurs au maximum.
   Par hypothèse de récurrence, `auxTriS reste max::fait` est une liste avec les éléments de `reste` triés suivis des éléments de `max::fait`.
   Les $n+1$ premiers éléments sont donc ceux de `aFaire` et ils sont suivis de `fait`. De plus les n premiers sont triés et sont suivi d'un terme qui les majore donc les éléments de `aFaire` sont triés. P_{n+1} est vraie.
   La preuve de `auxTriS` implique que `auxTriS liste []` retourne les éléments de `liste` triés ce qui prouve `triSelection`.

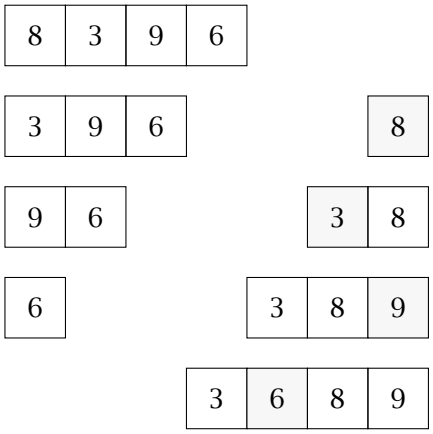
4. **Complexité**
   On note $C'(n)$ la complexité, en nombre de comparaisons, de `selectionMax` pour une liste de taille n. On a $C'(1) = 0$ et $C'(n+1) = C'(n) + 1$ donc $C'(n) = n - 1$.

   On note $C''(p,q)$ la complexité de `auxTriS aFaire fait` quand p est la taille de `aFaire` et q celle de `fait`.
   On a $C''(0,q) = 0$ et $C''(p+1,q) = C'(p+1) + C''(p,q+1)$.
   Ainsi $C''(p,q) = \sum_{k=1}^{p} C'(k) = \frac{p(p-1)}{2}$.

   La complexité de `triSelection` pour une liste de taille n est donc $\frac{n}{2}(n-1) = \mathcal{O}(n^2)$.

2-2 Tri par insertion

On va présenter un autre tri naturel où l'on diminue d'un élément à la fois l'ensemble des éléments à trier sans choisir. On construit alors une liste triée dans laquelle on doit insérer un élément en conservant l'ordre : c'est le tri par insertion. C'est le tri souvent employé par les joueurs de cartes ou par l'enseignant qui trie les copies.
Exemple : on veut trier 8, 3, 9, 6



Le programme s'écrit en suivant l'algorithme récursivement : pour trier une liste on trie la queue puis on insère la tête en respectant l'ordre. On peut remarquer que l'on trie "à l'envers" : les éléments sont en fait insérés à partir du dernier.
Il faut écrire une fonction d'insertion : il suffit de comparer l'élément à insérer avec la tête pour choisir de le placer en tête ou de l'insérer dans la suite.

```
let rec insertion x liste =
  match liste with
  |[] -> [x]
  |t::q when plusPetit t x -> t::(insertion x q)
  |_ -> x::liste;;

let triInsertion liste =
  match liste with
  |[] -> []
  |t::q -> let triee = triInsertion q in
           insertion t triee;;
```

Code III.2 Tri par insertion

Analyse

1. **Terminaison**
   Dans la fonction `insertion`, on fait un appel récursif avec une liste de taille strictement inférieure : on aboutit donc soit à une liste vide soit à une tête supérieure à l'élément à insérer. La fonction termine donc.
   De même la longueur de la liste est un variant de boucle pour `triInsertion`.

2. **Preuve de `insertion`**
   Soit P_n la propriété :
   pour tout liste `l` triée de taille n `insertion x l` renvoie une liste triée dont les éléments sont ceux de `t` ajoutés à `x`.
   ↪ P_0 est vraie car `[x]` est triée.
   ↪ On suppose P_n vraie. `l = t::q` est de taille $n+1$.
   - Si `t <= x` alors `t` est majoré par tous les élément de `q` et par `x` donc par tous les éléments de `ll = insertion x q` car cette liste comporte tous les élément de `q` avec `x` par hypothèse de récurrence.
     On a aussi `ll` triée donc `t::ll` est triée et contient tous les éléments.
   - Si `t > x` alors `x::t::q` est triée et contient tous les éléments.

   Dans les deux cas P_{n+1} est vraie.
   ↪ P_n est vraie pour tout n.

3. **Preuve de `triInsertion`**
   Soit P_n la propriété :
   pour tout liste `l` taille n `triInsertion l` renvoie une liste triée dont les éléments sont ceux de `l`.
   ↪ P_0 est trivialement vraie.
   ↪ On suppose P_n vraie. `l = t::q` est de taille $n+1$.
   Par hypothèse de récurrence, `ll = triInsertion q` est triée et contient les éléments de `q` car `q` est de longueur n.
   D'après la preuve précédente `insertion t ll` est triée et contient `t` les éléments de `ll` donc tous les éléments initiaux.
   Ainsi P_{n+1} est vraie.
   ↪ P_n est vraie pour tout n.

4. **Complexité**
   La complexité est simple à majorer : dans l'insertion on effectue au plus p comparaisons où p est la longueur de la liste dans laquelle on insère donc

$$C(n) \leqslant \sum_{i=1}^{n} (i-1) = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} \leqslant \frac{n^2}{2} = \mathcal{O}(n^2)$$

   Si la liste est dans l'ordre inverse on a $C(n) = \dfrac{n(n-1)}{2}$,
   par contre si la liste est déjà triée on ne fait que $n-1$ comparaisons.

3 Exercices

3-1 Analyses

Ex. 1 **Recherche dans un tableau**
Analyser l'algorithme de recherche d'un élément dans un tableau du chapitre I.

Ex. 2 **Somme d'une liste**
Analyser l'algorithme du calcul de la somme des termes d'une liste.

Ex. 3 Coefficients binomiaux
Pour calculer les coefficients binomiaux $\binom{n}{p}$ on peut utiliser la définition $\frac{n!}{p!.(n-p)!}$ mais cela demandera de calculer des grandes valeurs de la factorielle qui risquent de dépasser la taille maximale alors que le coefficient binomial reste calculable.

a. Écrire et analyser l'algorithme récursif qui calcule $\binom{n}{p}$ à l'aide de la formule de Pascal $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$.

b. Même question en utilisant $\binom{n}{p} = \frac{n}{p}\binom{n-1}{p-1}$ pour $n \geqslant p \geqslant 1$.

c. Lequel est-il judicieux de choisir ?

3-2 Somme de tranches

On souhaite calculer la somme maximale d'une tranche d'une liste, une tranche étant une suite de termes consécutifs de la liste.
Par exemple, pour la liste `[2; -5; 7; -4; 6; -8; 4; 3]` la somme maximale est 9 et elle correspond aux termes 7, -4 et 6.

Une première idée est de remarquer que les tranches d'une liste `t::q` sont soit des tranches de `q`, soit des tranches commençant par `t`.

Ex. 4 Sommes de tête
Écrire une fonction `maxTete : int list -> int` qui calcule la somme maximale des tranches d'une liste qui commencent par la tête de la liste.
En donner une analyse.

Ex. 5 Premier algorithme
En déduire un algorithme de calcul de tranche maximale. En donner une analyse.

On propose l'algorithme suivant

```
let trancheMax1 liste =
  let rec aux reste t_max fin =
    match reste with
    |[] -> t_max
    |t::q -> let fin1 = fin + t in
             aux q (max t_max fin1) (max 0 fin1)
  in aux liste 0 0 ;;
```

Ex. 6 **Analyse du second algorithme**
Cet algorithme est-il correct ? Améliore-t-il le précédent ?

3-3 Tris

Ex. 7 **Tri par sélection, autre écriture**
Le tri par sélection qui a été proposé fait appel à une fonction auxiliaire. On peut aussi le penser dans l'autre sens : on sépare un élément minimal et on le place en tête de la liste obtenue en triant le reste.
Écrire cet algorithme.

Ex. 8 **Complexité moyenne du tri par insertion**
On considère le tri des listes avec n éléments **distincts** dont tous les désordres sont équiprobables ; pour simplifier on supposera que les n éléments sont les n premiers entiers et l_σ est la liste associée à la permutation $\sigma : l_\sigma = [\sigma(1); \sigma(2); ...; \sigma(n)]$.
Si $C(\ell)$ est la complexité du tri de la liste ℓ, la complexité moyenne est donc $\overline{C}(n) = \frac{1}{n!} \sum_{\sigma \in S_n} C(l_\sigma)$.
Calculer cette moyenne.

Ex. 9 **Liste aléatoire**
Écrire une fonction qui renvoie une liste aléatoire, de taille n et comportant les n entiers 0, 1, ..., $n-1$.

Indication : `Random.int n` renvoie un entier pseudo-aléatoire compris entre 0 et $n-1$.

Chapitre IV

Diviser pour régner

L'informatique est souvent utilisée pour faire des calculs répétés ou portant sur des ensembles d'objets. Les boucles consistent à travailler sur une valeur ou un objet à la fois.
La récursivité appliquée à une liste peut être un peu différente : on considère la liste comme découpée en une tête et une queue, on traite la queue (récursivement) et on combine le résultat avec la tête.

Nous allons, dans ce chapitre, généraliser cette méthode en découpant l'ensemble des objets à traiter en deux parties (ou davantage), en traitant chaque partie puis en assemblant les résultats. C'est la méthode "diviser pour régner".

Ce chapitre contient divers exemple d'application de cette méthode.
On l'appliquera dans le chapitre suivant pour l'écriture d'algorithmes de tris qui seront plus efficaces que les tris vus dans le chapitre précédent.

Dans ce chapitre, la division entière, `a/b` dans **OCaml**, *sera notée $a \div b$ en mathématiques, c'est la notation de l'école primaire. $a \div b = \lfloor \frac{a}{b} \rfloor$.*

1 Exponentiation rapide

La fonction récursive classique peut s'écrire

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

Le nombre de multiplication pour calculer a^n est n.
On va prouver que l'on peut calculer plus efficacement les puissances en divisant n en 2 parties égales ou de différence 1.
Si n est pair, $n = 2p$, on peut écrire $a^{2p} = a^p.a^p$, on calcule a^p, **que l'on garde en mémoire**, puis on le multiplie par lui-même : on ajoute une seule multiplication au calcul de a^p.
De même, si $n = 2p + 1$, $a^{2p+1} = a^p.a^p.a$ permet de calculer a^n en n'ajoutant que 2 multiplications après avoir calculé a^p.
On va utiliser la récursivité pour poursuivre cette idée.

```
let rec expRapide a n =
  match n with
  |0 -> 1
  |n -> let b = expRapide a (n/2) in
        if n mod 2 = 0
        then b*b
        else a*b*b;;
```

Code IV.1 Exponentiation rapide

L'analyse est relativement simple

1. **Terminaison** Pour $n > 0$ on a $n/2 < n$ donc la variable n est un variant.

2. **Preuve** On prouve par récurrence sur n que le résultat est bien a^n.

3. **Complexité** Si on note $C(n)$ le nombre de multiplications effectuées lors de l'appel de `expRapide a n` on a $C(0) = 0$, $C(2p) = C(p) + 1$ et $C(2p+1) = C(p) + 2$.
   On a donc $C(n) \leqslant C(n \div 2) + 2$ pour tout $n \geqslant 1$.
   Si on a $2^p \leqslant n < 2^{p+1}$ alors $2^{p-1} \leqslant \lfloor \frac{n}{2} \rfloor < 2^p$ donc, par récurrence on a
   $C(n) \leqslant C(1) + 2p$ pour $2^p \leqslant n < 2^{p+1}$ car 1 est le seul entier tel que $2^0 \leqslant n < 2^1$.
   On en déduit que $C(n) \leqslant 2(p+1)$ pour $2^p \leqslant n < 2^{p+1}$ donc $p \leqslant \log_2(n)$.
   Ainsi la complexité est majorée par $2\log_2(n) + 2$, c'est un $\mathcal{O}(\log_2(n))$.
   Le gain de rapidité est net : pour $n = 10^6$ on passe de 10^6 multiplications à moins de cinquante.

La valeur de a^n tend vers l'infini très rapidement, il ne semble pas utile d'avoir un algorithme rapide pour les grandes valeurs de n. En fait l'exponentiation est utilisée pour de très grandes valeurs de n en faisant les calculs modulo un entier N ; dans ce cas l'algorithme ci-dessus est indispensable.
Remarque : la complexité n'est pas une fonction croissante de n.

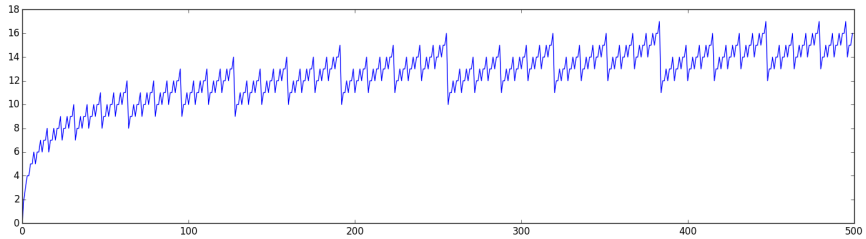


Figure IV.1 Complexité de l'exponentiation rapide

2 Recherche dans un tableau

On veut chercher un élément dans un ensemble.
Nous allons ici supposer que l'ensemble est représenté par un tableau **trié**.
On cherche donc à écrire une fonction `cherche x t` qui renvoie `true` ou `false` selon que x est ou non un élément du tableau `t`.
Pour tirer parti du tri des éléments, on va chercher à diviser le tableau : un test à une position médiane permet de chercher l'élément dans une moitié du tableau. Ainsi on va diminuer d'un facteur 2 la plage d'indices où chercher.
On supposera que le tableau est non vide.

```
let cherche x tab =
  let n = Array.length tab in
  let rec auxCh a b =
  if a = b
  then tab.(a) = x
  else let c = (a + b)/2 in
       if  tabl.(c) < x
       then auxCh (c+1) b
       else auxCh a c in
  aux 0 (n-1);;
```

Code IV.2 Recherche récursive dans un tableau trié

2-1 Analyse de la recherche

La fonction `cherche` ne fait qu'appeler la fonction auxiliaire ; nous allons étudier la fonction récursive `auxCh`.

1. **Terminaison** On montre par récurrence sur k la propriété
   $\mathcal{P}(k)$: l'appel de `auxCh a b` termine pour $a \leqslant b \leqslant a + k$.
   Pour $k = 0$, on a $a = b$ et `auxCh` renvoie le résultat de la comparaison `tab.(a) = x` donc termine : $\mathcal{P}(0)$ est vérifiée.

   On suppose que $\mathcal{P}(k)$ est valide avec $k \geqslant 0$.
   On suppose qu'on a $a \leqslant b \leqslant a + k + 1$.
   $b \geqslant a$ implique $c = (a + b) \div 2 \geqslant a$.
   De plus $c = (a + b) \div 2 \leqslant (2a + k + 1) \div 2 = a + (k + 1) \div 2$.
   On a $k + 1 \geqslant 1$ donc $(k + 1) \div 2 \leqslant \frac{k+1}{2} < k + 1$ d'où $c < a + k + 1$.
   En particulier $c \leqslant a + k + 1$.

`auxCh a b` fait alors appel
soit à `auxCh a c` avec $a \leqslant c \leqslant a+k$
soit à `auxCh (c+1) b` avec $c+1 \leqslant b = a+k+1 \leqslant c+k+1 = (c+1)+k$.
Dans les deux cas les appels terminent d'après $\mathcal{P}(k)$ donc l'appel de `auxCh a b` termine : $\mathcal{P}(k+1)$ est vérifié. On a prouvé la récurrence.

2. **Preuve** On montre par récurrence sur k la propriété
$\mathcal{Q}(k)$: l'appel de `auxCh a b` pour $a \leqslant b \leqslant a+k$ renvoie `true` si et seulement si il existe un indice i tel que $a \leqslant i \leqslant b$ et `tab.(i)` vaut x.
(*Par abus de langage, on dira dans ce cas que x est dans le tableau entre a et b.*)
Pour $k=0$, on a $a=b$ et `auxCh` renvoie le résultat de la comparaison `tab.(min) = x` : on teste le seul élément possible donc $\mathcal{Q}(0)$ est vérifiée.

   On suppose que $\mathcal{Q}(k)$ est valide avec $k \geqslant 0$.
   On suppose qu'on a $a \leqslant b \leqslant a+k+1$.
   On a vu qu'on faisait alors appel à `auxCh a1 b1` avec $a_1 = a$ et $b_1 = c$ ou $a_1 = c+1$ et $b_1 = b$ avec, dans les deux cas, $a_1 \leqslant b_1 \leqslant a_1 + k + 1$.
   - ↪ Si x n'est pas dans le tableau entre a et b alors il n'est pas dans le tableau entre a_1 et b_1 donc, d'après $\mathcal{Q}(k)$, l'appel récursif renvoie `false`.
   - ↪ Si x est dans le tableau avec `tab.(c) < x` alors x n'est pas dans le tableau entre a et c donc il est dans le tableau entre $c+1$ et b et, d'après $\mathcal{Q}(k)$, l'appel de `auxCh (c+1) b` renvoie `true`.
   - ↪ Si x est dans le tableau avec `tab.(c) = x` alors x est dans le tableau entre a et c donc, d'après $\mathcal{Q}(k)$, l'appel de `auxCh a c` renvoie `true`.
   - ↪ Si x est dans le tableau avec `tab.(c) > x` alors x n'est pas dans le tableau entre c et b donc il est dans le tableau entre a et $c-1$ et, d'après $\mathcal{Q}(k)$, l'appel de `auxCh a c` renvoie `true`.

   Dans tous les cas l'appel de `auxCh a b` renvoie la bonne réponse donc $\mathcal{Q}(k+1)$ est vérifiée.
   On a prouvé la récurrence.

3. **Complexité** On va évaluer la complexité en comptant le nombre de comparaisons entre la valeur recherchée et les valeurs du tableau.
   Lors de chaque appel de la fonction auxiliaire on effectue 1 comparaison.
   - ↪ Si le nombre d'indices parmi lesquels chercher est pair, $n = 2m$, on a $b = a+2m-1$ et $c = a+m-1$ donc on cherchera parmi m indices dans l'appel récursif entre a et c ou entre $c+1$ et b.
     Si on avait $n \leqslant 2^p$ alors on cherche parmi 2^{p-1} indices au plus dans l'appel récursif.
   - ↪ Si le nombre d'indices est impair, $n = 2m+1$, on a $b = a+2m$ donc $c = a+m$. Entre a et c il y a donc $m+1$ indices parmi lesquels chercher
     et m indices entre $c+1$ et b.
     Si on avait $n \leqslant 2^p$ alors $n \leqslant 2^p - 1$ car n est impair donc $m = \frac{m-1}{2} \leqslant 2^{p-1} - 1$ donc on cherche parmi 2^{p-1} indices au plus dans l'appel récursif.

↪ Ainsi, si $n \leqslant 2^p$, on aboutit, au plus tard après p appels, à chercher parmi 2^0 élément donc le programme s'arrête après au plus $p+1$ comparaisons.
↪ Si p est tel que $2^{p-1} < n \leqslant 2^p$ on a $p \leqslant \log_2(n)+1$ donc on effectue au plus $\log_2(n)+2$ comparaisons : la complexité est un $\mathcal{O}(\ln(n))$.

3 Produit de polynômes

On s'intéresse au produit de polynômes dont les degrés peuvent être grands.
Le polynôme $P = a_0 + a_1X + \cdots + a_nX^n$ sera représenté par la liste

```
polP = [a0; a1; ...; an]
```

Ex. 1 **Fonctions arithmétiques** Écrire les fonctions `somme`, `difference` et `produit` qui calculent les opérations sur deux polynômes. Donner les complexités en nombre d'opérations de flottants en fonction du degré des polynômes.

Le produit de deux polynômes de degré n, calculé en suivant la définition classique demande $(n+1)^2$ multiplications et $(n+1)^2$ additions.

Une méthode de type diviser pour régner permet d'améliorer ce résultat.
On peut écrire un polynôme de degré $n \leqslant 2p-1$ sous la forme

$$P = \sum_{k=0}^{2p-1} a_kX^k = \sum_{k=0}^{p-1} a_kX^k + X^p \sum_{k=0}^{p-1} a_{p+k}X^k = P_1 + X^pP_2$$

De même on écrit $Q = Q_1 + X^pQ_2$ pour Q de degré $m \leqslant 2p-1$.
On a $P.Q = P_1.Q_1 + X^p(P_1.Q_2 + P_2.Q_1) + X^{2p}P_2.Q_2$.
Il y 4 produits de polynômes de degré au plus $p-1$ à faire.
Un astuce difficile à trouver mais facile à prouver est que

$$P_1.Q_2 + P_2.Q_1 = (P_1+P_2)(Q_1+Q_2) - P_1.Q_1 - P_2.Q_2$$

On peut donc calculer les 3 termes avec seulement 3 produits de polynômes.

On va utiliser ce résultat de manière récursive.

1. Le nombre de termes est égal au degré augmenté de 1 donc

la taille de découpage est la moitié supérieure des tailles des listes.

2. On découpera les listes en deux parties par une fonction `separer k liste`.

```
let rec separer k liste =
  match k, liste with
  |0, _ -> [], liste
  |_, [] -> [], []
  |k, t::q -> let l1, l2 = separer (k-1) q in (t::l1), l2;;
```

3. Le produit par X^r sera réalisé par une fonction `decale k liste`.

```
let rec decale k liste =
  match k with
  |0 -> liste
  |k -> 0.0::(decale (k-1) liste);;
```

On peut alors écrire l'algorithme.

```
let produit_rapide p1 p2 =
  let n = max (List.length p1) (List.length p2) in
  let rec aux_p q1 q2 m =
    match q1, q2 with
    |[], _ -> []
    |_, [] -> []
    |[a], [b] -> [a*.b]
    |_ -> let p = (m+1)/2 in
          let a1, b1 = separer p q1 in
          let a2, b2 = separer p q2 in
          let r1 = aux_p a1 a2 p in
          let r3 = aux_p b1 b2 p in
          let r = aux_p (somme a1 b1) (somme a2 b2) p in
          let r2 = decale p (difference r (somme r1 r3)) in
          somme (somme r1 r2) (decale (2*p) r3) in
  aux_prod p1 p2 n;;
```

Code IV.3 Produit rapide de polynômes

3-1 Complexité

On note mult(p) le nombre de multiplications effectuées dans la fonction `aux_p q1 q2 p` et add(p) le nombre d'additions.
p est un majorant de la taille des listes `q1` et `q2` donc mult$(1) = 1$ et add$(1) = 0$.
Dans l'appel de `aux_p q1 q2 p` on effectue 3 appels de `aux_p` et 6 appels de sommes ou de différences avec des listes de taille m au plus, avec $m = \lceil \frac{p}{2} \rceil$.
On a donc mult$(p) = 3$mult(m) et add$(p) \leqslant 3$add$(m) + 6m$.
Si on majore p par 2^r, on a $m = \lceil \frac{p}{2} \rceil \leqslant 2^{r-1}$ d'où, par récurrence sur r, mult$(p) \leqslant 3^r$.
Si on note u_r un majorant de add(p) pour $p \leqslant 2^r$ la relation add$(p) \leqslant 3$add$(m) + 6m$ donne $u_r \leqslant 3.u_{r-1} + 6.2^{r-1}$. En posant $v_r = \dfrac{u_r}{3^r}$ on aboutit à $v_r \leqslant v_{r-1} + 2.\left(\dfrac{2}{3}\right)^{r-1}$.
On a donc $v_r \leqslant v_0 + \displaystyle\sum_{k=0}^{r-1} 2.\left(\frac{2}{3}\right)^{r-1} = 2\frac{1-\left(\frac{2}{3}\right)^r}{1-\frac{2}{3}} = 6\left(1-\left(\frac{2}{3}\right)^r\right) \leqslant 6$.
Ainsi on aboutit à add$(p) \leqslant 6.3^r$ pour $p \leqslant 2^r$.
On peut conclure que si P et Q sont des polynômes de même degré n avec $2^{r-1} \leqslant n < 2^r$, le nombre d'opérations est majoré par $7.3^r = 7.e^{r\ln(3)} = 7.2^{r\ln(3)/\ln(2)} = 7.(2^r)^\alpha \leqslant 7.(2n)^\alpha$.
La complexité est donc un $\mathcal{O}(n^\alpha)$ avec $\alpha = \frac{\ln(3)}{\ln(2)} \sim 1,6$.

4 Produit de matrices

On s'intéresse au produit de deux matrices carrées de taille n.
L'algorithme classique suit la définition mathématique : $c_{i,j} = \displaystyle\sum_{k=1}^{n} a_{i,k}b_{k,j}$.
Les matrices sont représentées par des tableaux doubles de flottants : `float array array`.

```
let produit_matrices a b =
  let n = Array.length a in
  (* Les matrices doivent avoir la même taille *)
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      let s = ref 0.0 in
      for k = 0 to (n-1) do
        s:= !s +. a.(i).(k) *. b.(k).(j) done;
      c.(i).(j) <- !s done done;
  c;;
```

La complexité, en nombre d'opérations, est $2.n^3$ car il y a une addition et une multiplication par valeur des indices i, j et k.

4-1 Méthode de Strassen

En particulier le produit de deux matrices 2×2 demande 8 multiplications :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}.\begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} aa'+bc' & ab'+bd' \\ ca'+dc' & cb'+dd' \end{pmatrix}$$

Strassen, en 1969, a proposé une méthode pour calculer ce produit avec 7 multiplications.
On calcule les 7 produits (formules P):
↪ $p_1 = (a+d).(a'+d')$
↪ $p_2 = (a-c).(a'+b')$
↪ $p_3 = (b-d).(c'+d')$
↪ $p_4 = a.(b'-d')$
↪ $p_5 = (a+b).d'$
↪ $p_6 = (c+d).a'$
↪ $p_7 = d.(a'-c')$

On les combine avec des additions ou des soustractions (formules C):
↪ $aa'+bc' = p_1+p_3-p_5-p_7$
↪ $ab'+bd' = p_4+p_5$
↪ $ca'+dc' = p_6-p_7$
↪ $cb'+dd' = p_1-p_2+p_4-p_6$

On remarquera qu'on a effectué 18 additions et soustractions.

4-2 Produit de matrices d'ordre 2^r

Une méthode de type diviser pour régner va utiliser le résultat ci-dessus pour les matrices d'ordre une puissance de 2. En effet on peut écrire ces matrices par blocs où les blocs ont une taille $2^{r-1} \times 2^{r-1}$. On calculera le produit des blocs avec la méthode de Strassen pour ne faire que que 7 produits, ceux-ci étant calculés récursivement.

1. On définit les 4 blocs de taille moitié ;

   `decoupe A` renvoie `A1, A2, A3, A4` tels que $A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}$.

2. On calcule les matrices P_1 à P_7 avec les formules P ci-dessus où l'on remplace les coefficients des matrices par les blocs correspondants.
   On aura besoin des fonctions `sum` et `sub` qui calculent la somme et la différence de deux matrices (carrées) de même taille.

3. On calcule ensuite les blocs de la matrice produit avec les formules C.

4. On reconstitue la matrice produit depuis ses blocs avec la fonction `assemblage`, inverse de la fonction `decoupe`.

Ex. 2 Fonctions auxiliaires
Écrire les fonctions `decoupe`, `sum`, `sub` et `assemblages` définies dans le texte.

```
let rec prod a b =
  let n = Array.length a in
  if n = 1
  then [|[|a.(0).(0) *. b.(0).(0)|]|]
  else let a1, a2, a3, a4 = decoupe a in
       let b1, b2, b3, b4 = decoupe b in
       let p1 = prod (sum a1 a4) (sum b1 b4) in
       let p2 = prod (sub a1 a3) (sum b1 b2) in
       let p3 = prod (sub a2 a4) (sum b3 b4) in
       let p4 = prod a1 (sub b2 b4) in
       let p5 = prod (sum a1 a2) b4 in
       let p6 = prod (sum a3 a4) b1 in
       let p7 = prod a4 (sub b1 b3) in
       let c1 = sub (sum p1 p3) (sum p5 p7) in
       let c2 = sum p4 p5 in
       let c3 = sub p6 p7 in
       let c4 = sum (sub p1 p2) (sub p4 p6) in
       assemblage c1 c2 c3 c4;;
```

Code IV.4 Produit de matrices de taille 2^r

4-3 Complexité

On note mult(r) le nombre de multiplications et add(r) le nombre d'additions et soustractions effectuées dans le produit de deux matrices carrées de taille 2^r.
On a mult$(0) = 1$ et add$(0) = 0$.
Les formules de Strassen impliquent mult$(r+1) = 7.$mult(r) donc mult$(r) = 7^r$
et add$(r+1) = 18.(2^r)^2 + 7.$add(r) car la somme et la différence de deux matrices carrées de taille n demandent chacune n^2 additions et additions.
Si on pose $a_r = 7^{-r}.$add(r) on aboutit à $a_r = a_{r-1} + \frac{18}{7}.\left(\frac{4}{7}\right)^{r-1}$.

d'où $a_r = a_0 + \frac{18}{7}\sum_{k=0}^{r-1}\left(\frac{4}{7}\right)^k = \frac{18}{7}\frac{1-\left(\frac{4}{7}\right)^r}{1-\frac{4}{7}} = 6\left(1-\left(\frac{4}{7}\right)^r\right)$.
Ainsi on aboutit à $\text{add}(r) = 6\,(7^r - 4^r)$: le nombre total d'opérations est donc $7^{r+1} - 6.4^r$.
Le nombre d'opération est inférieur à $2.n^3 = 2^{3r+1}$ à partir de $r = 10$, c'est-à-dire
$n = 2^{10} = 1024$. À ce moment le nombre d'opération est de l'ordre de 2.10^9.

4-4 Cas général

On peut alors calculer le produit de deux matrices quelconques (compatibles).

1. On calcule la plus petite puissance de 2, 2^r, qui majore les tailles des matrices.
2. On borde les matrices par des 0 pour obtenir des matrices carrées de taille 2^r.
3. On calcule le produit grâce à la fonction ci-dessus.
4. On extrait la matrice de la bonne taille du produit.

Ex. 3 **Fonctions auxiliaires** Écrire les fonctions
`majorant2 n` qui calcule la plus petite puissance de 2 minorée par n,
`borde mat n` qui ajoute des 0 à gauche et en dessous d'une matrice `mat` pour obtenir une matrice $n \times n$,
`extraction mat p q` qui extrait la matrice de taille $p \times q$ en haut à gauche de `mat`.

```
let produit_matrice_rapide a b =
  let p1 = Array.length a in
  let q1 = Array.length a.(0) in
  let p2 = Array.length b in
  let q2 = Array.length b.(0) in
  if q1 <> p2
  then failwith "Les matrices ne peuvent pas être multipliées"
  else let n = majorant2 (max (max p1 q1) (max p2 q2)) in
       let a1 = borde a n in
       let b1 = borde b n in
       extraction (prod a1 b1) p1 q2;;
```

Si on note n la plus grande taille des matrice dont on fait le produit, on est amené au produit de deux matrices de taille 2^r avec $2^{r-1} < n \leqslant 2^r$.
On a vu que la complexité était alors majorée par $7^{r+1} = 49.7^{r-1} = 49.2^{(r-1).\ln(7)/\ln(2)} \leqslant 49.n^\beta$ avec $\beta = \frac{\ln(7)}{\ln(2)} \simeq 2.81$. La complexité en $\mathcal{O}(n^\beta)$ n'apporte pas vraiment un gain par rapport à celle en $\mathcal{O}(n^\beta)$ de l'algorithme basique.

5 Transformée de Fourier rapide

La transformée de Fourier rapide est un outil indispensable dans le traitement du signal. Elle consiste à décomposer un signal temporel en une combinaison linaire de signaux périodiques élémentaires. Cependant la technique peut aussi être utilisée dans le cadre plus algébrique du produit de deux polynômes.
Nous avons vu que le produit de deux polynômes demandait un nombre d'opérations de l'ordre de n^2 où n est un majorant des degrés des polynômes. Nous avons vu qu'une méthode diviser pour régner permettait de ramener cette complexité en $\mathcal{O}(n^\alpha)$ avec α de l'ordre de $1,6$.
En changeant complètement de point de vue nous allons donner un algorithme en $\mathcal{O}(n.\ln(n))$.

5-1 Produit de polynômes

Pour calculer le produit de deux polynômes on peut s'appuyer sur le théorème d'interpolation : pour toute suite de couples $(a_0,b_0),(a_1,b_1),\ldots(a_n,b_n)$ il existe un unique polynôme P de degré n au plus tel que $P(a_i)=b_i$ pour tout $i\in\{0,1,2,\ldots,n\}$.
Le produit de deux polynômes P et Q de degrés respectifs p et q est un polynôme de degré $p+q$. Pour le calculer il suffit de se donner $p+q+1$ points $a_0,a_1,\ldots,a_{p+q}$ et de calculer les valeurs de P et Q en ces points.
Le produit sera alors le polynôme R défini par $R(a_i)=P(a_i).Q(a_i)$.
Le produit des valeurs est linéaire ainsi que l'évaluation d'un polynôme en un point mais le nombre de points induit une complexité quadratique. Le calcul du polynôme d'interpolation est aussi de complexité quadratique en général.
Cependant un choix judicieux des points d'interpolation va permettre de pouvoir appliquer une méthode diviser pour régner et de ramener en une complexité quasi-linéaire.

5-2 Racines de l'unité

P est un polynôme de degré au plus $N-1$: $P=a_0+a_1X+\cdots+a_{N-1}X^{N-1}$.
On considère les N racines N-ièmes de l'unité $r_k=e^{2ik\pi/N}$.
On note $p_k=P(r_k)=\sum_{m=0}^{N-1}a_m r_k^m$: on a donc

$$\begin{pmatrix}p_0\\p_1\\\vdots\\p_{N-1}\end{pmatrix}=\begin{pmatrix}r_0^0&r_0^1&\cdots&r_0^{N-1}\\r_1^0&r_1^1&\cdots&r_1^{N-1}\\\vdots&\vdots&&\vdots\\r_{N-1}^0&r_{N-1}^1&\cdots&r_{N-1}^{N-1}\end{pmatrix}.\begin{pmatrix}a_0\\a_1\\\vdots\\a_{N-1}\end{pmatrix}=r.\begin{pmatrix}a_0\\a_1\\\vdots\\a_{N-1}\end{pmatrix}$$

Calculer les valeur de P en les points r_k revient donc à calculer le produit de la matrice Ω par le vecteur colonne des composantes de P.
On a $\Omega = (\omega_{k,m})_{0\leqslant k,m<N}$ avec $\omega_{k,m} = r_k^m = e^{2ikm\pi/N}$.
La matrice $U = \overline{\Omega}.\Omega$ a pour coefficients
$u_{a,b} = \sum_{c=0}^{N-1} \overline{r_a^c} r_c^b = \sum_{c=0}^{N-1} e^{-2iac\pi/N} e^{2icb\pi/N} = \sum_{c=0}^{N-1} e^{2i(b-a)c\pi/N} = \sum_{c=0}^{N-1} \omega_{b-a}^c$.
Or $\sum_{c=0}^{N-1} \omega_{b-a}^c = \begin{cases} N & \text{si } \omega_{b-a} = 1 \\ \dfrac{1-\omega_{b-a}^N}{1-\omega_{b-a}} = 0 & \text{sinon} \end{cases}$ avec $\omega_{b-a} = 1$ si et seulement $a = b + kN$.
On a donc $U = NI_N$ puis $\omega^{-1} = \frac{1}{N}\overline{\Omega}$.
On peut calculer les coefficients de P à partir des p_k en multipliant par $\frac{1}{N}\overline{\Omega}$.
On voit ici que les opérations de calcul des valeur et de calculs du polynôme d'interpolation ont la même forme ; toute simplification de l'une sera applicable à l'autre.

5-3 Réduction des calculs

On suppose ici que N est pair : $N = 2M$.

$$\begin{aligned} p_k &= \sum_{m=0}^{N-1} e^{2ikm\pi/N} a_m = \sum_{m=0}^{2M-1} e^{2ikm\pi/2M} a_m \\ &= \sum_{m=0,\ m \text{ pair}}^{2M-1} e^{2ikm\pi/2M} a_m + \sum_{m=0,\ p \text{ impair}}^{2M-1} e^{2ikm\pi/2M} a_m \\ &= \sum_{q=0}^{M-1} e^{2ik(2q)\pi/2M} a_{2q} + \sum_{q=0}^{M-1} e^{2ik(2q+1)\pi/2M} a_{2q+1} \\ &= \sum_{q=0}^{M-1} e^{2ikq\pi/M} a_{2q} + e^{ik\pi/M} \sum_{q=0}^{M-1} e^{2ikq\pi/M} a_{2q+1} \end{aligned}$$

On remarque que, pour $0 \leqslant k < M$ on retrouve les formules de calcul des valeurs pour les polynômes $P_1 = a_0 + a_2X + \cdots + a_{2M-2}X^{M-1}$ et $P_1 = a_1 + a_3X + \cdots + a_{2M-1}X^{M-1}$ évaluées en les racines M-ièmes de l'unité : $p_k = p_{1,k} + e^{ik\pi/M} p_{2,k}$.
Pour $M \leqslant k' < 2M$ on pose $k' = k + M$ et on trouve $p_{k+m} = p_{1,k} - e^{ik\pi/M} p_{2,k}$.
Pour le calcul des a_m en fonction des p_k, on calcule le polynôme d'interpolation pour les points (ω_{2k}, p_{2k}) qui donne les coefficients $a_{1,0}, a_{1,1}, \ldots, a_{1,M-1}$ et le le polynôme d'interpolation pour les points $(\omega_{2k+1}, p_{2k+1})$ qui donne les coefficients $a_{2,0}, a_{2,1}, \ldots, a_{2,M-1}$.
Une démonstration semblable à celle ci-dessus donne

$$a_k = \frac{a_{1,k} + e^{-ik\pi/M} a_{2,k}}{2} \text{ et } a_{k+m} = \frac{a_{1,k} - e^{-ik\pi/M} a_{2,k}}{2} \text{ pour } 0 \leqslant k < M$$

5-4 Calcul pratique

La formule de réduction permet de ramener le calcul de la transformation pour $2M$ points à celui du calcul de deux transformation pour M points. Pour $N = 2^n$ on peut alors continuer jusqu'à un point et dans ce cas la transformation est l'identité.
On utilise les fonctions du module `Complex`.

```
let rec fft pol =
  let n = Array.length pol in
  if n = 1
  then [|pol.(0)|]
  else let pol1, pol2 = separation pol in
       let p1 = fft pol1 in
       let p2 = fft pol2 in
       let p = Array.make n Complex.zero in
       let m = n/2 in
       let pi = 4.0*.(atan 1.0) in
       for k = 0 to (m-1) do
         let e = Complex.polar  1.0 (pi*.(float_of_int k)
                                     /.(float_of_int m)) in
         p.(k) <- Complex.add p1.(k) (Complex.mul e p2.(k));
         p.(k+m) <- Complex.sub p1.(k) (Complex.mul e p2.(k)) done;
       p;;
```

La fonction `separation` extrait deux tableaux : l'un avec les termes d'indices pairs, l'autre avec les termes d'indice impair.

Ex. 4 **Fonctions inverse** Écrire la fonction `separation`.
Écrire une fonction `fft_inv` qui calcule la transformation inverse.

Pour calculer le produit de deux polynômes

1. on calcule la plus petite puissance de 2, 2^r, qui majore la somme des degrés
2. on construit des tableaux de tailles 2^r obtenus à partir des coefficients des polynômes (sous forme complexe) en ajoutant des 0,
3. on calcule, avec `fft`, les valeurs des polynômes aux points ω_k,
4. on calcule les produits de ces valeurs,
5. on détermine, avec `fft_inv`, le polynôme produit.

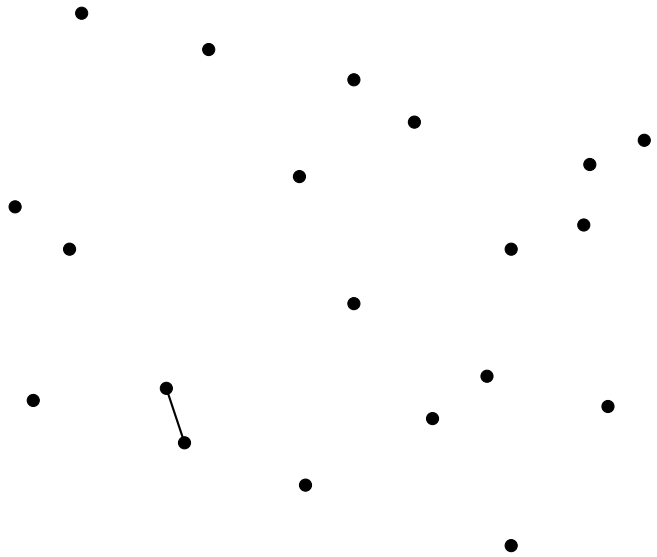
Ex. 5 Produit rapide de polynômes Écrire cette fonction.

5-5 Complexité

On note $C(r)$ le nombre d'opération effectuées par la fonction `fft` pour le traitement d'une liste de taille 2^r.
On a $C(0) = 0$ et $C(r+1) = 2C(r) + k.2^r$.
k est le nombre d'opérations effectuées lors du calcul des termes pour chaque i dans la boucle.
On pose, classiquement, $u_r = 2^{-r}.C(r)$: on a $u_{r+1} = u_r + \frac{k}{2}$.
Ainsi $u_r = \frac{k.r}{2}$ puis $C(r) = k'.r.2^r$.
La complexité de `fft_inv` est aussi un $\mathcal{O}(r.2^r)$.
Pour le produit on effectue des opération qui sont de complexité linéaire et on a $2^{r-1} < \deg(P) + \deg(Q) + 1 \leqslant 2^r$ donc la complexité est un $\mathcal{O}(m.\ln(m))$ où m est la somme des degrés des polynômes.

6 Points proches

Le but est de trouver la paire (P, Q) dans un nuage de points du plan telle que la distance de P à Q est minimale parmi toutes les distance entre points.



Les points **distincts** seront donnés sous la forme d'une liste de paires de flottants.
Par exemple les points dessinés ci-dessus sont codés sous la forme

```
let points = [(10.3, 5.4); ( 1.2, 2.5); ( 3.4, 2.7); (10.7, 2.4);
              ( 5.7, 1.1); ( 0.9, 5.7); ( 6.5, 7.8); ( 6.5, 4.1);
              (11.3, 6.8); ( 8.7, 2.9); ( 5.6, 6.2); ( 9.1, 5.0);
              ( 2.0, 8.9); ( 7.8, 2.2); ( 9.1, 0.1); ( 1.8, 5.0);
              ( 3.7, 1.8); ( 7.5, 7.1); (10.4, 6.4); ( 4.1, 8.3)];;
```

La distance entre deux points est calculée par la fonction

```
let distance p q =
  let x, y = p in
  let x', y' = q in
  sqrt ((x'-. x)**2.0 +. (y' -. y)**2.0);;
```

Ex. 6 **Premier algorithme**
Écrire une fonction `points_proches` qui prend une liste de points comme paramètre et qui renvoie une paire de points **distincts** de la liste qui réalise le minimum de distance entre points ainsi que la distance entre ces points.
Combien de distances entre points sont-elles calculées ?

6-1 Méthode diviser pour régner naïve

L'algorithme donné a une complexité quadratique.
On va utiliser une méthode diviser pour régner afin d'obtenir un algorithme de complexité quasi-linéaire.
Pour écrire cet algorithme on aura besoin de trier les points selon leur abscisse ou leur ordonnée. On suppose donnée une fonction de tri de complexité quasi-linéaire, telle que le tri-fusion qui sera étudié dans le chapitre suivant.
Ce tri est implémenté en **OCaml** par la fonction

```
List.sort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

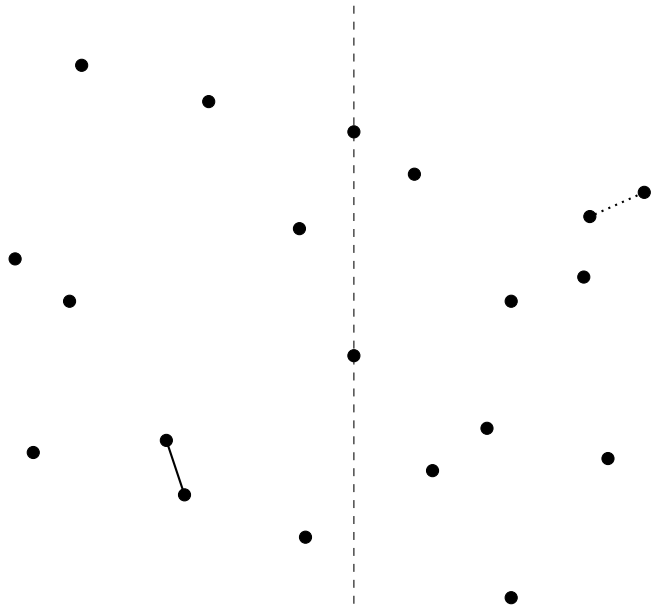
Le premier argument est une fonction de comparaison qui renvoie 0 si les deux éléments sont égaux, 1 si le premier est strictement supérieur au second et -1 sinon.

```
let compare_x p q =
   let x , y  = p in
   let x', y' = q in
   if x = x'
   then 0
   else if x > x'
        then 1
        else -1;;
```

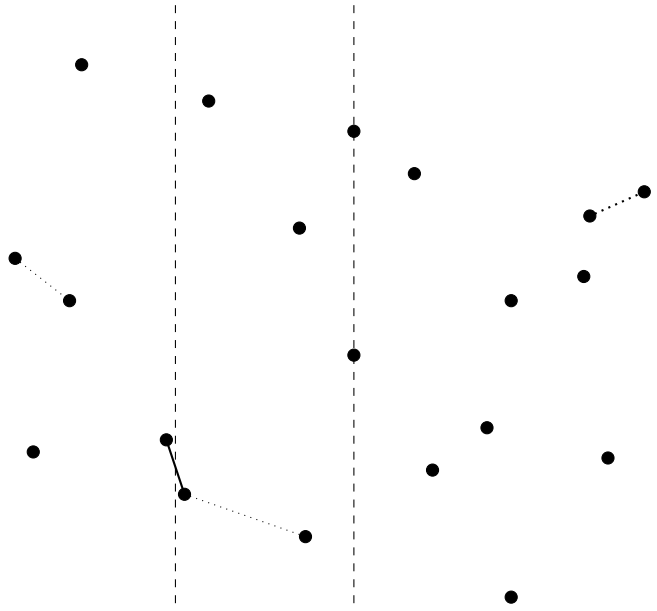
```
let compare_y p q =
   let x , y  = p in
   let x', y' = q in
   if y = y'
   then 0
   else if y > y'
        then 1
        else -1;;
```

Pour déterminer la paire de points de distance minimale on propose l'algorithme suivant :
↪ on trie la liste des points selon les valeurs de x, la première coordonnée,
↪ on sépare en deux partie égales ou de cardinaux qui ne diffèrent que de 1,
↪ on calcule récursivement la paire de points de distance minimale dans chaque partie,
↪ on sélectionne laquelle des paires de points est la plus proche.

Voici une représentation de l'étape finale pour l'algorithme ci-dessus.
On a calculé les minimums de chaque coté et on trouve bien le minimum global.



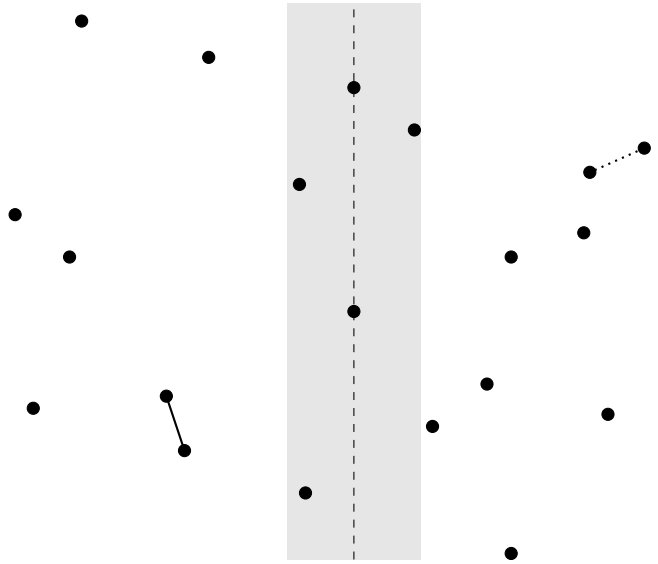
Le problème est qu'en agissant de la sorte on n'a pas toujours le bon résultat : en effet la distance minimale peut être atteinte pour deux points de chaque coté de la séparation. C'est le cas de l'exemple lors du second appel récursif de la partie gauche.



Le minimum de gauche n'est pas calculé, on aurait eu comme résultat le minimum des deux distances pointillées à gauche.
Il faut donc rassembler les résultats plus finement.

6-2 Bande centrale

On note x_0 une valeur de séparation : les points de gauche ont une abscisse inférieure ou égale à x_0 et les points de droite ont une abscisse supérieure ou égale à x_0. Dans la suite on prendra pour valeur de x_0 l'abscisse du premier point, p_0, de la partie de droite : `p0 = List.hd l2` si `l1, l2` est le résultat de `moities liste`.
On note δ le minimum de deux distances minimales entre points de la partie gauche et entre points de la partie droite. C'est un candidat à la distance minimale mais il peut exister une distance plus petite entre points de chaque coté de la frontière entre les deux parties.
On peut remarquer que, pour un tel couple de points,
les abscisses appartiennent à $[x_0 - \delta; x_0 + \delta]$.



On va donc extraire les points de la bande $[x_0 - \delta; x_0 + \delta] \times \mathbb{R}$ et on cherche la distance minimale. Il faut cependant éviter de retrouver une complexité quadratique : tous les points pourraient se retrouver dans cette bande.
Pour chercher les distances minimales à partir d'un point (x, y) on cherche les distance avec les successeurs (x', y') , c'est-à-dire $y' \geqslant y$, en s'arrêtant lorsque $y' \geqslant y + \delta$ car on ne trouvera plus de points à une distance plus proche de (x, y) que δ.

On peut donc écrire l'algorithme.
On commence par trier, une fois pour toute, la liste selon les abscisses.
On applique alors, récursivement, le calcul des points proches d'une liste

1. On sépare la liste.

Ex. 7 Écrire une fonction `moities liste n` qui reçoit une liste (de taille n) et qui renvoie deux liste `l1` et `l2` telles que `liste = l1 @ l2`, `l1` est de taille $n \div 2$ et `l2` de taille $n - n \div 2$.

2. On calcule les points proches de chaque partie, avec la distance.

3. On calcule le minimum des deux distances, δ, et on détermine la bande autour du point de séparation, de largeur δ.

Ex. 8 Écrire une fonction `bande liste p0 delta` qui reçoit une liste de points, un point et un flottant et qui renvoie la liste des éléments de cette liste qui vérifient que leur abscisse appartient à l'intervalle $[x_0 - \delta; x_0 + \delta]$ où x_0 est l'abscisse de `p0` (et δ est la valeur de `delta`).

4. On trie la bande selon l'ordonnée.

5. On recherche la paire de points proches de la bande avec une distance inférieure à δ, s'il en existe.

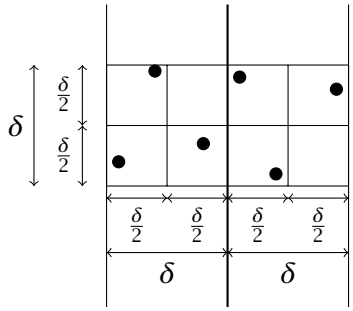
Ex. 9 Écrire une fonction `points_proches_bande liste delta` qui prend une liste de points et un flottant comme paramètres et qui renvoie `Some p q d` si p et q sont des points de la bande à distance minimale d avec $d \leqslant \delta$ ou `None` si un tel triplet n'existe pas.

6. On conclut en comparant les résultats.

Ex. 10 Écrire une fonction `points_proches liste` qui prend une liste de points comme paramètre et qui renvoie une paire de points **distincts** de la liste qui réalise le minimum de distance entre points ainsi que la distance entre ces points.

6-3 Complexité

On va maintenant prouver que l'on a bien amélioré la complexité.
La clé est de constater que la fonction `points_proches_bande` ne fait pas beaucoup de tests.



On a séparé les points en deux parties et on a calculé la valeur de δ.
On cloisonne la bande en carrés de cotés $\frac{\delta}{2}$; la distance minimale entre deux points d'un même carré est la diagonale, $\frac{\sqrt{2}\delta}{2}$, qui inférieure à δ. Comme chaque carré est inclus dans une partie (droite ou gauche) les distances entre deux points est au moins δ.
On en déduit que chaque carré contient au plus 1 point.
Ainsi chaque point de la bande est comparé à 8 points au plus.
Ainsi, si on note $C(n)$ le nombre de calculs de distances dans la recherche des points dans la fonction récursive, on a $C(n) \leqslant C(n_1) + C(n_2) + 8m$.
où n_1 et n_2 sont les nombres de points à gauche et à droite $n_1 = n \div 2$ et $n_2 = n - n_1$
et m est le nombre de points dans la bande, $m \leqslant n$ (la bande peut contenir les n points).
On a $C(2) = 1$ et, pour $n \leqslant 2^p$, on a $n_1 \leqslant 2^{p-1}$ et $n_2 \leqslant 2^p$.
On prouve donc classiquement que $C(n) \leqslant 8.p.2^p$ pour $n \leqslant 2^p$ ce qui donne une complexité en $\mathcal{O}(n.\ln(n))$ en choisissant p tel que $2^{p-1} < n \leqslant 2^p$.
On a bien une complexité quasi-linéaire en nombre de calculs de distances.

Cependant ce calcul est trompeur : en effet on a estimé le nombre de calculs de distances mais le tri de la bande pourrait introduire de nombreuses comparaisons qui n'ont pas été prises en compte.
On note $K(n)$ le nombre maximal d'opérations et de comparaisons dans l'exécution de la fonction auxiliaire pour des listes de taille n.

↪ Dans la fonction `bande` on fait au plus n comparaisons,
↪ Dans la fonction `point` on fait au plus 8 comparaisons,
↪ Dans la fonction `meilleur` on fait au plus 1 comparaison,
↪ Dans la fonction `points_bande` on appelle `point` et `triplet` pour chaque point de la liste, on fait au plus $9n$ comparaisons.

On a donc $K(n) \leqslant K(n_1) + K(n_2) + 1 + n + \alpha.n\log_2(n) + 9n + 1$
où $\alpha.n\log_2(n)$ est un majorant de la complexité du tri.
Si on note k_p un majorant de $K(n)$ pour $2^{p-1} < n \leqslant 2^p$ on a $k_p \leqslant 2k_{p-1} + \alpha p.2^p + 10.2^p + 2$.
On pose $u_p = 2^{-p}k_p$: $u_p \leqslant u_{p-1} + \alpha p + 10 + 2^{1-p}$.
$u_0 = 0$ donne alors $u_p \leqslant \alpha\frac{p(p+1)}{2} + 10p + 2$
d'où $K(n) \leqslant 2^p(\alpha\frac{p(p+1)}{2} + 10p + 2) = \mathcal{O}(n.\ln^2(n))$.
Ajouter le tri initial, en $\mathcal{O}(n.\ln(n))$, ne change pas l'ordre de complexité.

Ex. 11 Comment peut-on assurer une complexité en $\mathcal{O}(n.\ln(n))$?
On pourra utiliser la fusion du tri fusion.

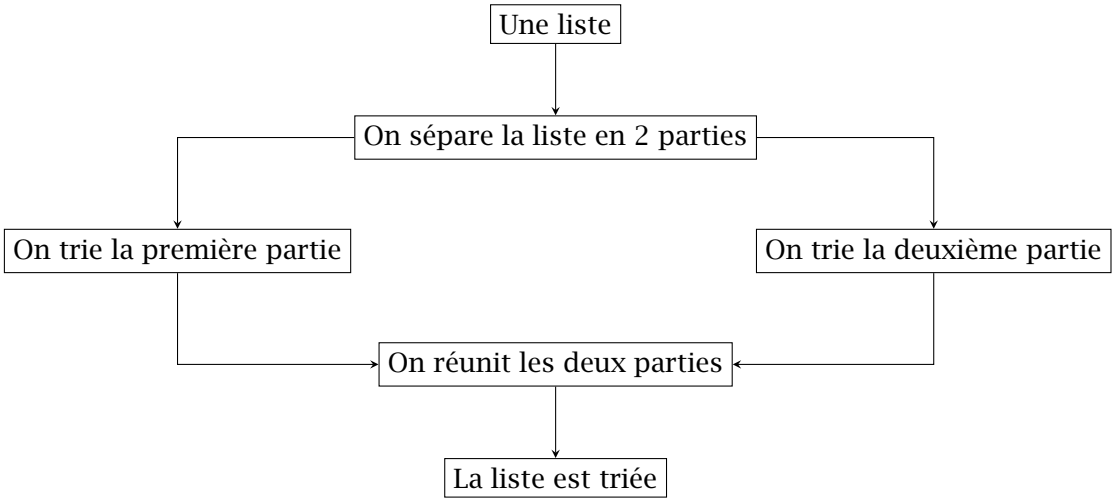
Chapitre V

Tris rapides

Nous avons étudié deux tris.
↪ Pour le tri par sélection, l'algorithme peut se simplifier en on ajoute le minimum devant le reste trié.
↪ Pour le tri par insertion, l'algorithme peut se simplifier en on insère un élément dans le reste trié.

On peut rassembler les deux algorithmes sous la forme
1. on sépare un élément du reste : le plus petit dans le cas du tri par sélection, le premier dans le cas du tri par insertion
2. on trie les éléments restants de manière récursive
3. on ajoute l'élément séparé : c'est automatique dans le cas du tri par sélection, on l'insère dans le cas du tri par insertion.

On peut généraliser en utilisant une méthode diviser pour régner :
1. On sépare en deux parties.
2. On trie chaque partie de manière récursive
3. On assemble les deux parties triées.




```
let rec tri ensemble =
  if casTerminal
  then traitement
  else let e1, e2  = separation ensemble in
       let t1 = tri e1 in
       let t2 = tri e2 in
       assembler t1 t2;;
```

Il serait idéal de trouver un tri pour lequel séparation et assemblage sont simples mais cela semble impossible. Nous allons proposer deux tris : pour l'un la séparation est immédiate

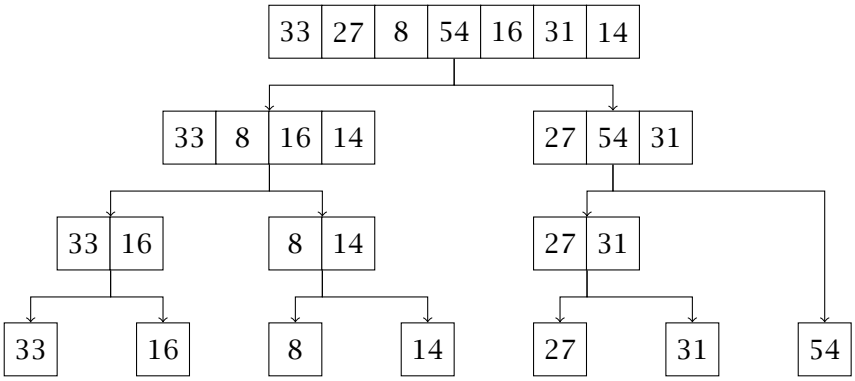
mais l'assemblage est difficile, c'est le tri-fusion, pour l'autre la séparation sera la partie coûteuse, c'est le tri pivot.

	Séparation facile, assemblage difficile	Séparation difficile, assemblage facile
Un singleton et le reste	Tri par insertion	Tri par sélection
Deux parties tailles quelconques	Tri fusion	Tri pivot

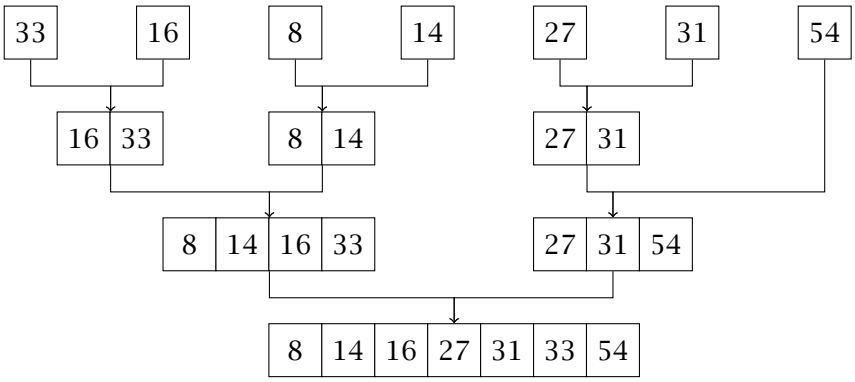
1 Tri fusion

Le tri-fusion sépare les données en deux parts presque égales, sans critère.
Dans le cas des listes on peut écrire la séparation, par exemple, en envoyant alternativement les éléments dans les listes qui recoivent les éléments ; la complexité est linéaire.

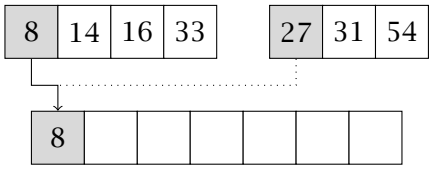
```
let rec separation liste =
  match liste with
  |[] -> [],[]
  |[x] -> [x],[]
  |t::(tt::q) -> let l1,l2 = separation q in
                (t::l1), (tt::l2);;
```

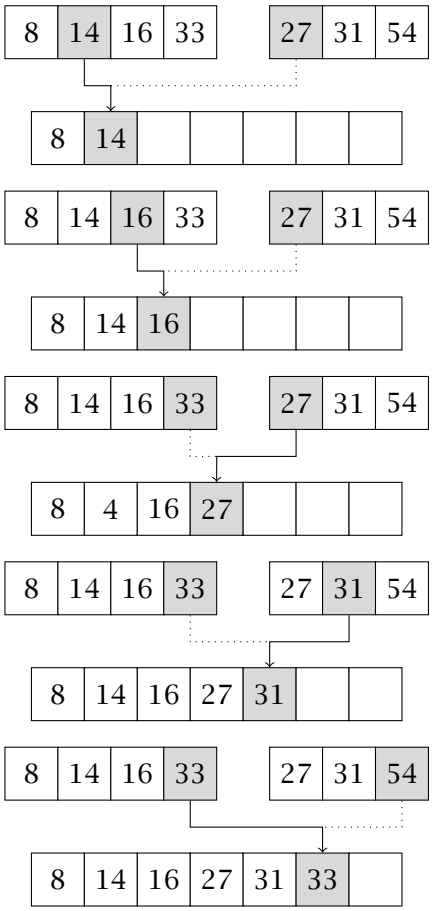


Il reste à écrire la fusion de deux liste ordonnées.

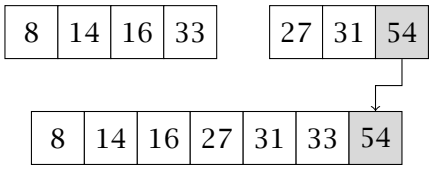


Pour cela on place, dans l'ordre, les éléments des deux listes en choisissant le plus petit des éléments non encore placé, il est en tête des restes des listes car les deux listes sont triées.





Pour le dernier cas il n'y a plus de choix.




```
let rec fusion l1 l2 =
  match l1,l2 with
  |[],_ -> l2
  |_,[] -> l1
  |t1::q1,t2::q2 -> if t1 < t2
                   then t1::(fusion q1 l2)
                   else t2::(fusion l1 q2);;
```

Code V.1 Fusion de listes triées

La fonction de tri découle de l'introduction.

```
let rec triFusion liste =
  match liste with
  |[] -> []
  |[x] -> [x]
  |_ -> let l1, l2 = separation liste in
        let lt1 = triFusion l1 in
        let lt2 = triFusion l2 in
        fusion lt1 lt2;;
```

Code V.2 Tri fusion

2 Analyse du tri fusion

Terminaison

La fonction `separation` termine pour des listes de taille 0 ou 1 et, sinon, est appelée récursivement en diminuant la taille de 2. Elle termine donc en un temps fini.
Dans la fonction `fusion` la somme des longueurs des deux listes diminue strictement lors d'un appel récursif, c'est un variant donc la fonction termine.
La fonction `triFusion` termine car elle fait appel à des fonctions qui terminent et, lors des appels récursifs, reçoit une liste de longueur strictement inférieure comme paramètre.

Preuve

La fonction de tri doit renvoyer une liste avec les mêmes éléments placés dans l'ordre croissant.
Il faut donc prouver que `separation` renvoie deux liste qui contiennent exactement tous les éléments de la liste initiale ; cela se fait par une récurrence simple.
On prouve ensuite par récurrence sur la somme des tailles des listes que `fusion` appliquée à deux listes triées renvoie une liste triée avec les mêmes éléments.

- C'est évident si la somme des longueur est nulle car alors les deux listes sont vides et `fusion` renvoie une liste vide, qui est triée.
- On suppose prouvé le résultat dans le cas de listes dont la somme des tailles est n.

On considère deux listes de somme de longueurs égale à $n+1$.
↪ Si l'une est vide la fonction renvoie l'autre qui est triée et contient tous les éléments.
↪ Si aucune n'est vide alors l'élément en tête est un minorant de chaque liste et le minimum des deux têtes est un minorant de l'union.
On suppose `l1 = t1::q1` et `l2 = t2::q2` avec t1 <= t2.
On applique la fonction récursivement à `q1` et `t2::q2` dont la somme des tailles est n donc l'hypothèse de récurrence implique que le résultat est trié et contient tous les éléments de l'union, c'est-à-dire tous les éléments de `l1` et `l2` sauf `t1`. En ajoutant `t1` en tête on obtient bien tous les éléments du départ dans une liste triée car `t1` est un minorant et le reste est trié.

La preuve du tri se fait alors par récurrence (généralisée) sur la taille des listes à trier.

Complexité

↪ La séparation ne comporte aucune comparaison
↪ La fusion effectue une comparaison par étape tant que les deux listes ne sont pas vides. Le dernier élément placé n'est pas comparé donc la fusion effectue au plus $n-1$ comparaisons.
↪ La séparation d'une liste de taille n donne deux listes de tailles n_1 et n_2 ; de plus, si on a $n \leqslant 2^p$ alors $n_1 \leqslant 2^{p-1}$ et $n_2 \leqslant 2^{p-1}$.
On note u_p le nombre maximal de comparaisons effectuées lors du tri d'une liste de taille 2^p au plus, on a alors $u_p \leqslant u_{p-1} + u_{p-1} + n - 1 \leqslant 2u_{p-1} + 2^p$. et $u_0 = 0$
Si on pose $v_p = 2^{-p}u_p$ on a $v_p \leqslant v_{p-1} + 1$ et $v_0 = 0$ donc $v_p \leqslant p$ puis $u_p \leqslant 2^p.p$.
Pour une liste de taille n avec $2^{p-1} < n \leqslant 2^p$, le nombre de comparaisons dans le tri est majoré par $p.2^p \leqslant (\log_2(n)+1).2n$.

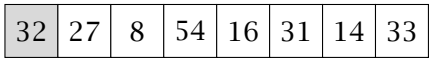
Complexité du tri fusion

Le tri fusion d'une liste de taille n a une complexité en $\mathcal{O}(n\log_2(n))$ dans le pire des cas.

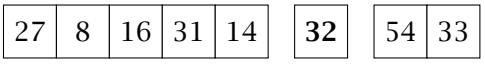
3 Tri pivot (ou tri rapide)

Le tri pivot (QUICKSORT en anglais) inverse la difficulté : plutôt que fusionner deux sous-listes triées qui viennent d'un découpage arbitraire il découpe en séparant "petits" et "grands" ; pour cela on sélectionne les termes plus petits qu'un terme choisi dans la liste, le pivot, dans une liste et les termes plus grands dans une autre.
Le choix le plus simple pour le pivot est le premier terme de la liste.

La séparation de la liste suivante



peut donner, avec le premier terme comme pivot,




```
let rec filtrage liste pivot =
  match liste with
  |[] -> [], []
  |t::q -> let petit, grand = filtrage q pivot in
           if t < pivot
           then t::petit, grand
           else petit, t::grand;;
```

Code V.3 Filtrage de liste

La fonction de tri découle de l'introduction.

```
let rec triPivot liste =
  match liste with
  |[] -> []
  |t::q -> let l1, l2 = filtrage q t in
           let lt1 = triPivot l1 in
           let lt2 = triPivot l2 in
           lt1@(t::lt2);;
```

Code V.4 Tri pivot

4 Analyse du tri pivot

Terminaison

La longueur de la liste diminue à chaque appel de `filtrage` donc la fonction termine en un temps fini.
Lors de l'appel récursif dans le tri pivot, les liste filtrées ont une longueur strictement inférieure car elles ne contiennent pas le terme pivot. La terminaison se prouve alors par récurrence sur la longueur de la liste.

Preuve de `filtrage`

On prouve par récurrence sur la longueur de la liste que `filtrage liste pivot` renvoie deux listes `petit` et `grand` telles que `petit` contient les éléments de la liste originale qui sont strictement inférieurs au pivot avec la même multiplicité et `grand` contient les éléments de la liste originale qui sont supérieurs ou égaux au pivot avec la même multiplicité.

↪ C'est évident dans le cas d'une liste vide.

↪ On suppose que la propriété est vraie pour les liste de taille n.
`liste = t::q` est de taille $n+1$
Lors de l'exécution de `filtrage liste pivot`, l'appel récursif de `filtrage q pivot` concerne une liste de taille n. L'hypothèse de récurrence implique que `filtrage q pivot` renvoie deux liste qui contiennent tous les éléments de `q` avec la même multiplicité et qui sont séparés par le pivot. On ajoute alors l'élément manquant de la liste initiale dans la liste adaptée ce qui prouve la propriété pour les liste de taille $n+1$.

Preuve de `triPivot`

La fonction de tri doit renvoyer une liste avec les mêmes éléments placés dans l'ordre croissant : (*). C'est bien entendu le cas quand la liste initiale est vide.
On suppose que la propriété (*) est vérifiée pour toutes les listes de taille inférieure ou égale à n. On considère une liste de taille $n+1$: `liste = t::q`.

1. `filtrage q t` renvoie deux liste `l1` et `l2` telles que `l1` contient les éléments de `q` qui sont strictement inférieurs à `t` avec la même multiplicité et `l2` contient les éléments de `q` qui sont supérieurs ou égaux à `t` avec la même multiplicité.
2. D'après l'hypothèse de récurrence, `lt2` contient les éléments de `l2` triés. Il y a donc les éléments de `q` qui sont supérieurs ou égaux à `t` avec la même multiplicité et triés.
3. Ainsi `t::l2` contient les éléments de la liste initiale qui sont supérieurs ou égaux à `t` avec la même multiplicité sous forme triée.

4. D'après l'hypothèse de récurrence, `lt1` contient les éléments de `l1` triés. Il y a donc les éléments de `q` qui sont strictement inférieurs à `t` avec la même multiplicité et triés.
5. On conclut que `lt1@(t::l2)` est formée des éléments de la liste initiale avec la même multiplicité et triés.

Complexité

Dans la fonction `filtrage pivot liste` on fait une comparaison par élément de `liste` donc la complexité est la longueur de `liste`.
On note `C(liste)` le nombre de comparaisons dans le tri de `liste`.
Comme il n'y a pas de comparaisons dans l'assemblage des listes on conclut que, si `|liste|` est la longueur d'une liste et si `filtrage pivot l` renvoie les listes `lg` et `ld` on a `C([]) = 0` et `C(l) = l - 1 +C(lg) + C(ld)`. Si on suppose que `liste` est formée de n éléments en ordre strictement croissant alors la fonction `filtrage t q` renverra toujours le couple `[], q`. La formule de la complexité devient alors `C(t::q) = liste - 1 + 0 + C(q)` donc la complexité se calcule
`C(liste)` $= \sum_{k=1}^{n} k-1 = \frac{n(n-1)}{2}$, on aboutit au maximum du nombre de comparaisons car c'est le nombre de parties à 2 éléments.

Complexité du tri pivot

Le tri pivot d'une liste de taille n a une complexité en $\mathcal{O}(n^2)$ dans le pire des cas et une complexité en $\mathcal{O}(n\log_2(n))$ en moyenne.

5 Exercices

Complexité du tri-fusion

Dans l'algorithme du cours de tri par fusion on note C_n le nombre maximal de comparaisons nécessaire pour trier une liste de n éléments.

Ex. 1 a) Montrer que $C_n = C_{\lfloor n/2 \rfloor} + C_{\lceil n/2 \rceil} + n - 1$ avec $C_1 = 0$.
b) On pose $D_n = C_{n+1} - C_n$; prouver que $D_n = D_{\lfloor n/2 \rfloor} + 1$ avec $D_1 = 1$.
c) Prouver que $C_n = n\lfloor \log_2(n) \rfloor + n + 1 - 2^{\lfloor \log_2(n) \rfloor + 1}$.

Complexité moyenne du tri pivot

Définition
$C(\ell)$ est le nombre de comparaisons effectuées lors du tri de la liste ℓ.
On a $C([\,]) = 0$ et $C(\ell) = |\ell| - 1 + C(\ell_g) + C(\ell_d)$ où ℓ_g et ℓ_g sont les sous-listes de gauche et de droite obtenues après le filtrage de ℓ.
Pour calculer la complexité moyenne on va considérer que les différentes listes triées sont les permutations d'une liste initiale $[a_1; a_2; \cdots; a_n]$ triée ; on note $\ell_\sigma = [a_{\sigma(1)}; a_{\sigma(2)}; \cdots; a_{\sigma(n)}]$
La complexité moyenne est donc $\overline{C}(n) = \dfrac{1}{n!} \displaystyle\sum_{\sigma \in S_n} C(\ell_\sigma)$; on pose $\overline{C}(0) = 0$.
Cette complexité moyenne est indépendante des éléments $a_1 < a_2 < \cdots < a_n$.
Regroupement
Pour $\sigma(1) = k$, le filtrage de ℓ_σ crée deux listes $\ell_{\sigma,g}$ et $\ell_{\sigma,d}$ avec $\ell_{\sigma,g}$ qui est une permutation de $[a_1; a_2; \cdots; a_{k-1}]$ et $\ell_{\sigma,d}$ qui est une permutation de $[a_{k+1}; a_{k+2}; \cdots; a_n]$.
Selon les permutations les positions initiales des éléments de $\ell_{\sigma,g}$ et $\ell_{\sigma,d}$ sont distribuées dans les positions 2 à n.
Pour toute permutation σ on définit
↪ $I_-(\sigma) = \{i \in \{2, \ldots, n\} \ ; \ \sigma(i) < \sigma(1)\} = \sigma^{-1}(\{1, 2, \ldots, \sigma(1) - 1\})$,
c'est l'ensemble des positions des éléments plus petits que $\sigma(1)$ et
↪ $I_+(\sigma) = \{i \in \{1, 2, \ldots, n\} \ ; \ \sigma(i) > \sigma(1)\} = \sigma^{-1}(\{\sigma(1) + 1, \ldots, n\})$,
c'est l'ensemble des positions des éléments plus grands que $\sigma(1)$.

$(I_-(\sigma), I_+(\sigma))$ définit une partition de $\{2, 3, \ldots, n\}$ qu'on note $E_{1,n}$.
Toute partie A de $E_{1,n}$ définit une partition $(A, \widehat{A})$ avec $\widehat{A} = E_{1,n} \setminus A$ et on pose
$S_n(A) = \{\sigma \in S_n \ ; \ I_-(\sigma) = A, \ I_+(\sigma) = \widehat{A}\}$; on a $S_n = \displaystyle\bigcup_{A \subset E_{1,n}} S_n(A)$

Ex. 2 Prouver que $|S_n(A)| = (k-1)!(n-k)!$ pour $|A| = k-1$.

Action du filtrage
La fonctions de filtrage appliquée à ℓ_σ définit des bijections de A vers $\{1, 2, \ldots, k-1\}$ et de $\widehat{A}$ vers $\{k+1, \ldots, n\}$ qui sont indépendantes de $\sigma \in S_n(A)$ car le filtrage ne dépend que de la distribution des petites et des grandes valeurs relativement à k donc de A et B.
Ainsi la liste $\ell_{\sigma,g}$ ne dépend que de la restriction de σ à A et la liste $\ell_{\sigma,d}$ ne dépend que de la restriction de σ à $\widehat{A}$.

Ex. 3 Prouver que $\sum_{\sigma \in S_n(A)} C(\ell_{\sigma,g}) = (n-k)!(k-1)!\overline{C}(k-1)$ et
$\sum_{\sigma \in S_n(A)} C(\ell_{\sigma,d}) = (k-1)!(n-k)!\overline{C}(n-k)$.
En calculant les valeurs de $\sum_{\sigma \in S_n,\ \sigma(1)=k} C(\ell_{\sigma,g})$ et $\sum_{\sigma \in S_n,\ \sigma(1)=k} C(\ell_{\sigma,d})$,
prouver que $\overline{C}(n) == n - 1 + \frac{1}{n}\sum_{k=1}^{n} \overline{C}(k-1) + \frac{1}{n}\sum_{k=1}^{n} \overline{C}(n-k)$.

Ex. 4 Prouver que $n\overline{C}(n) = (n+1)\overline{C}(n-1) + 2(n-1)$.
En posant $\overline{C}(n) = (n+1)v_n$, calculer $\overline{C}(n)$.

Nombre d'inversions dans une liste

Certains sites marchands font des suggestions d'achats basées sur vos achats précédents.
Un outil utile au cœur des algorithmes qui permettent ces recommandations est la comparaison des classements.
Le problème est de comparer votre classement de différents objets à celui d'autres usager pour discerner ceux qui vous ressemblent et ensuite de proposer les articles choisis par ceux qui vous ressemblent.
Une manière simple de comparer deux classements est de déterminer le nombre d'inversions : ce sont les paires d'objets qui sont classés en ordres opposés dans les deux classements.
Par exemple votre classement est $a_1 > a_2 > a_3 > a_4 > a_5$.
Quelqu'un d'autre a classé $a_2 > a_4 > a_1 > a_3 < a_5$.
Entre ces deux classements les paires (a_1, a_2), (a_1, a_4) et (a_3, a_a) ont été inversées (et elles seules). Le nombre d'inversions est donc 3.
On peut donc modéliser cela sous la forme suivante.
Soit $S = s_1, s_2, \ldots, s_n$ une liste de n entiers distincts.
On dit que les indices i et j forment une **inversion** si $i < j$ et $s_i > s_j$.

On cherche donc à dénombrer le nombre d'inversions dans une liste donnée.

Ex. 5 Écrire une solution qui utilise le tri par insertion.

Ex. 6 Écrire une version du tri-fusion qui calcule le nombre d'inversions.

Médiane

Lors d'une étude d'un ensemble de valeurs on souhaite souvent trouver une valeur représentative. On utilise souvent la moyenne à cet effet.
Cependant il y a quelques inconvénients :
↪ la moyenne n'est que rarement un élément de l'ensemble
↪ elle est sensible aux valeurs erratiques : une suite de 99 termes de valeurs 1 et 1 terme de valeur 100 aura une moyenne de 1,99, ce qui peut être peu signifiant.

Un autre outil possible est la médiane : dans une liste de n termes on cherche le terme de rang $\lfloor \frac{n+1}{2} \rfloor$.
Dans l'ensemble $\{3, 12, 5, 8, 17, 9, 13\}$ la médiane est 9 ;
dans l'ensemble $\{3, 12, 5, 8, 17, 9, 4, 13\}$ la médiane est 8 (on aurait pu choisir 9).

Pour résoudre le problème on va le généraliser :
trouver le terme de rang p dans un ensemble
Une méthode immédiate consiste à trier l'ensemble puis à déterminer le terme de rang p.
On va améliorer cela (en moyenne) en utilisant le découpage du tri pivot. En effet celui-ci partitionne un ensemble de taille n en trois sous-ensembles :
↪ le pivot,
↪ les éléments de l'ensemble strictement inférieurs au pivot, de taille k avec $0 \leqslant k \leqslant n-1$,
↪ les autres éléments (ils sont supérieurs ou égaux au pivot, il y en a $n-k-1$).

Il y a alors 3 cas selon la valeur de k :
↪ si $k = p-1$ le pivot est la valeur recherchée,
↪ si $p \leqslant k$ on cherche le p-ième élément dans le premier sous-ensemble,
↪ si $k < p-1$ on cherche le $p-1-k$-ième élément dans le second sous-ensemble.

Ex. 7 Écrire l'algorithme.

Ex. 8 Quelle est la complexité moyenne d'une recherche ?

Chapitre VI

Types abstraits

1 Introduction

1-1 Retour sur les listes et les tableaux

Nous avons, jusqu'à présent, utilisé les structures de données fournies par le langages pour maintenir un assemblage d'éléments. Ce sont les tableaux et les listes.
Ces deux structures ont des différences.

1. Quand on connaît sa position, l'accès à un élément d'un tableau se fait en temps constant, c'est-à-dire que le temps est le même pour tous les éléments.
   À l'inverse, le temps d'accès à un élément dans une liste est proportionnel à sa position.
   Cela n'induit pas de différence lorsque l'on parcourt toutes les valeurs mais devient signifiant quand on utilise la structure comme une fonction sur l'ensemble des indices.

2. Les listes sont annoncées comme ayant la possibilité d'ajouter un élément. Il est facile de fabriquer une nouvelle liste à partir d'une autre en lui ajoutant un élément.
   Ici facile signifie que l'instruction est simple, c'est `t::q`, et qu'elle se fait en temps constant, indépendant de la longueur de la liste.

   Pour ajouter un élément à un tableau, il "suffit" de tout recopier.

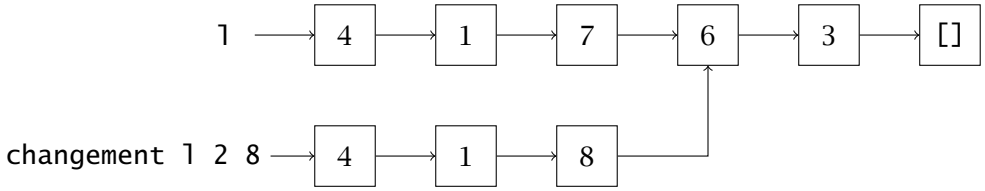
   ```
   let adjonction x tab =
     let n =Array.length tab in
     let t = make_vect (n+1) x in
     for i = 0 to (n-1)  do t.(i) <- tab.(i) done;
     t;;
   ```

   Ce qui change par rapport aux listes est la complexité temporelle, proportionnelle à la taille, et aussi la complexité spatiale car les éléments sont dédoublés.

3. On peut modifier la valeur d'un emplacement dans un tableau en conservant le même tableau (`t.(i) <- x`), la modification se fait "en place", c'est toujours le même tableau
   Par contre les listes sont **persistantes** c'est-à-dire qu'elle ne sont pas modifiables.
   Pour changer une valeur dans une liste il faut créer une nouvelle liste :

   ```
   let rec changement liste i x =
     match liste, i with
     |[], _ -> failwith "Dépassement"
     |t::q, 0 -> x::q
     |t::q, i -> t::(changement q (i-1) x);;
   ```

Les premières valeurs de la listes sont alors dédoublées.



1-2 Autres structures

Lorsque l'on résout un problème on est souvent amené à utiliser des structures de données adaptées : on utilisera souvent des listes ou des tableaux en choisissant l'outil qui permet une meilleure complexité.
Cependant il y a des types de données spécialisées qui sont utilisés régulièrement et il peut être utile de les étudier afin de les reconnaître lorsqu'ils adviennent.
Les fonctions usuelles seront alors déjà connues.
Cela permet, de plus, une meilleure portabilité des algorithmes entre différents langages.

Voici quelques structures qui sont souvent rencontrées.

- ↪ Les assemblages dynamiques de données : on veut ajouter des éléments et en enlever. Plusieurs structures seront possibles selon la manière d'enlever un élément.
- ↪ Les ensembles (au sens mathématiques) : on veut pouvoir faire des unions, intersections, différences.
- ↪ Les fonctions de A vers B quand A n'est pas un ensemble de la forme $\{0, 1, \ldots, n-1\}$: c'est la notion de dictionnaire.
- ↪ Des assemblages avec des relations possibles entre les éléments. C'est la structure très importante de graphe, elle sera étudiée en seconde année. Les arbres, qui feront l'objet d'un prochain chapitre, sont des cas particuliers de graphes.

1-3 Types de données abstraits

Nous allons, dans ce chapitre, étudier les structures en définissant leur spécification ou **type de données abstrait** et nous en donnerons souvent plusieurs implémentations. Le plus souvent ce seront les critères de complexité d'une des fonctions qui seront déterminants dans le choix d'une implémentation.

1. Le type de données abstrait décrit ce que l'on veut faire avec les données, c'est la partie interface. On y définit les noms des fonctions, leur signature ainsi que les propriétés (on parle d'axiomes) que ces fonctions doivent vérifier afin de spécifier de manière unique leur comportement.
2. L'implémentation décrit **comment** on parvient à réaliser l'interface. C'est elle qui détermine la vitesse d'exécution des fonctions définies par le type de données abstrait.

2 Assemblages dynamiques

La première structure qui nous intéresse est celle où on gère des objets. On considère donc un contenant qui peut recevoir et dont on peut extraire des objets.
Il y a ici deux possibilités.

1. On peut choisir que les opérations se feront "en place", c'est-à-dire qu'ajouter ou extraire un élément ne crée pas un nouvel objet, cela ne fait qu'en modifier le contenu.
   On dit que la structure est **impérative** : les modifications seront faites par des instructions sans résultat. C'est le cas lors de la modification de la valeur d'un tableau.
2. On peut choisir des données persistantes, dans ce cas ajouter ou extraire un élément définit une nouvelle variable, c'est le comportement des listes.
   On dit que la structure est **fonctionnelle** : les modifications seront faites par l'application d'une fonction en produisant un nouvel assemblage.

Les fonctions de bases sont
↪ créer un contenant vide,
↪ tester si un contenant est vide,
↪ ajouter un élément à un contenant,
↪ voir un des éléments sans l'enlever,
↪ enlever un élément d'un contenant.
↪ On pourra parfois utiliser une fonction qui renvoie le nombre d'éléments dans le contenant.

On différencie les structures selon la méthode d'extraction.
↪ Si on veut retirer le dernier élément ajouté, on parle de **pile**.
↪ Si on veut retirer le premier élément ajouté non encore retiré, on parle de **file**.
↪ Si chaque élément est associé à une valeur qui mesure sa priorité et qu'on veut retirer l'élément de plus grande priorité, on parle de **file de priorité**

2-1 Piles

Pile

Une pile (stack en anglais) est une collection d'objets de même nature dans laquelle on peut ajouter un élément et retirer le dernier élément ajouté.

Le principe est : dernier arrivé, premier parti (Last In First Out : **LIFO**).
Il existe de nombreux exemples de piles dans la vie courante :
↪ gestion de l'historique des pages visitées dans un navigateur,
↪ gestion de l'annulation des dernières opérations effectuées dans un logiciel,
↪ exécution d'une fonction récursive …

On aura donc besoin de créer un type `'a stack` où `'a` désigne le type des éléments.
On écrira les fonctions avec les noms usuels anglais.

↪ `createStack : 'a -> 'a stack` ; l'argument d'entrée est un modèle d'élément de la pile qui permettra dans certaines implémentations de créer la pile, sa valeur n'a pas d'importance, seul son type est signifiant.
↪ `isEmptyStack : 'a stack -> bool`.
↪ L'ajout est appelé `push`, sa signature est
   1. `'a stack -> 'a -> 'a stack` dans le cas fonctionnel,
   2. `'a stack -> 'a -> unit` dans le cas impératif.
↪ `top : 'a stack -> 'a` pour voir le dernier élément ajouté.
↪ Le retrait du dernier élément ajouté est `pop`, sa signature est
   1. `'a stack -> 'a stack` dans le cas fonctionnel,
   2. `'a stack -> unit` dans le cas impératif.

Voici les axiomes qui doivent être vérifiés par ces fonctions dans le cas d'une forme fonctionnelle ; ils imposent des conditions concernant la compositions des fonctions.
1. `isEmptyStack (createStack u)` renvoie `true`.
2. `isEmptyStack (push pile x)` renvoie `false`.
3. `top (createStack u)` renvoie une erreur.
4. `top (push pile x)` renvoie `x`.
5. `pop (createStack u)` renvoie une erreur.
6. `pop (push pile x)` renvoie `pile`.

Les axiomes, dans le cas d'une forme impérative, imposent des conditions sur la successions des instructions.
1. `let pile = createStack u; isEmptyStack pile` renvoie `true`.
2. `push pile x; isEmptyStack pile` renvoie `false`.
3. `let pile = createStack u; top pile` renvoie une erreur.
4. `push pile x; top pile` renvoie `x`.
5. `let pile = createStack u; pop pile` renvoie une erreur.
6. `push pile x; pop pile` est neutre, on peut supprimer les deux instructions.

Implémentation fonctionnelle par des listes

Les listes sont une implémentation fonctionnelle immédiate des piles.

```
let createStack u = [];;
let isEmptyStack pile = pile = [];;
let push pile x = x::pile;;
let top = List.hd;;
let pop = List.tl;;
```

On pouvait, bien entendu, écrire les fonctions de listes. Par exemple

```
let top pile =
  match pile with
  |[] -> failwith "La pile est vide"
  |t::q -> t;;
```

Implémentation impérative par des listes

Si on veut modifier l'assemblage en place on peut encapsuler une liste dans un enregistrement. Cela revient en fait à créer une référence de liste.

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

let createStack u ={contenu = []};;

let isEmptyStack pile = pile.contenu = [];;

let push pile x = pile.contenu <- x::(pile.contenu);;

let top pile = List.hd pile.contenu;;

let pop pile =
  pile.contenu <- List.tl pile.contenu;;
```

Code VI.1 Piles impératives

On remarquera que, dans les deux cas, les opérations prennent un temps constant.

Implémentation impérative par des tableaux

On peut aussi utiliser un tableau en maintenant la position de la position libre. Les opérations seront plus simples mais la pile aura une taille maximale : il faudra gérer la possibilité d'une pile pleine.
La valeur de la taille sera maintenu dans une variable globale qu'on pourra modifier avant la création; cependant il est conseillé de ne pas utiliser cette variable dans les autres fonctions.

```
let taillePile = 500;; (* Par exemple *)

type 'a stack =
  {mutable taille : int; contenu : 'a array};;

let createStack u =
  {contenu = Array.make taillePile u; taille = 0};;

let isEmptyStack pile = pile.taille = 0;;
```

```
let push pile x =
  let n = pile.taille in
  if n < Array.length pile.contenu
  then (pile.contenu.(n) <- x; pile.taille <- n+1)
  else failwith "La pile est remplie";;

let top pile =
  let n = pile.taille in
  if n = 0
  then failwith "La pile est vide"
  else pile.contenu.(n-1);;

let pop pile =
  let n = pile.taille in
  if n = 0
  then failwith "La pile est vide"
  else pile.taille <- (n-1);;
```

Code VI.2 Piles impératives bornées

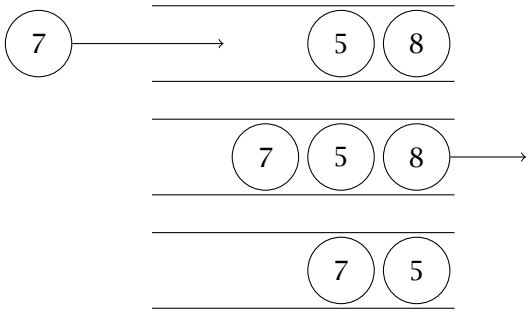
Ici encore les opérations prennent un temps constant, à l'exception de la création.

2-2 Files

File

Une **file** (queue en anglais) est une collection d'objets de même nature dans laquelle on peut ajouter un élément et retirer l'élément qui a été ajouté le premier parmi ceux qui restent.

Le principe est que le premier arrivé sera premier parti (First In, First Out : **FIFO**).
C'est le principe des files d'attente dans la vie courante.
En informatique ce même principe est dénommé buffer (tampon).

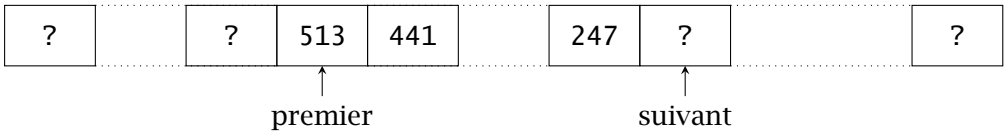


On se restreint ici à une structure impérative. Les fonctions sont donc
↪ `createQueue : 'a -> 'a queue`,
↪ `isEmptyQueue : 'a queue -> bool`,
↪ l'ajout est `enque : 'a queue -> 'a -> unit`,
↪ `first : 'a queue -> 'a`
permet de voir le premier élément ajouté parmi ceux qui restent,
↪ le retrait du premier élément ajouté parmi ceux qui restent est
`deque : 'a queue -> unit`.

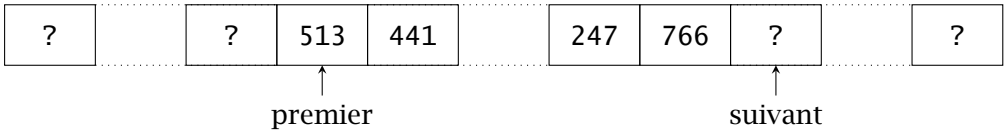
Les axiomes doivent décrire le comportement de `first` et `deque` :
1. `let f = createQueue u in isEmptyQueue f` renvoie `true`.
2. `enque f x; isEmptyQueue f` renvoie `false`.
3. `let f = createQueue u in first f` renvoie une erreur.
4. `enque f x; first f` renvoie `x` **si `f` est la file vide**
   et a le même effet que `first f ; enque f x` sinon.
5. `let f = createQueue u in deque f` renvoie une erreur.
6. `enque f x; deque f` a le même effet que `deque f ; enque f x` **si `f` n'est pas vide** et, sinon, remet la file dans un état vide.

Implémentation par un tableau simple

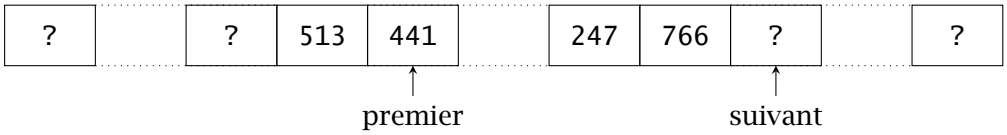
La première idée pour implémenter les files est celle des tickets qui sont fournis par un distributeur et qui donnent l'ordre de passage.
↪ Il y a un nombre fini de numéros possibles : c'est représentable par un tableau.
↪ Le prochain numéro à appeler est maintenu par l'indice du premier élément qui sera vu ou extrait.
↪ Le prochain numéro à attribuer est maintenu par l'indice où placer un élément à insérer.



Si on ajoute 766 on aboutit à



Si on retire un élément on arrive à



On remarquera que les élément sortis sont toujours dans le tableau.
De plus, lorsqu'on a ajouté le nombre maximal d'élément, la file est remplie mais quand on les retire tous on arrive à la situation assez paradoxale d'une file vide et pleine à la fois ; en revenant à l'analogie du distributeur de tickets, c'est le cas quand tous les tickets ont étés distribués et tous ont été appelés.

Le type est un enregistrement avec 3 champs :

```
type 'a queue = {contenu : 'a array;
                 mutable premier : int;
                 mutable suivant : int};;
```

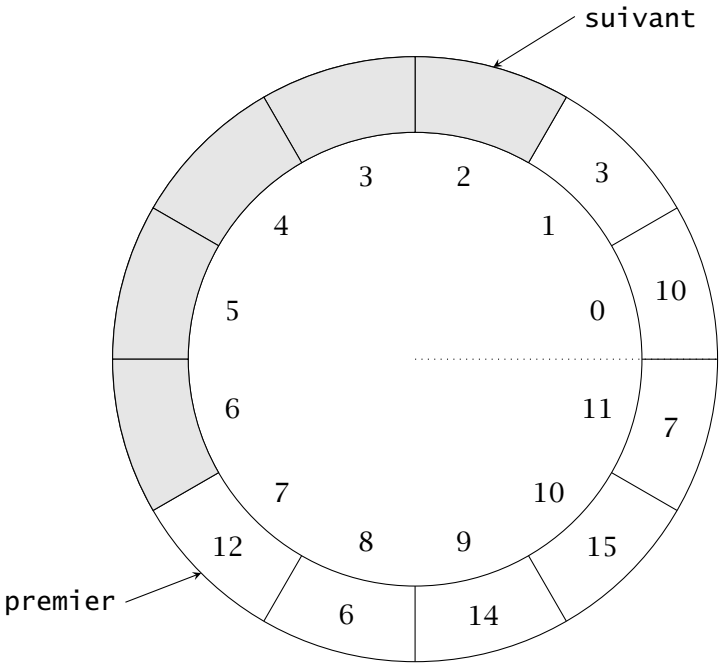
Le cas vide et plein décrit ci-dessus correspond à la situation où `premier` et `suivant` sont tous deux égaux à la taille de `contenu`.

```
let tailleFile = 500;;

let createQueue u = {contenu = Array.make tailleFile u;
                     premier = 0;
                     suivant = 0};;

let isEmptyQueue file =
  file.premier = file.suivant;;
```

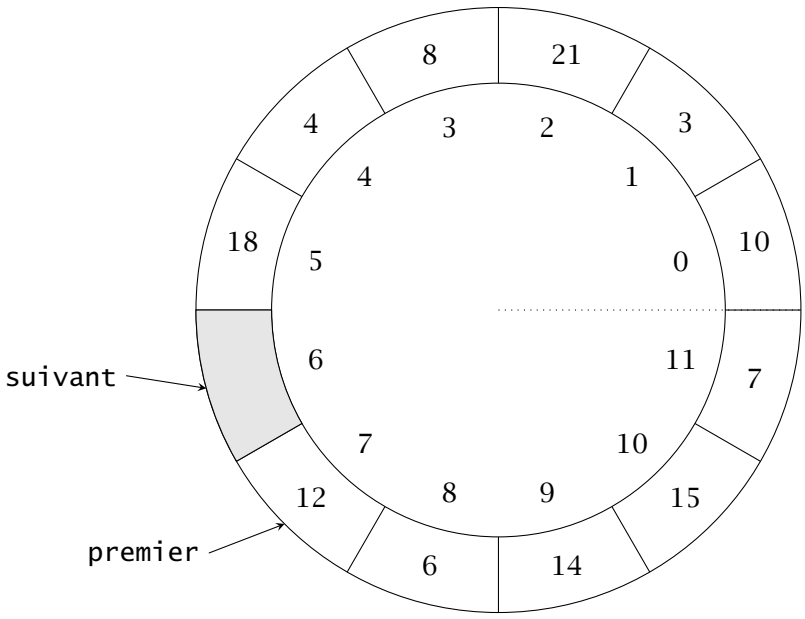
```
let enque file x =
  let n = file.suivant in
  if n < Array.length file.contenu
  then (file.contenu.(n) <- x;
        file.suivant <- n + 1)
  else failwith "File remplie";;

let first file =
  if isEmptyQueue file
  then failwith "File vide"
  else let n = file.premier in
       file.contenu.(n);;

let deque file =
  if isEmptyQueue file
  then failwith "File vide"
  else let n = file.premier in
       file.premier <- n+1;;
```

Code VI.3 Files impératives limitées

Implémentation par un tableau circulaire



La solution au problème décrit ci-dessus est de recharger le distributeur de tickets.
Cela correspond à l'idée de revenir à l'indice 0 quand on incrémente `premier` et `suivant` lorsqu'ils sont égaux à l'indice maximum.
Pour cela il suffit de remplacer `k + 1` par `(k + 1) mod longueur`.
On fait "boucler" le tableau sur lui-même.
Le cas d'une file vide correspond à `premier = suivant`, comme lors de l'initialisation. Le cas d'une file pleine devrait correspondre à la situation où toutes les valeurs sont occupées mais alors on aurait aussi `premier = suivant`. Pour pouvoir différentier le cas vide du cas plein on "sacrifie" une possibilité d'occupation en testant `premier = (suivant + 1) mod longueur`. La représentation ci-dessous est pleine.



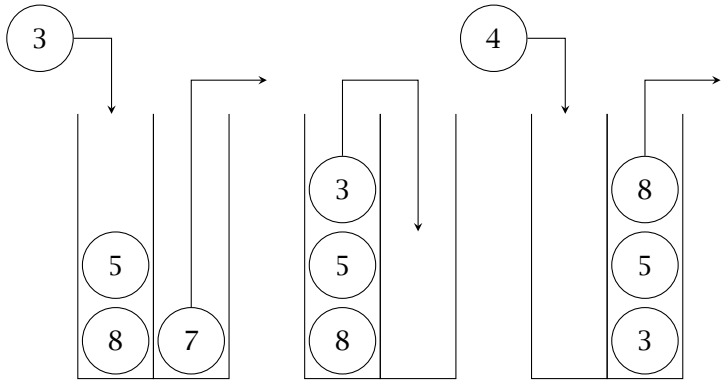

```
let tailleFile = 500;;

type 'a queue = {contenu : 'a array;
                 mutable premier : int;
                 mutable suivant : int};;

let createQueue u = {contenu = Array.make tailleFile u;
                     premier = 0;
                     suivant = 0};;

let isEmptyQueue file = file.premier = file.suivant;;
```

```
let enque file x =
  let n = Array.lenfth file.contenu in
  let s = file.suivant in
  let p = file.premier in
  if p <> (s + 1) mod n
  then begin file.contenu.(s) <- x;
             file.suivant <- (s + 1) mod n end
  else failwith "La file est pleine";;

let first file =
  if isEmptyQueue file
  then failwith "La file est vide"
  else let p = file.premier in
       file.contenu.(p);;

let deque file =
  if isEmptyQueue file
  then failwith "La file est vide"
  else let p = file.premier in
       let n = Array.length file.contenu in
       file.premier <- (p+1) mod n;;
```

Code VI.4 Files impératives circulaires

Implémentation par deux piles

Ce qui empêche d'implémenter la structure de file à l'aide d'une liste est que l'ajout et le retrait d'un élément se font à deux positions opposées de la liste : il faudrait parcourir toute la pile et la reconstruire soit pour ajouter les éléments à la fin de liste soit pour aller chercher l'élément final.
L'idée est d'employer 2 piles : une pour accumuler les éléments en entrée, l'autre pour contenir les éléments en sortie. Quand la seconde est vide mais que la première ne l'est pas on retourne la première dans la deuxième.



La complexité dans le pire des cas de `retirer` est linéaire car retourner une liste a une complexité proportionnelle à la longueur de la liste.
Cependant chaque élément ajouté n'intervient que pour 1 dans la complexité du retournement. Quand on ajoute et retire n éléments la complexité totale est linéaire donc la complexité pour chaque action est constante : on parle de **complexité amortie.**
On utilise la fonction de retournement d'une liste : `List.rev`.

```
type 'a queue = {mutable entree : 'a list;
                 mutable sortie : 'a list};;

let createQueue u = {entree = [] ; sortie = []};;

let isEmptyQueue file =
  file.entree = [] && file.sortie = [];;
```

```
let enque file x =
  file.entree <- x::(file.entree);;

let renverse file =
  file.sortie <- List.rev file.entree;
  file.entree <- [];;

let first file =
  if isEmptyQueue file
  then failwith "La file est vide"
  else (if file.sortie = [] then renverse file;
        List.hd file.sortie );;

let deque file =
  if isEmptyQueue file
  then failwith "File vide"
  else (if file.sortie = [] then renverse file;
        file.sortie <- List.tl file.sortie);;
```

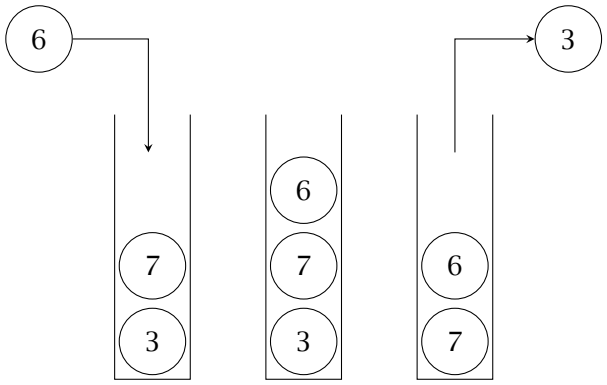
Code VI.5 Files impératives avec deux listes

2-3 Files de priorité

File de priorité

Une **file de priorité** est une collection d'objets de même nature, chacun étant associé à un entier, sa priorité, dans laquelle on peut ajouter un élément et retirer l'élément qui a la plus grande priorité (ou la plus petite priorité).

C'est le principe, par exemple, des traitements des urgences dans hôpital ou du traitement des tâches dans un Operating System (Linux, Mac OSX, Windows).
Dans l'exemple ci-dessous la propriété est la valeur.



Les fonctions, dans le cas d'une une structure impérative avec priorité minimale, sont
↪ `createPQ : 'a -> 'a pQueue`,
↪ `isEmptyPQ : 'a pQueue -> bool`,
↪ l'ajout est `add : 'a pQueue -> 'a -> int -> unit` (l'entier est la priorité),
↪ `next : 'a pQueue -> 'a * int`
permet de voir l'élément de priorité minimale et sa priorité,
↪ le retrait de l'élément prioritaire est `remove : 'a queue -> unit`.

Le comportement peut être décrit par l'axiome de détermination du suivant.
Si `next fileP` renvoie `(x, n)` et si on exécute `add fileP y m; next fileP` alors
↪ si $n < m$, la valeur renvoyée est `(x, n)`
↪ si $n \geqslant m$, la valeur renvoyée est `(y, m)`

Nous allons implémenter cette structure sous forme impérative.
On emploiera une liste ou un tableau qui pourra être
1. trié, l'ajout se fera par insertion
2. ou non trié, le retrait se fera en recherchant le terme de priorité minimale.

Les fonctions ont une complexité constante sauf
1. `add` dans le cas d'un assemblage trié qui a une complexité en $\mathcal{O}(n)$,
2. `next` et `remove` dans le cas d'un assemblage non trié qui ont une complexité en $\mathcal{O}(n)$,

où n est le nombre d'éléments dans la file d'attente.
On verra en deuxième année un type de données concret qui permet une complexité en $\mathcal{O}(\ln(n))$ pour toutes les fonctions où n est le nombre d'éléments dans la file d'attente.

```
let nMax = 500;;

type 'a pQueue =
  {mutable taille : int; contenu : ('a*int) array};;

let createPQ u =
  {contenu = Array.make nMax (u,0); taille = 0};;

let isEmptyPQ fileP = fileP.taille = 0;;
```

```
let add fileP x prio =
  let n = fileP.taille in
  if n < Array.length fileP.contenu
  then (fileP.contenu.(n) <- (x, prio);
        fileP.taille <- n+1)
  else failwith "La file est remplie";;

let indiceMin tableau borne =
  (* Le tableau est supposé non vide *)
  (* La borne est strictement inférieure à la longueur *)
  let indMin = ref 0 in
  let prioMin  = ref (snd tableau.(0)) in
  for i = 1 to borne do
    let (x,n) = tableau.(i) in
    if n < !prioMin then (prioMin := n; indMin := i) done;
  !indMin;;
```

Code VI.6 Files de priorité avec un tableau non trié 1

```
let next fileP =
  let n = fileP.taille in
  if n = 0
  then failwith "La file est vide"
  else let i = indiceMin fileP.contenu (n-1) in
       fileP.contenu.(i);;

let remove fileP =
  let n = fileP.taille in
  if n = 0
  then failwith "La file est vide"
  else let i = indiceMin fileP.contenu (n-1) in
       fileP.contenu.(i) <- fileP.contenu.(n-1);
       fileP.taille <- n-1;;
```

Code VI.7 Files de priorité avec un tableau non trié 2

```
type 'a pQueue = {mutable contenu : ('a * int) list};;

let createPQ u = {contenu = []};;

let isEmptyPQ fileP = fileP.contenu = [];;

let rec insere (x,n) liste =
  match liste with
  |[] -> [(x,n)]
  |(y,p)::q when n <= p -> (x,n)::liste
  |t::q -> t::(insere (x,n) q);;

let add fileP x prio =
  fileP.contenu <- insere (x,prio) fileP.contenu;;

let next fileP = List.hd fileP.contenu;;

let remove fileP =
  fileP.contenu <- List.tl fileP.contenu;;
```

Code VI.8 Files de priorité avec une liste triée

2-4 Exercices

Implémentations

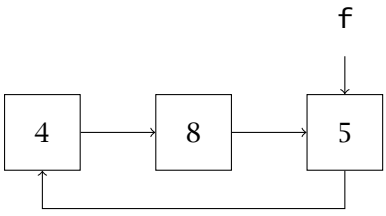
Ex. 1 **Redimensionnement de tableaux**
Écrire une fonction `resize` qui reçoit un tableau de taille notée n renvoie un tableau de taille double dont les n premiers éléments sont ceux du paramètre.
En déduire une écriture des fonctions d'une pile qui utilise les tableaux et qui n'a pas de taille maximale.

Ex. 2 **Axiomes fonctionnels de files**
Donner les axiomes dans le cas d'une forme fonctionnelle de file.

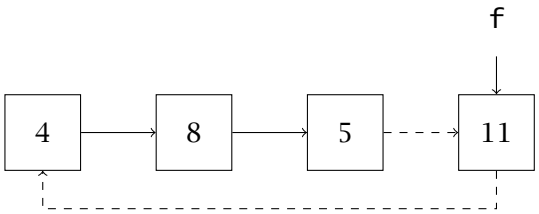
Ex. 3 **Fonctions de files avec un tableau circulaire** Dans les files définies par un tableau "circulaire" on peut ajouter un champs booléen qui indique si la file est vide.
Le cas où les deux indices sont égaux signifiera alors que la file est pleine.
Donner une écriture possible pour les fonctions dans cette implémentation.

Files par chaînage circulaire

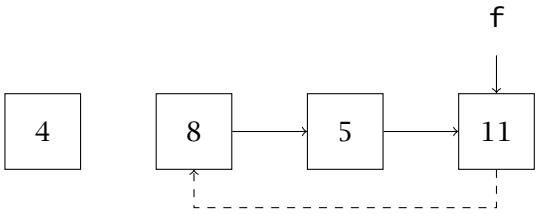
Pour obtenir une complexité constante dans les files on a vu qu'on pouvait utiliser une paire de listes. Nous allons proposer ici une implémentation plus directe qui reprend le principe des listes : des cellules chaînées. Le principe est d'associer à chaque élément son successeur dans l'ordre d'insertion en convenant d'associer le premier élément inséré au dernier : on "boucle" la liste.
La file est accessible par le dernier élément inséré.
↪ Dans le cas de la file 4, 8, 5 (ajoutés dans cet ordre) on obtient



↪ Pour ajouter un élément, on crée une cellule avec sa valeur. Elle est associée à l'élément final et le précédent élément lui est maintenant associé. On "casse" donc la liaison entre l'ancien dernier élément et le premier. Voici le résultat de `enque f 11`



↪ Pour voir ou enlever le premier élément de la file il suffit de suivre le lien.
Voici le résultat de `deque f`



Implémentation

1. Les cellules seront représentées par le type

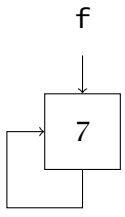
```
type 'a cellule = {valeur : 'a ; mutable suivant : 'a cellule};;
```

2. Pour gérer la possibilité d'une file vide on utilisera un type optionnel

```
type 'a queue = 'a cellule opt;;
```

`None` est la pile vide,
`Some c` est la file associée à une cellule `c`.

3. Une file qui n'a qu'un élément doit être associée à elle-même ; on devra la définir avec le mot-clé `rec` : `let rec c = {valeur = x; suivant = c}`.



Ex. 4 Type fonctionnel pour les files
Donner une écriture des fonctions de piles dans cette implémentation.

Applications

Ex. 5 Parenthésage
On souhaite déterminer les parenthèses associées dans une chaîne de caractères supposée bien parenthésée. Le résultat doit être la liste des couples d'indices qui correspondent à des parenthèses associées. Par exemple, le résultat pour l'entrée `"(1+((2-3)*(4+5)))"` peut être `[(0,16), (3, 15), (10, 14), (4, 8)]`
Écrire une fonction `parentheses` en utilisant une pile avec les fonctions associées.

Ex. 6 Génération des nombres en binaire
Pour créer les chaînes de caractères correspondant aux n premiers entiers on peut utiliser une file d'attente que l'on initialise avec `"1"`.
Ensuite on répète n fois :
1. on lit le premier élément, noté `ch`,
2. on ajoute dans la file `ch^"0"` (concaténation de 0 à droite),
3. on ajoute dans la file `ch^"1"`,
4. on ajoute `ch` dans la liste,
5. on vide le premier terme de la file

Écrire une fonction qui reçoit un entier et renvoie la liste des écritures binaires.

3 Dictionnaires

Dictionnaire

Un dictionnaire est une structure de données où l'on stocke des éléments dotés d'une clé, de sorte à pouvoir extraire l'élément correspondant à une clé donnée.

C'est donc une fonction définie depuis un ensemble de clés vers un ensemble.
On peut citer de nombreux exemples issus de la vie courante :

↪ un dictionnaire au sens usuel,
↪ un annuaire téléphonique,
↪ de manière générale, toute base de données.

Les dictionnaires seront définis à partir de deux types, un pour la clé, l'autre pour la valeur : on définira le type `'a 'b dict`.
On veut pouvoir utiliser les fonctions (on utilise un type impératif)
↪ `dicoVide : 'a*'b -> 'a 'b dict` pour créer un dictionnaire vide,
↪ `ajouter : 'a 'b dict -> 'a -> 'b -> unit`
pour ajouter un élément ou remplacer la valeur si la clé est déjà définie,
↪ `defini : 'a 'b dict -> 'a -> bool` pour tester s'il existe un élément de clé donnée,
↪ `association : 'a 'b dict -> 'a -> 'b`
pour déterminer la valeur associée à une clé dans le dictionnaire.

Dans le cas où on a besoin de parcourir le dictionnaire on pourra définir aussi
↪ `estVide : 'a 'b dict -> bool` pour tester si le dictionnaire est vide,
↪ `enlever : 'a 'b dict -> 'a -> unit` pour retirer le couple associé à une clé,
↪ `dict_to_list : 'a 'b dict -> ('a * 'b) list`
pour convertir le dictionnaire en liste des éléments (on peut omettre la clé).

Ex. 7 Implémentation par liste d'association
On peut choisir de stocker les couples dans une liste ; on parle de liste d'association.
`type ('a,'b) dict = mutable contenu : ('a*'b) list;;`
Écrire les fonctions pour cette implémentation. Quelle est la complexité ?

Ex. 8 Implémentation naïve par tableaux
Lorsque les clés sont entières et appartiennent à $\{0,1,2,...,nMax-1\}$ on peut représenter le dictionnaire par un tableau dont les éléments sont des options du type `'b`.
On définira le type `type 'b dict = ('b option) array;;`.
Écrire les fonctions pour cette implémentation. Quelle est la complexité ?

3-1 Tables de hachage

Si on dispose d'une fonction `f` **injective** depuis l'ensemble des valeurs possibles des clés vers un ensemble de la forme $\{0,1,2,...,N-2,N-1\}$ on peut réaliser un dictionnaire avec la méthode ci-dessus.
On devra simplement stocker le couple `(cle, valeur)` à la position `f clé`.

L'inconvénient de cette implémentation provient de la nécessité d'avoir une fonction injective. Pour s'assurer de cette propriété il est souvent indispensable d'avoir une fonction qui prend des valeurs très grandes : le tableau défini risque alors d'être à la fois très volumineux et très peu rempli.

On va donc abandonner l'exigence d'injectivité. La fonction sera appelée **fonction de hachage** et la structure employée est une **table de hachage**.
Bien entendu le problème est qu'alors deux clés peuvent être envoyées vers la même valeur : on parle de **collision**.
Il existe des techniques efficaces pour résoudre les conflits créés par les collisions :

- ↪ L'**adressage ouvert** : pour effectuer une insertion, on sonde la table de hachage jusqu'à trouver une place vide dans laquelle placer la clé, l'ordre de sondage peut être déterminé par une fonction de hachage améliorée,
- ↪ le **chaînage** : on place dans une même liste (d'association) tous les éléments envoyés vers la même valeur. On combine ainsi les deux implémentations vues ci-dessus.

Nous allons implémenter le chaînage.
Nous ne discuterons pas ici du choix (difficile) de la fonction de hachage : la principale exigence est d'obtenir une répartition équilibrée des valeurs dans l'ensemble $\{0, 1, 2, \ldots, N-2, N-1\}$. Le plus souvent on construit une fonction à valeurs entières puis on calcule son résultat modulo N.
Remarque la fonction `a mod b` de **OCaml** renvoie un résultat négatif quand `a` est négatif (mais le reste est majoré par `b` en valeur absolue). Il faudra redéfinir la fonction

```
let modulo a b =
  if a >= 0
  then a mod b
  else b + (a mod b);;
```

La fonction de hachage sera incluse dans la donnée.

```
type ('a, 'b) dict =  {h : 'a -> int;
                       contenu : ('a * 'b) list array};;
```

Dans la création on passera la fonction et la taille du tableau en paramètres.

```
let dicoVide f nMax =
   {h = f; contenu = Array.make nMax []};;

let ajouter dico cle valeur =
  let i = dico.h cle in
  let rec auxPlus liste =
    match liste with
    |[] -> [(cle, valeur)]
    |(k, x)::q when k = cle -> (cle, valeur) :: q
    |(k, x)::q -> (k, x) :: (auxPlus q) in
  dico.contenu.(i) <- auxPlus dico.contenu.(i);;
```

Code VI.9 Tables de hachage 1

```
let defini dico cle=
  let i = dico.h cle in
  let rec auxCh liste =
    match liste with
    |[] -> false
    |(k, x)::q when k = cle -> true
    |(k, x)::q -> auxCh q in
  auxCh dico.contenu.(i);;

let association dico cle =
  let i = dico.h cle in
  let rec auxLire liste =
    match liste with
    |[] -> failwith "Clé non trouvée"
    |(k, x)::q when k = cle -> x
    |(k, x)::q -> auxLire q in
  auxLire dico.contenu.(i);;
```

Code VI.10 Tables de hachage 2

Ex. 9 Fonctions supplémentaires

Écrire les fonctions `estVide`, `enlever` et `dict_to_list`

Ex. 10 Recherche des couples de somme donnée

On veut chercher les couples d'éléments d'un tableau `tab` dont la somme est x.
Le résultat est une liste de couples d'indices i et j tels que
$i < j$ et `tab.(i) + tab.(j) = x`.
Pour `tab = [5; 12; 14; 6; 1; 9; 18]`,
`trouver_paires tab 15` doit renvoyer `[(3, 5); (2, 4)]`.

1. Proposer une fonction simple ; quelle est sa complexité ?
2. On suppose qu'on dispose d'une fonction `tri` qui trie la liste avec une complexité en $\mathcal{O}(n\ln(n))$: peut-on améliorer la complexité de la recherche ?
3. Proposer une fonction de complexité linéaire en utilisant une table de hachage.

Chapitre VII

Arbres

1 Arbres binaires (homogènes)

1-1 Définition

Les listes sont un type récursif obtenu en attachant une liste à un élément ; on a vu qu'on pouvait alors parcourir tous les éléments en détachant l'élément de tête et en continuant sur la lista attachée.
Nous allons optimiser ce procédé en attachant **deux** structures récursives à un élément, ses **fils**. Le gain sera possible si on sait, à chaque moment, lequel des deux fils contient l'élément que l'on recherche ; ce sera le cas des arbres binaires de recherche en seconde année et dans d'autres structures.
Nous verrons qu'une généralisation simple permet de définir une structure qui représente de nombreuses situations de calcul.
L'usage est de représenter graphiquement les arbres avec la racine en haut.

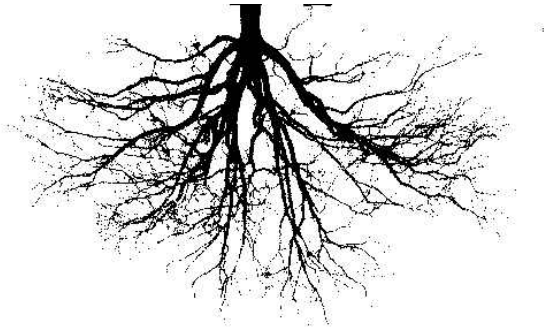


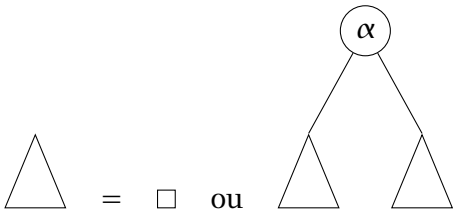
Figure VII.1 Représentation d'un arbre

Nous allons donc définir une structure avec des bifurcations doubles.

Arbre binaire de type α

Une arbre de type α est
1. soit l'arbre vide, ou **feuille**
2. soit un **nœud** formé d'une racine de type α, d'un fils droit et d'un fils gauche tous deux étant des arbres binaire de type α.

Si on représente une feuille par □ et un arbre par  , on a

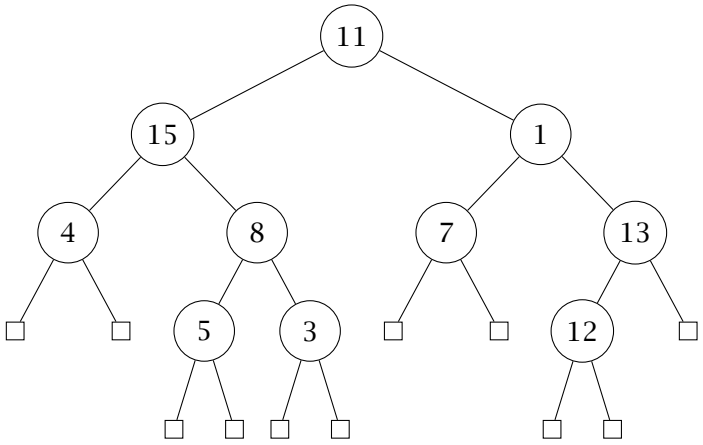


La définition se traduit directement en type récursif :

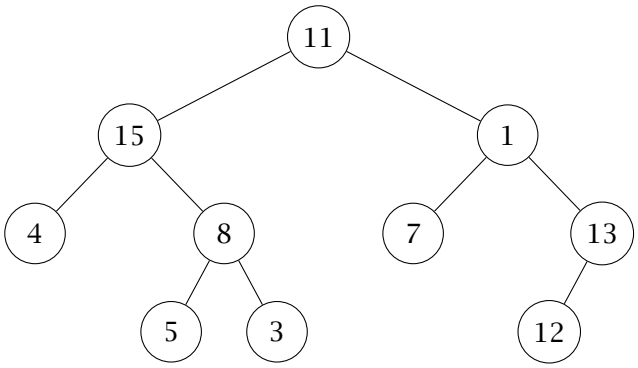
```
type 'a arbre = F | Noeud of 'a arbre * 'a * 'a arbre;;
```

Exemple

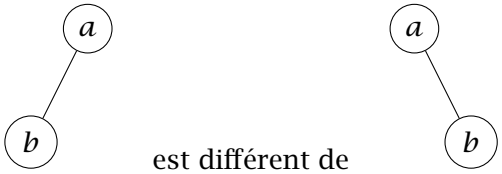
```
let a = Noeud(Noeud(Noeud(F, 4, F),
                    15,
                    Noeud(Noeud(F, 5, F), 8, Noeud(F, 3, F)))
              11,
              Noeud(Noeud(F, 7, F),
                    1,
                    Noeud(Noeud(F, 12, F), 13, F)));;
```



Le plus souvent, on ne représentera pas les feuilles.



On notera qu'alors on doit distinguer, pour les nœuds avec un seul fils non vide, le cas où c'est le fils gauche qui est vide du cas où c'est le fils droit qui est vide.



On peut écrire les fonctions qui renvoient respectivement la racine, le fils droit et le fils gauche d'un arbre.

```
let racine arbre =
  match arbre with
  |F -> failwith "L'arbre est une feuille"
  |Noeud (_,r,_) -> r;;
```

```
let filsG arbre =
  match arbre with
  |F -> failwith "L'arbre est une feuille"
  |Noeud (g,_,_) -> g;;
```

```
let filsD arbre =
  match arbre with
  |F -> failwith "L'arbre est une feuille"
  |Noeud (_,_,d) -> d;;
```

Dans la pratique le filtrage permettra souvent de ne ne pas utiliser ces fonctions.

1-2 Dénombrements

Taille

La taille d'un arbre est le nombre de nœuds qui le composent.

```
let rec taille a =
   match a with
   |F -> 0
   |Noeud (g,_,d) -> 1 + taille g + taille d;;
```

Profondeur

La profondeur d'un nœud ou d'une feuille est le nombre total de relations de parenté entre la racine et lui.

1. La racine est à une profondeur de 0, ses fils ont une profondeur de 1.
2. La profondeur des fils d'un nœud est égale à la profondeur du nœud augmentée de 1.
3. La profondeur n'un nœud dans un arbre ne se calcule pas facilement dans un langage de programmation ; en effet, rien n'est prévu pour "remonter" jusqu'à la racine.

Ex. 1 Décoration par la profondeur
Écrire une fonction `profondeur : 'a arbre -> ('a * int) arbre` qui transforme un arbre en ajoutant la profondeur aux nœuds.

Hauteur

La hauteur d'un arbre non vide, c'est-à-dire non réduit à une feuille, est la profondeur maximale de ses nœuds.
Par convention la hauteur d'un arbre vide est -1.

```
let rec hauteur a =
   match a with
   |F -> -1
   |Noeud (g,_,d) -> 1 + max (hauteur g) (hauteur d);;
```

Nombre de feuilles

Le nombre de feuilles d'un arbre est égal à la taille augmentée de 1.

Démonstration

Elle peut se faire par récurrence (généralisée) sur la taille.

1. *Si la taille est nulle l'arbre est une feuille, il y a bien* $0+1$ *feuille.*
2. *On suppose que la propriété est vraie pour les arbres de taille* $n-1$ *au plus avec* $n \geqslant 1$. *On considère un arbre de taille* n *; sa taille est au moins 1 donc l'arbre est un nœud : il admet un fils droit, de taille* n_d*, et un fils gauche, de taille* n_g*. On a* $n = n_d + n_g + 1$ *donc* $n_d < n$ *et* $n_g < n$*. On peut appliquer l'hypothèse de récurrence donc il y a au total* $n_d + 1 + n_g + 1 = n + 1$ *feuilles.*

La propriété est donc vraie pour tout n*.*

Encombrement

La taille n et la hauteur h d'un arbre binaire vérifient

$$h+1 \leqslant n \leqslant 2^{h+1}-1$$
$$\lfloor \log_2(n+1) \rfloor - 1 \leqslant h \leqslant n-1$$

Démonstration

Pour un arbre vide on a $h=-1$ *et* $n=0$ *: les inégalités sont vraies.*

On suppose dans la suite que l'arbre est non vie.

On note N_k *le nombre de nœuds à la profondeur* k *d'un arbre.*

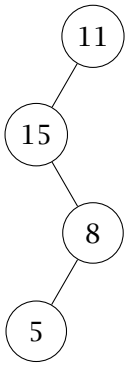
Chaque nœud à la profondeur k *admet au plus 2 nœuds fils (il peut admettre 1 ou deux feuilles comme fils) et chaque nœud à la profondeur* $k+1$ *est le fils d'un nœud à la profondeur* k *: on en déduit que* $N_{k+1} \leqslant 2N_k$*.*

Le seul nœud à la profondeur 0 est la racine (l'arbre est non vide) donc $N_0 = 1$. *On en déduit qu'on a* $N_k \leqslant 2^k$ *donc le nombre de nœuds vérifie* $n = \sum_{k=0}^{h} N_k \leqslant \sum_{k=0}^{h} 2^k = 2^{h+1} - 1$.

Il existe un nœud à la profondeur h, *d'où* $N_h \geqslant 1$.
Comme on a $N_h \leqslant 2N_{h-1}$ *on doit avoir* $N_{h-1} \geqslant 1$, *par récurrence descendante sur* k *on a* $N_k \geqslant 1$ *pour tout* k *compris entre* 0 *et* h. *On a ainsi* $n = \sum_{k=0}^{h} N_k \geqslant \sum_{k=0}^{h} 1 = h + 1$.

La deuxième partie découle de la première.

L'inégalité $h \leqslant n-1$ est une égalité lorsque, dans l'arbre, tous les nœuds ont un fils vide, sauf le dernier, à la profondeur $n-1$ qui a obligatoirement deux fils vides. C'est un cas dégénéré qui produit un arbre semblable à une liste : les nœuds se suivent les uns après les autres.
C'est la situation qu'on veut éviter ; on utilise les arbres en partant de la racine et on parvient à un élément après au plus h accès à un nœud, si on a h de l'ordre de n on n'a rien gagné par rapport à une liste chaînée.
Par contre le cas où h est de l'ordre de $\log_2(n)$ est plus intéressant : si on sait où chercher tous les éléments sont accessibles en un temps quasi-constant.



1-3 Induction structurelle

On peut souvent démontrer des propriétés sur les arbres en utilisant une démonstration par récurrence portant sur la taille (ou sur la hauteur) ; c'est ce qui a été proposé ci-dessus.
Il peut être utile de théoriser ce procédé en donnant une méthode de démonstration qui ne fasse appel qu'à la structure des arbres sans utiliser une mesure de ceux-ci (taille ou hauteur).

Induction structurelle

$\mathcal{P}(a)$ est une propriété portant sur les arbres.
↪ Si $\mathcal{P}(f)$ est vraie pour tout feuille f,
↪ et si $\mathcal{P}(a)$ est vraie dès lors que $\mathcal{P}(g)$ et $\mathcal{P}(d)$ sont vraies, pour tout arbre a de la forme `Noeud(g,x,d)`,
↪ alors $\mathcal{P}(a)$ est vraie pour tout arbre a.

On peut ainsi reprendre la démonstration sur le nombre de feuilles.
1. *Si l'arbre est une feuille, dont de taille 0, il y a bien* $0+1$ *feuille.*
2. *On suppose que deux arbres* g *et* d*, de tailles respectives* n_g *et* n_h*, ont respectivement* n_g+1 *et* n_d+1 *feuilles.*
   L'arbre `Noeud(g,x,d)` *est de taille* $n = n_d + n_g + 1$ *et il admet pour feuilles les feuilles de* g *et de* d *donc il admet* $n_d + 1 + n_g + 1 = n + 1$ *feuilles.*

La propriété est démontrée par induction structurelle.

Ce type de démonstration permet une rédaction moins lourde mais elle est parfois délicate à mettre en œuvre ; il est impératif d'énoncer avec soin la propriété portant sur les arbres que l'on veut prouver.
N.B. On peut noter que l'on peut aussi utiliser l'induction structurelle pour les listes : si une propriété portant sur les listes
↪ est valide pour la liste vide et
↪ est valide pour une liste `t::q` dès qu'elle est valide pour `q`

alors elle est vérifiée pour toutes les listes.

2 Parcours d'un arbre homogène

Parcourir un arbre homogène consiste à accéder à chaque élément une fois et une seule.
Cela permet de définir des fonctions portant sur tous les éléments de l'arbre et de transformer la structure bidimensionnelle de l'arbre en une structure linéaire.
Il y a plusieurs types de parcours selon que l'on privilégie la hiérarchie de la hauteur (parcours en largeur) ou celle provenant de l'ordre des fils, dans ce dernier cas on différenciera encore selon la position de la racine.
Pour les programmes en Caml on supposera qu'on a une fonction `traiter` qui effectue une action sur la valeur de la racine.

2-1 Parcours en profondeur

Pour parcourir (ou traiter) un arbre binaire en profondeur on choisit de parcourir le fils gauche avant le fils droit à chaque hauteur. Il reste à traiter la racine.
Il y a trois possibilités.
↪ Parcours préfixe : on traite la racine puis les fils.
↪ Parcours infixe : on traite le fils gauche puis la racine puis le fils droit.
↪ Parcours postfixe (ou suffixe) : on traite les fils puis la racine.

```
let rec infixe a =
  match a with
  |F -> ()
  |Noeud (g,r,d) -> infixe g;
                    traiter r;
                    infixe d;;
```

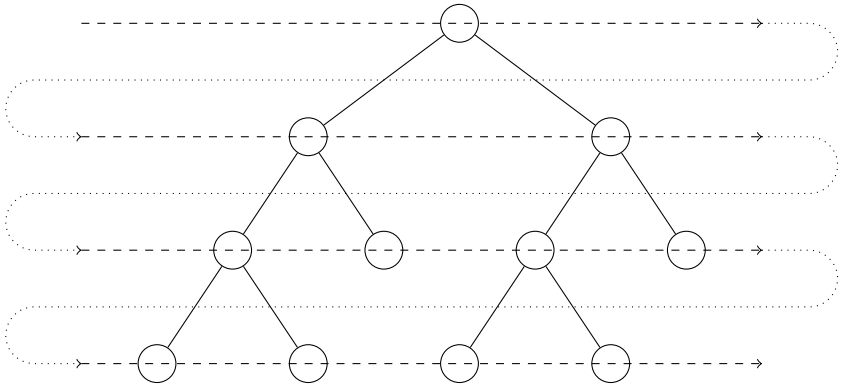
Code VII.1 Parcours infixe d'un arbre

`traiter` est une fonction de résultat `unit` qui traite chaque nœud.

Ex. 2 **Parcours préfixe et postfixe**
Écrire les fonctions de traitement d'un arbre par les parcours préfixe et postfixe.

2-2 Parcours en largeur

Dans le parcours en largeur, on visite d'abord la racine puis ses fils (les nœuds de profondeur 1) puis les nœuds de profondeur 2 et ainsi de suite.
À profondeur égale, on visite les nœuds de gauche à droite.



Si on ne veut pas lire la structure de l'arbre plusieurs fois on doit mettre en attente les fils des nœuds de profondeur k pour les traiter après avoir traité tous les nœuds de profondeur k. Pour cela on utilise une file.
On utilise le module `Queue` de OCaml qui implémente une file impérative de type `Queue.t` dont les fonctions utilisées sont

```
↪ Queue.create : unit -> 'a t
↪ Queue.add : 'a -> 'a t -> unit
↪ Queue.take : 'a t -> 'a
↪ Queue.is_empty : 'a t -> bool
```

```
let parcoursProf a =
  let file = Queue.create () in
  Queue.add a file;
  while not (Queue.is_empty file) do
    match (Queue.take file) with
   |F -> ()
   |Noeud (g,r,d) -> traiter r;
                     Queue.add g file;
                     Queue.add d file done;;
```

Code VII.2 Parcours en largeur d'un arbre

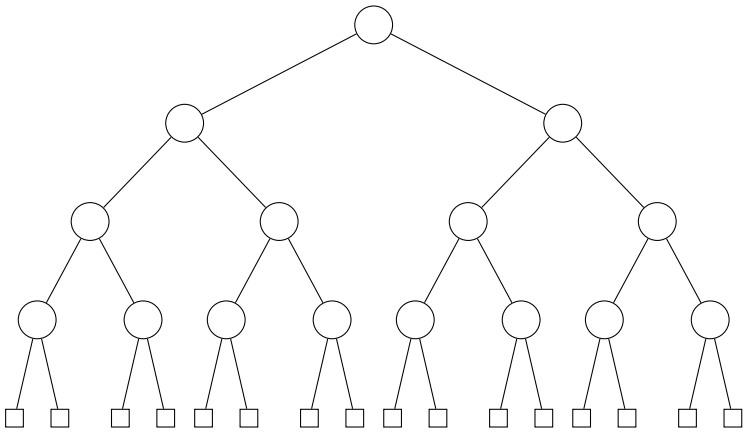
3 Équilibre d'arbres homogènes

Nous allons, dans cette partie, étudier des cas particuliers inverses du cas d'un arbre liste où la hauteur h est majorée par $k\log_2(n)$ où k est une constante.
Les propriétés sont donc celles de la structure de l'arbre, on ne représentera pas la valeur des nœuds.

3-1 Complétude

Arbre complet

Une arbre de hauteur h est complet si toutes ses feuilles sont à une profondeur $h+1$.



Il y a d'autres caractérisations possibles d'un arbre complet, c'est-à-dire des définitions alternatives.

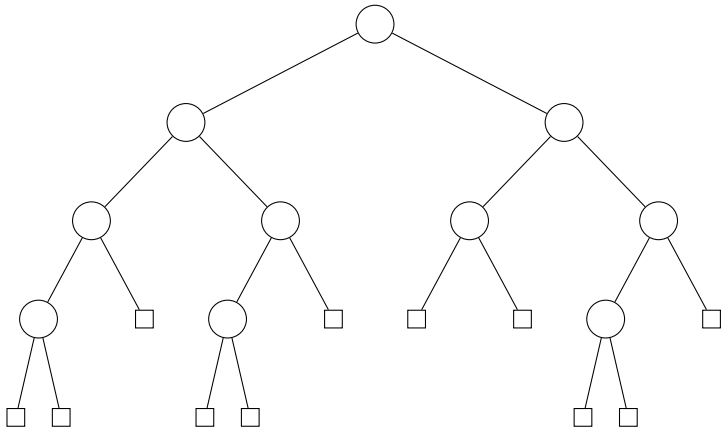
Ex. 3 Caractérisations des arbres complets

1. Prouver qu'un arbre de hauteur h est complet si et seulement si, pour tout $k \in \{0, 1, \ldots, h\}$, le nombre, N_k, de nœuds de profondeur k est $N_k = 2^k$.
2. En déduire qu'un arbre de hauteur h est complet si et seulement si sa taille n vérifie $n = 2^{h+1} - 1$.
3. Prouver qu'un arbre est complet si et seulement si, pour tout nœud de l'arbre, les deux fils ont la même hauteur.

En particulier cela implique que la taille d'un arbre complet ne peut être que de la forme $2^{h+1} - 1$ si h est la hauteur de l'arbre.
On peu diminuer la contrainte et conserver la majoration de la taille en fonction de la hauteur pour des arbres de taille quelconque.

Arbre quasi-complet

Une arbre de hauteur h est quasi-complet si toutes ses feuilles sont à une profondeur h ou $h + 1$.



Ex. 4 Caractérisation des arbres quasi-complets
Prouver qu'un arbre de hauteur h est quasi-complet si et seulement si, pour tout $k \in \{0, 1, \ldots, h-1\}$, le nombre, N_k, de nœuds de profondeur k est $N_k = 2^k$.

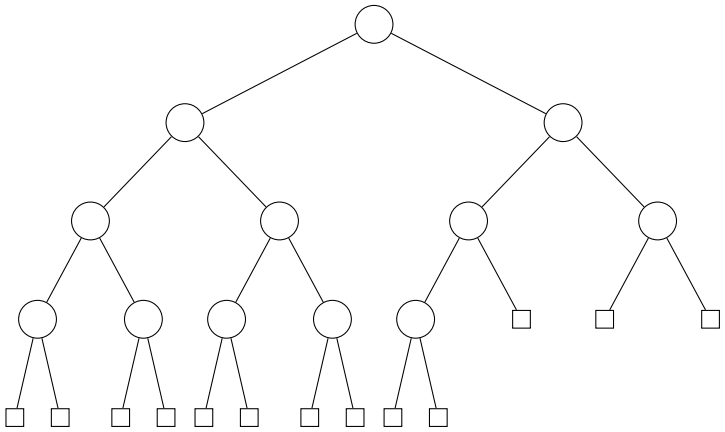
Les arbres quasi-complets ont une hauteur optimale.

Ex. 5 Majoration de la hauteur
Prouver que si h est la hauteur et n la taille d'un arbre quasi-complet alors on a $h = \lfloor \log_2(n) \rfloor$.

Les nœuds à la profondeur h n'ont pas un ensemble fixé d'emplacements. On peut rigidifier un peu les positions.

Arbre quasi-complet à gauche

Une arbre de hauteur h est quasi-complet à gauche s'il est quasi-complet et si toutes ses feuilles de profondeur h sont à la droite de tous les nœuds de profondeur h.
Les arbres quasi-complets à gauche sont ceux dont les feuilles sont parcourues en dernier lors d'un parcours en largeur qui visite les nœuds et les feuilles.

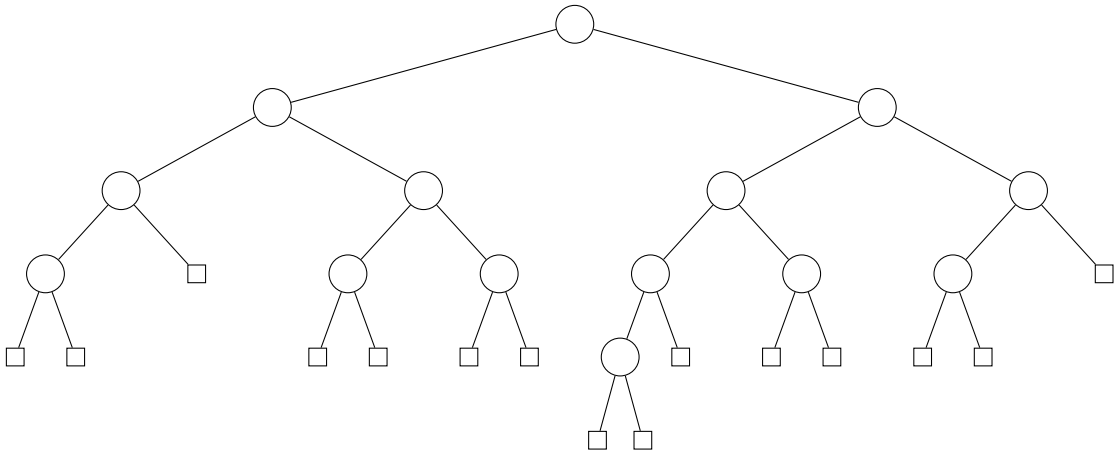


3-2 Équilibre

Arbre équilibré

Une arbre est équilibré si tous ses nœuds ont des fils g et d dont les hauteurs, h_g et h_d, vérifient $|h_d - h_g| \leqslant 1$.

Ex. 6 Majoration du déséquilibre
Prouver qu'un arbre quasi-complet est équilibré.



Hauteur d'un arbre équilibré

Il existe une constante A telle que, p our un arbre équilibré de hauteur h et de taille n, $h \leqslant A\log_2(n)$

La suite de Fibonacci, (F_n), est définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$.

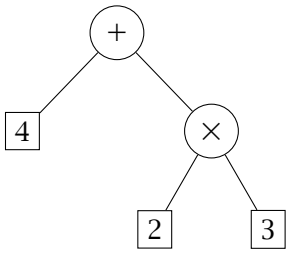
Ex. 7 Démonstration
Montrer que si a est un arbre équilibré de hauteur h alors il contient au moins $F_{h+3} - 1$ nœuds. En déduire le théorème ci-dessus.

4 Généralisations

4-1 Arbres hétérogènes

Il nous semble aller de soi que l'écriture $4+2*3$ représente un calcul qui effectue d'abord $2*3$ pour calculer ensuite $4+6$ et donner 10. Cette lecture est le résultat d'un long entraînement qui nous a appris les règles de priorité. Nous savons aussi que si on veut faire calculer l'addition avant la multiplication on doit parenthéser cette addition et écrire $(4+2)*3$ pour obtenir 18. Lorsque l'on demande des calculs à un ordinateur, celui-ci doit appliquer ces règles de priorité ; dans la pratique il transforme notre expression en une structure non ambiguë qui permettra ensuite d'effectuer les calculs. Cette structure est celle d'un arbre qui utilise deux types d'objets :

- ↪ des objets initiaux ou atomiques, ici des entiers : dans l'arbre ils seront représentés aux extrémités, ce sont les **feuilles**
- ↪ des assemblages, ici des opérations, sur des objets déjà construits, ce sont les **nœuds** de l'arbre.



L'arbre défini à partir de $4+2*3$ est
Le type sera

```
type ('a, 'b) arbre = Feuille of 'f
                    |NoeudH of (('f,'n) arbre * 'n * ('f,'n) arbre);;
```

Les arbres homogènes sont des cas particuliers d'arbres hétérogènes ; les feuilles sont de type `unit`, elles représentent les feuilles vides.

4-2 Plus que deux

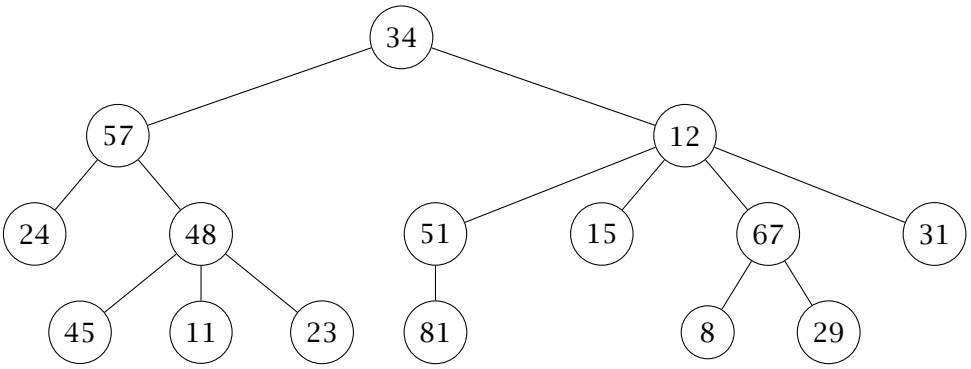
On peut aussi généraliser en n'imposant plus le nombre d'éléments associés à chaque nœud, on revient au cas d'arbres homogènes..

Arbres généraux

Une arbre de type α est défini récursivement comme la donnée d'un élément de type α et d'une liste (éventuellement vide) d'arbres de type α.

On notera qu'on ne parle plus d'arbre vide : tout arbre contient au moins un nœud, ni de feuilles : il n'est pas nécessaire de marquer l'absence d'un fils puisqu'il n'y a pas de nombre prédéterminé de fils.
La traduction en Caml est immédiate :

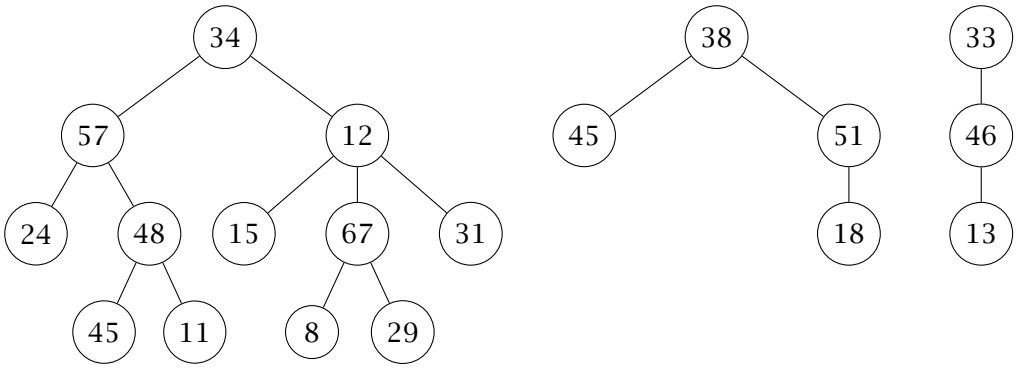
```
type 'a treeG = N of 'a * 'a tree list;;
```



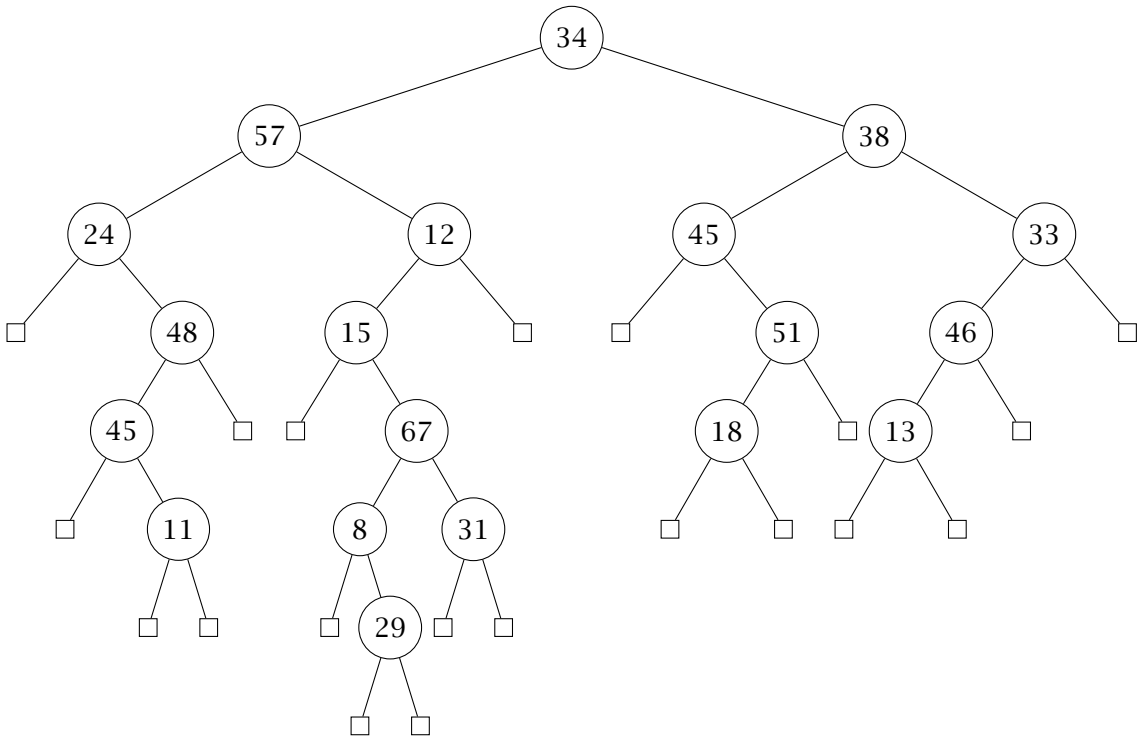
On peut réintroduire le vide en considérant des listes d'arbres ou **forêts**.
On définit alors les deux types et un arbre vide est alors une forêt sans arbre.

```
type 'a foret = 'a arbre list
 and 'a arbre = N of 'a * 'a foret;;
```

On notera qu'on a défini deux types tels que chacun utilise l'autre dans sa définition : on fait intervenir une récursivité **croisée**. On doit alors utiliser le mot-clé **and** pour associer les deux définitions.



On peut transformer une forêt en arbre binaire en associant à une liste d'arbre le nœud dont la racine est la valeur du premier arbre dans la liste, le fils droit est associé à queue de la liste et le fils gauche est associé à la forêt du premier arbre de la liste.
L'exemple ci-dessus devient



5 Exercices

5-1 Dénombrements

Ex. 8 Égalité de Kraft

On note N_k le nombre de nœuds à la profondeur k et F_k le nombre de feuilles à la profondeur k d'un arbre.

Montrer que $N_{k+1} + F_{k+1} = 2N_k$ et $N_0 + F_0 = 1$.

Si la hauteur de l'arbre est h, prouver l'égalité de Kraft $\sum_{k=0}^{h+1} \frac{F_k}{2^k} = 1$.

Ex. 9 Longueur de cheminement

La longueur de cheminement d'un arbre est la somme des profondeurs de tous ses nœuds.

1. Déterminer une fonction qui calcule la longueur de cheminement d'un arbre.
2. Si LC est la longueur de cheminement d'un arbre de taille n, prouver

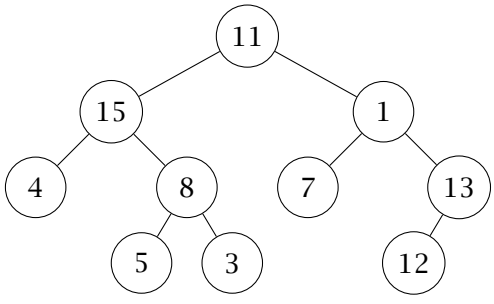
$$\sum_{k=1}^{n} \lfloor \log_2(k) \rfloor \leqslant LC \leqslant \frac{n(n-1)}{2}$$

5-2 Parcours

Ex. 10 Conversion vers liste

Écrire des fonctions qui renvoient les éléments d'un arbre binaire sous la forme d'une liste en suivant les différents parcours (infixe, préfixe et postfixe).

Par exemple l'arbre



donnera respectivement `[4; 15; 5; 8; 3; 11; 7; 1; 12; 13]`,
`[11; 15; 4; 8; 5; 3; 1; 7; 13; 12]` et `[4; 5; 3; 8; 15; 7; 12; 13; 1; 11]`.

5-3 Arbres complets et quasi-complets

Ex. 11 Longueur de cheminement

On note LC la longueur de cheminement (voir exercice 9) d'un arbre.
Prouver qu'un arbre de taille n est quasi-complet si et seulement si sa longueur de cheminement est minimale parmi les arbre de taille n et que cette longueur de cheminement vaut $LC = (n+1)h - 2^{h+1} + 2$ avec $h = \lfloor \log_2(n) \rfloor$.

Ex. 12 Numérotation des nœuds

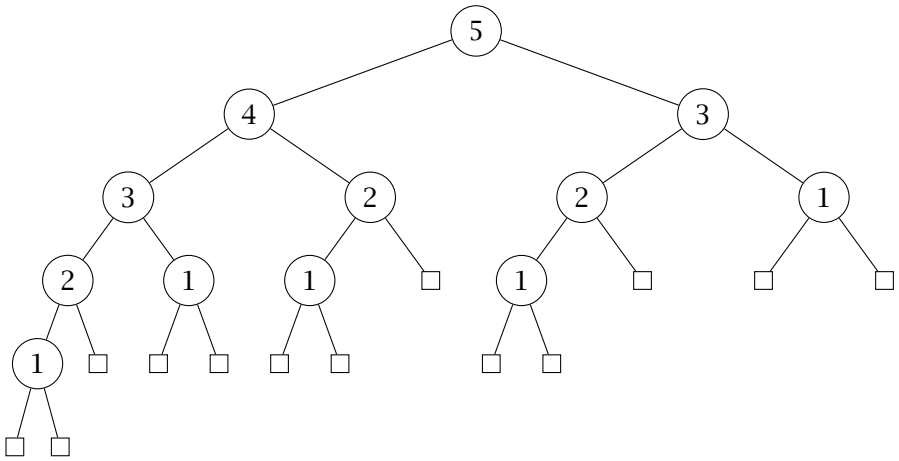
On numérote les nœuds d'un arbre quasi-complet à gauche en suivant un parcours en largeur entre 1 et n où n est la taille de l'arbre.
On note $N(i)$ le nœud de numéro i.
Lorsque les fils de $N(i)$ sont des nœuds quels sont leurs numéros ?
Quel est le père de $N(i)$ pour $i \geqslant 2$?

5-4 Arbres équilibrés

Ex. 13 Test d'équilibrage

Écrire une fonction`test_equilibre` qui renvoie `true` ou `false` selon que l'arbre est équilibré ou non.

On définit les arbres de Fibonacci par :
1. fibo(0) est l'arbre vide,
2. fibo(1) est le nœud de racine 1 et de feuilles vides
3. fibo(n+2) est le nœud de racine $n+2$ et de feuilles fibo(n+1) à gauche et fibo(n) à droite.



Ex. 14 Construction des arbres de Fibonacci
Écrire une fonction `fibo` telle que `fibo n` renvoie fibo(n).

Ex. 15 Hauteur des arbres de Fibonacci Quelle sont la hauteur et la taille de fibo(n) ?
En déduire qu'un arbre de Fibonacci est équilibré.
Prouver que fibo(n) admet des feuilles à la profondeur n et à la profondeur $\lceil \frac{n}{2} \rceil$ pour $n \geqslant 1$.

5-5 Arbres généraux

Ex. 16 Calculs
En généralisant raisonnablement les définitions, écrire les fonctions `taille_gen` et `hauteur_gen` qui calculent la taille et la hauteur d'un arbre général.

Ex. 17 Conversions
Écrire les fonctions `foret2bin` et `bin2foret` qui convertissent une forêt en un arbre binaire homogène (`tree`) et réciproquement.

Chapitre VIII

Programmation dynamique

1 Première approche, quelques suites

1-1 Suite de Fibonacci, rappels

On a déjà étudié la programmation de la suite de Fibonacci :

$$F_n = \begin{cases} 0 & \text{si n=0} \\ 1 & \text{si n=0} \\ F_{n-1} + F_{n-2} & \text{si n} > 1 \end{cases}$$

Le programme récursif que l'on écrit naturellement n'est pas efficace.

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

Par contre on peut garder en mémoire les deux dernières valeurs calculées
↪ soit dans deux variables référencées

```
let fibo n =
  let f = ref 0 in
  let f_suiv = ref 1 in
  for i = 1 to n do
    let  f_old = !f in
    f := !f_suiv;
    f_suiv = !f + !f_old done;
!f;;
```

↪ soit dans les variables d'une fonction auxiliaire

```
let fibo n =
  let rec aux_fibo k f1 f2 =
    if k = 0
    then f1
    else aux_fibo (k - 1) f2 (f1 + f2) in
  aux_fibo n 0 1;;
```

Dans ce cas l'algorithme est de complexité linéaire.

1-2 Nombres de Catalan

Les nombres de Catalans sont définis par

$$C_n = \begin{cases} 1 & \text{si n=0} \\ \sum_{k=0}^{n-1} C_k C_{n-1-k} & \text{si n>0} \end{cases}$$

On peut écrire l'algorithme récursif qui applique la définition

```
let rec ctl n =
  match n with
  |0 -> 1
  |n -> let c = ref 0 in
           for k = 0 to (n-1) do
             c := !c + (ctl k)*(ctl (n-1-k)) done;
          !c;;
```

Le temps de calcul est très long : près de 4 minutes pour calculer C_{20}.

Ex. 1 Complexité
On note $\gamma(n)$ le nombre d'opérations, additions et multiplications effectuées pour calculer C_n. Montrer que $\gamma(0) = 0$ et $\gamma(n) = 2\sum_{k=0}^{n-1} \gamma(k) + 2n$.
On pose $u_n = \sum_{k=0}^{n} \gamma(k)$, prouver que $u_n = 3u_{n-1} + 2n$ puis $u_n = \frac{1}{2}3^{n+1} - n - \frac{3}{2}$.
En déduire $\gamma(n)$.

On peut améliorer un peu les choses en remarquant que chaque C_k est utilisé deux fois.

```
let rec ctl n =
  match n with
  |0 -> 1
  |n -> let c = ref 0 in
           let p = n/2 in
           for k = 0 to (p-1) do
             c := !c +2* (ctl k)*(ctl (n-1-k)) done;
          if n mod 2 = 1
         then (let cc = ctl p in c := !c + cc**cc);
         !c;;
```

La complexité est améliorée : 78 secondes pour calculer C_{30}.

Ex. 2 **Complexité**

Montrer qu'on a maintenant $\gamma(n) = \sum_{k=0}^{n-1} \gamma(k) + \frac{3n}{2}$ pour n pair et $\gamma(n) = \sum_{k=0}^{n-1} \gamma(k) + \frac{3n+1}{2}$ pour n impair ; on admet qu'alors $\gamma(n) = \mathcal{O}(2^n)$.

Pour pouvoir effectuer moins de calculs, on va éviter de répéter les calculs des C_k. Pour cela on les **mémorise** ; ici le plus simple est de calculer le tableau des valeurs, pas-à-pas.

```
let catalan n =
  let c = Array.make (n+1) 0 in
  c.(0) <- 1;
  for p = 1 to n do
    for  k = 0 to (p-1) do
        c.(p) <- c.(p) + c.(k)*c.(p - 1 - k) done done;
   c.(n);;
```

Ex. 3 **Complexité**

Calculer la complexité de `catalan`.

Ex. 4 **Emploi de listes**

Proposer un calcul des nombres de Catalan de même complexité asymptotique, qui utilise des listes. On pourra, si besoin, utiliser `List.rev` pour retourner une liste.

2 Un exemple moins élémentaire

2-1 Présentation du problème

Le lycée vient d'installer une nouvelle salle de T.P. d'informatique et, bien entendu, tout le monde voudrait pouvoir l'utiliser

Vous êtes chargé d'affecter la salle en essayant d'en optimiser l'attribution. La condition incontournable est qu'il n'y ait pas deux classes qui occupent la salle en même temps.

Chaque personne donne donc son horaire et espère pouvoir bénéficier de la salle.
Ces demandes sont définies sous la forme de triplets
$\{(d_1, f_1, v_1), (d_2, f_2, v_2), \ldots, (d_n, f_n, v_n)\}$.

↪ d_i est la date du début de l'occupation souhaitée, sous forme d'un entier ; on peut, par exemple, énumérer depuis la première heure de cours le lundi de la rentrée,
↪ f_i est la date de la fin de l'occupation souhaitée,
↪ v_i est la valeur pour l'occupation, ce peut être la durée, le nombre d'élèves concernés, leur importance (on pourrait privilégier les optionnaires d'informatique), …

On choisit un implémentation des demandes sous la forme

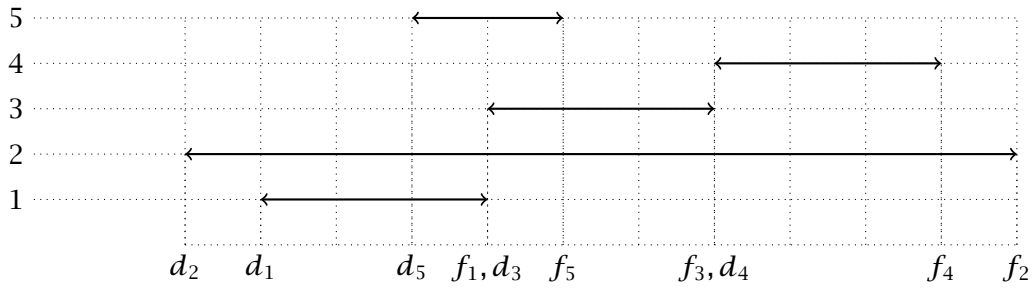
```
type demande = {num : int; debut : int; fin : int; valeur : int};;
```

`num` permet de conserver une référence au numéro de la demande.
Les demandes seront placées dans une liste.
À partir de ces données on cherche une solution.

↪ Un **planning d'occupation** est une suite d'entiers distincts de $\{1, 2, \ldots, n\} : P = (i_1, i_2, \ldots, i_p)$ qui vérifie $f_{i_k} \leqslant d_{i_{k+1}}$ pour tout $k < p$.
↪ La valeur du planning est $V(P) = \sum_{k=1}^{p} v_k$.
↪ Une solution est un planning d'occupation de valeur maximale.

2-2 Cas simple : algorithme glouton

Nous allons commencer par un cas particulier pour lequel $v_i = 1$ pour tout i.
Optimiser le planning consiste alors à contenter le maximum de demandes.
Exemple
On considère les demandes $\{(1,1,4,1),(2,0,11,1),(3,4,7,1),(4,7,10,1),(5,3,5,1)\}$.
On peut les représenter graphiquement.

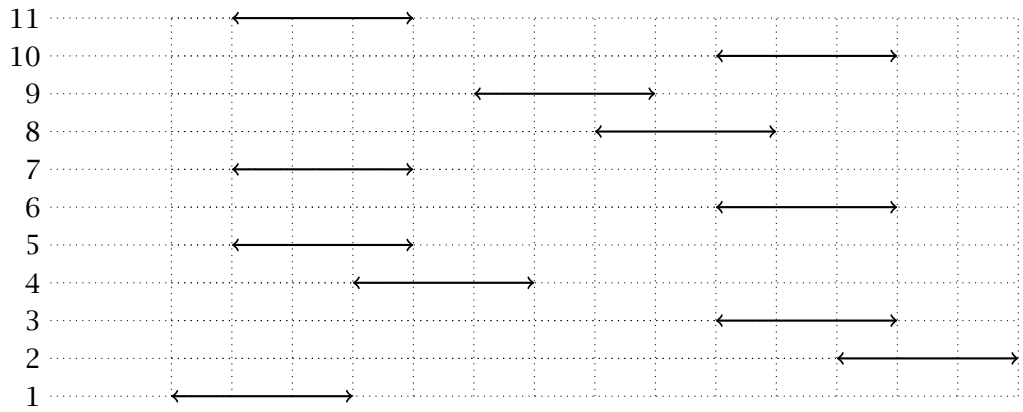


On voit facilement que le meilleur planning est (1,3,4) : sa valeur est 3.
On va chercher s'il existait un moyen "simple" de trouver ce planning. Pour cela on cherche un algorithme **glouton** c'est-à-dire qui trouve une solution en regardant la problème sans recul, juste en optimisant l'instant.
On recherche le profit instantané sans penser au futur (glouton se dit **greedy** en anglais).

↪ On pourrait choisir comme premier élément celui qui commence le plus vite :
on voit sur l'exemple que cela conduit au planning non optimal (2).

↪ On pourrait choisir de prendre l'occupation la plus courte et de construire une solution à partir de cela. Ici on choisirait 5 et on ne pourrait obtenir qu'un planning de valeur 2 : (5,4)

↪ On pourrait choisir de prendre l'occupation qui interfère le moins possible avec d'autres puis de compléter avec ce principe. L'exemple suivant montre que cela peut conduire à un planning (1,9,2) non optimal car il existe le planning (1,4,8,2).



↪ On peut enfin choisir de libérer au plus vite la place : on choisit l'intervalle qui finit le premier. Cela semble fonctionner

```
E est l'ensemble des indices
A est vide
Tant que E est non vide
    on choisit i dans E tel que f(i) est minimal
    on enlève de E les indices j tels que d(j) < f(i)
    on ajoute i à A (à droite)
On renvoie A
```

Ex. 5 Algorithme glouton

1. Écrire en Caml l'algorithme ci-dessus.
2. Prouver qu'il détermine bien un planning optimal.
3. Quelle est la complexité ?

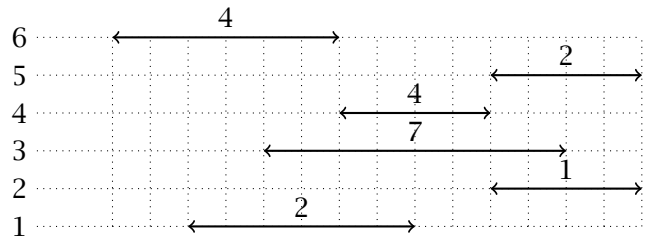
La complexité en $\mathcal{O}(n^2)$ peut être améliorée en triant les demandes au préalable par un tri en $O(n\log_2(n))$: il suffit alors de lire la liste une seule fois pour ajouter ou non chaque demande selon que son début est compatible ou pas avec la fin déjà atteinte.

Ex. 6 Algorithme glouton amélioré

Écrire en OCaml cet algorithme ; on supposera déjà écrite une fonction de tri.
Quelle est la complexité ?

2-3 Cas général

Dans le cas général la valeur d'un planning est calculée en pondérant les différentes demandes. On ne connaît pas, dans ce cas, d'algorithme glouton c'est-à-dire qui construise une solution en choisissant une demande à la fois et en construisant ainsi une solution globalement optimale par des décisions locales. En fait ce qui est étonnant est que cela ait pu fonctionner.



Le planning optimal est (6, 4, 5) de valeur 10.

Il existe un algorithme très simple à énoncer :

```
maximum = 0
reponse = vide
Pour toute partie I de {1,2,...,n}
  si les intervalles ne se chevauchent pas
    on calcule la somme des valeurs, V
    si V > maximum
      maximum = V
      reponse = I
Renvoyer (maximum,reponse)
```

On doit alors définir toutes les parties de I, il y en a 2^n pour n demandes puis calculer la somme des valeurs. La complexité est donc en $\mathcal{O}(n2^n)$ ce qui n'est pas raisonnable.
On va améliorer cet algorithme pour aboutir à une solution utilisable.

Forme récursive

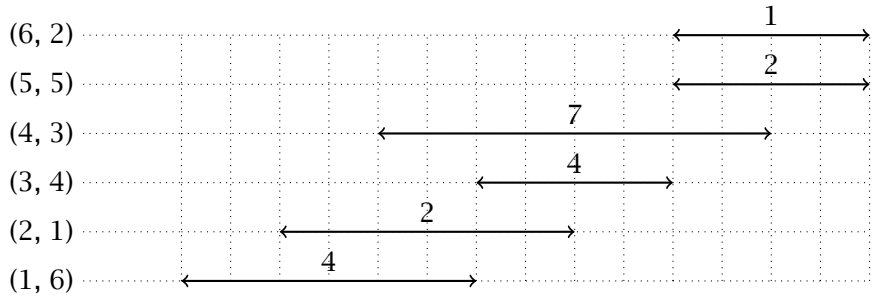
Une manière récursive de penser l'algorithme est de séparer les parties selon qu'elles contiennent ou non la dernière demande (ou la première).

1. On calcule le maximum des valeurs pour les $n-1$ premières demandes
2. On calcule la somme de v_n et du maximum des valeurs pour les demandes compatibles avec la dernière demande.
3. On renvoie le maximum de ces deux valeurs.

Cette écriture peut donner des calculs efficaces dans certains cas mais reste en $\mathcal{O}(2^n)$ dans le pire des cas.

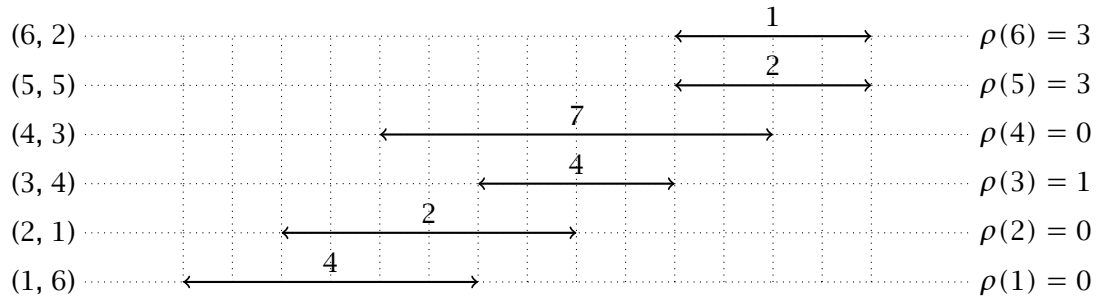
Pré-traitement

Nous allons faciliter le calcul des demandes compatibles avec une demande donnée.
On commence par trier la liste des demandes selon la date de fin.
Les demandes sont alors $(d'_1, f'_1, v'_1), (d'_2, f'_2, v'_2), \ldots, (d'_n, f'_n, v'_n)$ avec $f'_1 \leqslant f'_2 \leqslant \cdots \leqslant f'_n$.



Le numéro entre parenthèses est celui de la demande originale.

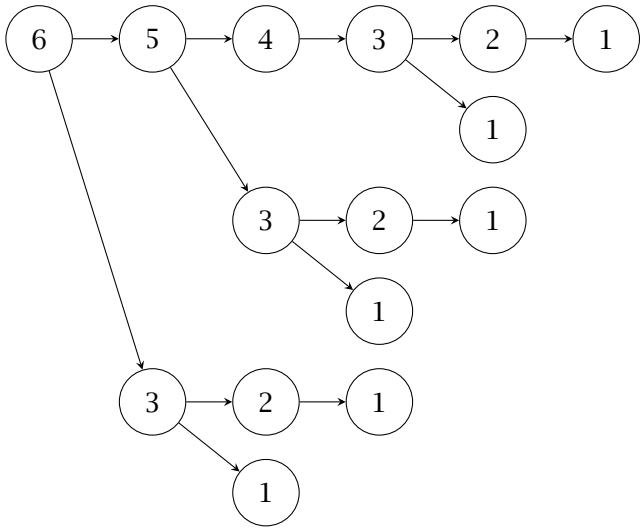
On aura besoin de déterminer les demandes compatibles avec la dernière. Comme on le fera de manière récursive on cherche les demandes finissant avant la k-ième et compatibles avec celle-ci. Pour chaque k on définit $\rho(k)$ comme le plus grand entier j tel que $f'_j \leqslant d'_k$.



En séparant les cas selon que la dernière demande (n) appartient ou non à une distribution, la somme optimale vaut v'_n plus la somme optimale pour les $\rho(n)$ premiers termes ou la somme optimale pour les $n-1$ premiers termes.

$$opt(k) = \max\{opt(k-1), v_k + opt(\rho(k))\}$$

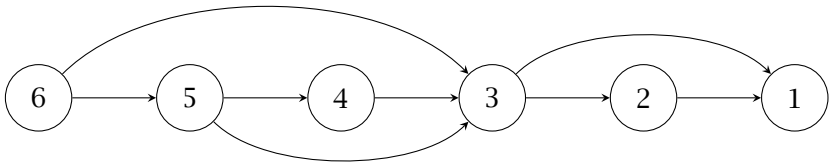
La complexité reste en $\mathcal{O}(2^n)$ mais on voit que cela provient du fait qu'on calcule plusieurs fois la même chose. Les différents calculs sont symbolisés ci-dessous.



Pour chaque valeur la flèche horizontale indique le calcul de $opt(k-1)$ et la flèche vers le bas indique le calcul de $opt(rho(k))$. On n'a pas indiqué le calcul de $opt(0)$.

Mémoïsation

Un moyen simple de diminuer la complexité est alors de retenir (on parle de **mémoïsation**) les différentes valeurs calculées.
Les calculs calculs sont symbolisés par le graphe suivant.



Pour conserver les valeurs déjà calculées, comme ces valeurs dépendent d'un paramètre entier et borné, on peut utiliser un tableau.
On peut utiliser un type optionnel pour les valeurs du tableau afin de déterminer facilement si une valeur a déjà été calculée. On conservera alors une structure récursive.
Il peut être plus lisible de renoncer à la récursivité et de construire le tableau des valeurs, `opt`, pas-à-pas. On n'a alors plus besoin du type optionnel.

Implémentation

On définit les demandes sous la forme d'un enregistrement. Chaque demande va être caractérisée par un numéro et contient ses dates (sous forme d'entiers, par exemple) de début et de fin ainsi que son poids.

```
type demande = {num : int;
                debut : int;
                fin : int;
                valeur : int};;
```

L'ensemble des demandes est sous la forme d'un tableau (`demande vect`), dénommé `tab`.
Deux demandes sont comparées par la date de fin :

```
let inferieur d1 d2 = d1.fin < d2.fin;;
inferieur : demande -> demande -> bool = <fun>
```

Ex. 7 **Tri de tableau**
Écrire une fonction de tri qui reçoit un tableau et une fonction de comparaison et qui renvoie un tableau trié sans modifier le tableau initial (le tri n'est pas en place). On pourra utiliser le tri-fusion.

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

Ex. 8 **Fonction rho**
Écrire une fonction qui reçoit un tableau trié de demandes et qui renvoie le tableau des valeurs de ρ : `rho.(k)` devra être la valeur de ρ pour la demande placée à la k-ième position.
Les valeurs de ρ seront représentée par un type optionnel pour gérer la non-existence d'une demande précédant une demande donnée.
Quelle est la complexité ?

Ex. 9 **Calcul de l'optimum**
Écrire une fonction qui reçoit un tableau de demandes et qui renvoie la valeur optimale d'attribution.
Quelle est la complexité ?

Ex. 10 **Calcul de la distribution optimale**
Écrire une fonction qui renvoie la valeur optimale d'attribution et la liste des numéros d'une demande qui réalise ce maximum.

3 Principe

Nous avons, dans la partie précédente, réussi à transformer un algorithme de complexité exponentielle en algorithme de complexité polynomiale. Nous allons essayer de mettre en évidence les caractéristiques du problème qui ont permis cette amélioration.
Ces principes sont à comprendre comme des indication de la méthode de programmation dynamique ; il semble impossible d'en donner une définition stricte.

Qu'avons nous fait ?

À partir d'un algorithme naïf itératif nous avons donné une interprétation récursive. Celle-ci a alors été améliorée en remarquant que l'on calculait toujours les mêmes valeurs et qu'il pouvait être judicieux de garder ces valeurs en mémoire.
Ce qui a permis cela est que le calcul pour un ensemble de n demandes pouvait se calculer à partir du même calcul pour des ensembles plus petits.
On notera qu'à ce moment il est souvent plus simple d'écrire un algorithme itératif pour faire les calculs de tous les ensembles.
On applique ici un principe fondamental de la programmation.
Éviter de calculer plusieurs fois le même résultat, surtout dans un algorithme récursif.

Conditions de la programmation dynamique

- ↪ On doit pouvoir calculer une solution (le plus souvent récursivement) pour une entrée à partir de solutions pour des entrées plus petites.
- ↪ Le nombre de sous-cas doit être polynomial.
- ↪ La programmation dynamique est souvent employée pour résoudre des problèmes d'optimisation satisfaisant le principe de Bellman "*Dans une séquence optimale (de décisions ou de choix), chaque sous-séquence doit aussi être optimale*".

On peut considérer que calculer les termes de la suite de Fibonacci en sauvegardant les valeurs successives dans un tableau plutôt que d'appliquer la définition récursive revient à employer les idées de programmation dynamique.

4 Exercices

4-1 PLSSC

En biologie, on a souvent besoin de comparer l'ADN de deux (ou plusieurs) organismes. Un échantillon d'ADN est une suite de molécules appelées bases, les bases possibles étant l'adénine, la guanine, la cytosine et la thymine. Si l'on représente chacune de ses bases par son initiale, un échantillon d'ADN s'exprime sous la forme d'une chaîne de caractères de type

$$s = ACCGGTCGAGTGCCAGTTGACG$$

Comparer deux échantillons d'ADN revient à déterminer leur degré de similitude qui mesure la façon dont deux organismes sont apparentés. Il existe plusieurs définitions de la similitude, nous allons choisir celle de la plus longue sous-séquence commune.

Étant donné une séquence (suite finie) $X = (x_0, x_1, \ldots, x_m)$ une séquence $Z = (z_0, z_1, \ldots, z_p)$ est une **sous-séquence** de X si elle est une suite extraite de X c'est-à-dire s'il existe une séquence $(i_0, i_1, \ldots, i_p)$ strictement croissante d'indices telle que $Z_k = X_{i_k}$ pour tout $k \in \{0, 1, \ldots, p\}$.
Par exemple $Z = ATCA$ est une sous-séquence de $X = GATACGA$ correspondant aux indices $1, 2, 4, 6$.
Le degré de similitude entre deux séquences d'ADN peut être mesuré par la longueur de la plus longue séquence commune (noté PLSSC dans la suite) aux deux échantillons.

Comme d'habitude la méthode brutale (comparer toutes les sous-séquences) est inexploitable.
Soient $X = (x_0, x_1, \ldots, x_n)$ et $Y = (y_0, y_1, \ldots, y_m)$ deux séquences et $Z = (z_0, z_1, \ldots, z_p)$ une plus longue sous-séquence commune. La récurrence est basée sur la remarque :
↪ si $x_n = y_m$ alors on peut choisir $z_p = x_n$ et $Z' = (z_0, z_1, \ldots, z_{p-1})$ est une PLSSC de $X' = (x_0, x_1, \ldots, x_{n-1})$ et $Y' = (y_0, y_1, \ldots, y_{m-1})$
↪ si $x_n \neq y_m$ alors Z est une PLSSC de X et Y' ou de X' et Y.

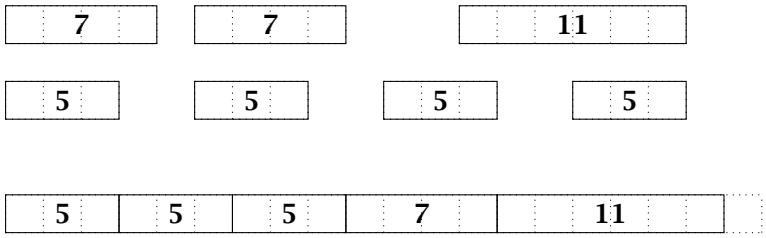
Ex. 11 **Preuve** Prouver l'assertion ci-dessus.

Ex. 12 **Algorithme** Écrire une fonction qui calcule une PLSSC.

4-2 Sac à dos

De manière rapide : comment emmener une valeur maximale dans un volume donné ?
Plus formellement, on considère
↪ n objets ayant chacun un encombrement (poids ou volume) w_i et une valeur v_i (cette valeur peut être w_i)
↪ un contenant (le sac-à-dos) de capacité totale W

L'objectif est de trouver une partie $S \subset \{1, 2, \ldots, n\}$ telle que $\sum_{i \in S} w_i \leqslant W$ et qui maximise la valeur totale $\sum_{i \in S} v_i$.
Par exemple si on a 4 objets de taille 3 et de valeur 5, 2 objets de taille 4 et de valeur 7 et un objet de taille 6 et de valeur 11 alors le meilleur remplissage d'un sac de taille 20 se fait avec 3 objets de taille 3,1 objet de taille 4 et l'objet de taille 6 pour une valeur totale de 33.



1. On peut imaginer remplir
   ↪ les plus valorisés d'abord
   ↪ les moins (ou les plus) encombrants d'abord
   ↪ les meilleurs qualité-prix d'abord
   Aucun ne fonctionne dans tous les cas.

2. L'algorithme naïf consiste à calculer les poids de tous les sous-ensembles possibles puis de garder le plus grand qui reste inférieur à W. On aboutit à une complexité impraticable en $O(n2^n)$.

3. Cet algorithme devient récursif si on remarque que la meilleure solution est
   ↪ soit la meilleure solution pour les $n-1$ premiers objets
   ↪ soit la solution avec le dernier objet et la meilleure solution avec $n-1$ objet et un encombrement qui ne dépasse pas $W - w_n$.

4. On a donc la condition qui permet d'appliquer la programmation dynamique :
   les sous-problèmes sont indexés par
   ↪ k, le nombre d'objets que l'on peut inclure
   ↪ w, le poids qu'il ne faut pas dépasser
   Le nombre de sous cas est $(n+1)(W+1)$, c'est un $O(nW)$.
   La relation de récurrence est
   $$\texttt{opt}(k+1, w) = \texttt{max}(\texttt{opt}(k, w), \texttt{opt}(k, w - w_{k+1}) + v_{k+1})$$

On suppose que les données sont sous la forme d'un tableau `objets` de couples `(poids, valeur)`, chaque couple représentant un objet par son poids et sa valeur.

Ex. 13 **Poids optimal** Écrire une fonction `sacAdos objets wMax` qui calcule le poids optimal.

Ex. 14 **Remplissage optimal** Modifier la fonction pour qu'elle renvoie, de plus, la liste des objets qui forment une réponse optimale.

4-3 Produit de matrices

On considère une suite de n matrices d'entiers (pour simplifier) à multiplier : on veut calculer $M_1.M_2.\cdots.M_n$.
Il est possible d'évaluer ce produit en utilisant l'algorithme classique permettant de multiplier deux matrices, après avoir placé des parenthèses pour préciser l'ordre dans lequel les produits binaires doivent être effectués.

Ex. 15 **Produit de matrices** Écrire une fonction qui calcule le produit de deux matrices. Précisez le coût, en nombre de multiplication d'entiers en fonction des tailles des matrices.

La multiplication des matrices étant associative, tous les parenthésages aboutissent à une même valeur du produit. Cependant, la manière dont une suite de matrices est parenthésée peut avoir un impact crucial sur le coût d'évaluation du produit.

Ex. 16 **Associativité** Donner un exemple de dimensions pour trois matrices M_1, M_2 et M_3 tel que le coût du calcul de $(M_1.M_2).M_3$ soit différent de celui de $M_1.(M_2.M_3)$. Montrez que le rapport entre ces deux coûts peut être arbitrairement grand.

On peut représenter les dimensions des matrices $(M_1, \cdots, M_n)$ par un tableau de $n+1$ entiers `d` tels que, pour tout i, la matrice M_i a pour dimensions `(d.(i-1), d.(i))`.
On note, pour $i \leq j$, $p_{i,j}$ le nombre minimum de multiplications d'entiers nécessaires pour le calcul de $M_i.M_{i+1}.\cdots.M_{j-1}.M_j$.

Ex. 17 **Récurrence**
Montrer que si $i < j$ alors $p_{i,j} = \min\{p_{i,k} + p_{k+1,j} + d_{i-1}d_k d_j \ ; \ i \leqslant k < j\}$.

Ex. 18 **Optimum** En déduire une fonction qui calcule le nombre minimal de multiplications entières à effectuer pour calculer le produit d'une suite de matrices avec, pour argument, le tableau des dimensions d.
Déterminer la complexité.

Par exemple pour $M_1 \in \mathcal{M}_{2,4}$, $M_2 \in \mathcal{M}_{4,3}$, $M_3 \in \mathcal{M}_{3,1}$ et $M_4 \in \mathcal{M}_{1,5}$,
c'est-à-dire `d = [|2; 4; 3; 1; 5|]`, la réponse doit être 30.

Ex. 19 **Parenthésage** Écrire une fonction qui détermine l'ordre de calcul optimal du produit $M_1.M_2.\cdots.M_n$ sous la forme d'un parenthésage.
On pourra définir un tableau qui retient la coupure k qui fait les produits optimaux.

Dans l'exemple ci-dessus, la réponse peut être `"((M1.(M2.M3)).M4)"`

Chapitre IX

Réponses aux exercices

Dans ce chapitres sont proposées des réponses aux exercices.
Ces solutions n'ont pas la prétention d'être optimales, le but est de montrer que l'on peut arriver à écrire une solution en utilisant les éléments du cours. Parfois plusieurs réponses sont proposées, cela correspond à des méthodes différentes mais parfois à des lectures distinctes de l'énoncé.
Il recommandé de chercher les exercices avant d'en lire la solution.

1 Introduction

Éléments de réponse pour l'exercice 1

```
let calculer_u n u0 f =
  let u = ref u0 in
  for i = 1 to n do u := (f !u) done;
  !u;;
```

Éléments de réponse pour l'exercice 2

```
let longueurCollatz a =
  let u = ref a in
  let n = ref 0 in
  while !u > 1 do
    u:= if !u mod 2 = 0 then !u /2 else 3 * !u +1;
    n := !n + 1 done;
  !n;;

let hauteurCollatz a =let hauteurCollatz a =
  let u = ref a in
  let h = ref a in
  while !u > 1 do
    u:= if !u mod 2 = 0 then !u /2 else 3 * !u +1;
    if !u > !h then h := !u done;
  !h;;
```

Éléments de réponse pour l'exercice 3

```
let somme t =
  let n = Array.length t in
  let s = ref 0 in
  for i = 0 to (n-1) do s := !s + t.(i) done;
  !s;;
```

Éléments de réponse pour l'exercice 4

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

Éléments de réponse pour l'exercice 5

```
let occurences t =
  let n = Array.length t in
  let occ = Array.make 10 0 in
  for i = 0 to (n-1) do
    let k = t.(i) in occ.(k) <- occ.(k) + 1 done;
  occ;;
```

Éléments de réponse pour l'exercice 6

```
somme (somme (somme 2 3) 4) (somme 2 (somme 3 4));;
```

Éléments de réponse pour l'exercice 7

```
let unimodal t =
  let reponse = ref true in
  let monte  = ref true in
  let n = Array.length t in
  for i = 0 to (n-2) do
    if t.(i+1) < t.(i) && !monte
    then monte := false;
    if t.(i+1) > t.(i) && (not !monte)
    then reponse := false done;
  !reponse;;
```

Éléments de réponse pour l'exercice 8

```
let comp f g = let h x = f (g x) in h;;

#comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Éléments de réponse pour l'exercice 9
La fonction calcule l'itérée n-ième de f.

```
#itere : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Éléments de réponse pour l'exercice 10

```
let estH n =
  let nn = ref n in
  while !nn mod 2 = 0 do nn := !nn / 2 done;
  while !nn mod 3 = 0 do nn := !nn / 3 done;
  while !nn mod 5 = 0 do nn := !nn / 5 done;
  !nn = 1;;

let indSuivant p t =
  let i = ref 0 in
  while t.(!i) <= p do i := !i + 1 done;
  !i;;

let min3 a b c = min (min a b) c;;

let tableauH n =
  let t = Array.make n 0 in
  t.(0) <- 1;
  for i = 1 to (n-1) do
    let der = t.(i-1) in
    let a = indSuivant (der/2) t in
    let b = indSuivant (der/3) t in
    let c = indSuivant (der/5) t in
    t.(i) <- min3 (2*t.(a)) (3*t.(b)) (5*t.(c)) done;
  t;;

let hamming n = (tableauH n).(n-1);;

244140625
```

Éléments de réponse pour l'exercice 11

```
let echanger t i j =
    let temp = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- temp;;


let retourner t i j =
    let c = (i+j-1)/2 in
    for k =i to c do echanger t k (i+j-k) done;;

let suivant t =
    let tt = Array.copy t in
    let n = Array.length t in
    let j = ref (n-2) in
    while !j >= 0 && tt.(!j) >= t.(!j+1) do j := !j - 1 done;
    if !j == -1
    then [||]
    else let k = ref (n-1) in
         while tt.(!k) <= tt.(!j) do k := !k - 1 done;
         echanger tt !j !k;
         retourner tt (!j+1) (n-1);
         tt;;
```

2 Récursivité

Éléments de réponse pour l'exercice 1
Une première idée est de calculer u_{n-1} puis d'appliquer la récurrence.

```
let rec suite u0 n =
  if n <= 0
  then u0
  else let u = suite u0 (n-1) in (2.0*.u +. 1.0)/.(3.0 +. u);;
```

On peut remarquer que le n-ième terme de la suite commençant par u_0 est le $n-1$-ième terme de la suite commençant par u_1.

```
let rec suite u0 n =
  if n <= 0
  then u0
  else suite ((2.0*.u0 +. 1.0)/.(3.0 +. u0)) (n-1);;
```

Cet algorithme est récursif terminal.

Éléments de réponse pour l'exercice 2

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

La fonction effectue n multiplications, ce qui parait normal. Nous verrons au chapitre suivant qu'on peut écrire un algorithme qui effectue beaucoup moins de multiplications.

Éléments de réponse pour l'exercice 3

```
let rec ackermann m n =
  if m = 0
  then n+1
  else if n = 0
       then ackermann (m-1) 1
       else ackermann (m-1) (ackermann m (n-1));;
```

Éléments de réponse pour l'exercice 4

```
let rec binomial n p =
  if p = 0
  then 1
  else (n*(binomial (n-1) (p-1)))/p;;
```

Éléments de réponse pour l'exercice 5

1.

```
let pgcd a b =
  if b = 0
  then a
  else pgcd b (a mod b);;
```

2. La propriété est vraie pour $p = 1$ car on a $b > 0$ si on fait un appel donc $b \geqslant 1 = F_1$ et alors $a > b$ donne $a \geqslant 2 = F_2$.
   Si la propriété est vraie pour un entier p, on suppose que `pgcd a b` effectue $p + 1$ appels récursifs. Alors `pgcd b (a mod b)` effectue p appels récursifs donc on a $b \geqslant F_{p+1}$ et $c \geqslant F_p$ où c est le reste de la division de a par b. Comme on a $a > b$ on a $c \leqslant a - b$ donc $a \geqslant c + b \geqslant F_p + F_{p+1} = F_{p+2}$: la propriété est vraie pour $p + 1$.
   La propriété est donc vérifiée pour tout p.
3. On suppose qu'on a $0 \leqslant a \leqslant F_n$ et $0 \leqslant b \leqslant F_n$ et que `pgcd a b` effectue p appels récursifs. Pour $p = 0$ on a bien $0 \leqslant n$. On suppose maintenant $p \geqslant 1$.
   - ↪ Si on a $a > b$ alors on doit avoir $F_{p+1} \leqslant a \leqslant F_n$ donc $p \leqslant n - 1$.
   - ↪ Si on a $a = b$ alors le premier appel sera `pgcd b 0` qui renvoie b directement : on a fait 1 appel et on abien $1 \leqslant n$.
   - ↪ Si on a $a < b$ alors le premier appel sera `pgcd b a` qui sera suivi de $p - 1$ appels. Pour $0 < a < b$ on a $p - 1 \geqslant 1$ et le premier cas ci-dessus donne $p - 1 \leqslant n-$ donc $p \leqslant n$. Si on avait $a = 0$, il n'y avait qu'un appel et alors $1 \leqslant n$.

   Dans tous les cas on a au plus n appels donc n calculs par `mod`.

Éléments de réponse pour l'exercice 6

1.

```
let rec catalan n =
  match n  with
  |0 -> 1
  |n -> let c = ref 0 in
        for k = 0 to (n-1) do
          c := !c + (catalan k)*(catalan (n-1-k)) done;
        !c;;
```

2. La fonction fait de nombreuses fois le même calcul lors des appels ; elle fait aussi deux fois les mêmes calculs à chaque étape.
3.

```
let catalan1 n =
  let cat = Array.make (n+1) 1 in
  for k = 1 to n do
    let c = ref 0 in
    for i = 0 to (k-1) do c := !c + cat.(i)*cat.(k-1-i) done;
    cat.(k) <- !c done;
  cat.(n);;
```

Éléments de réponse pour l'exercice 7

Le vrai cas terminal est celui d'une liste réduite à un élément.

```
let rec maximum liste =
  match liste with
  |[] -> failwith "Liste vide"
  |[a] -> a
  |t::q -> max t (maximum q);;
```

Éléments de réponse pour l'exercice 8

```
let rec carre liste =
  match liste with
  |[] -> []
  |t::q -> (t*t)::(carre q);;
```

```
let rec applique f liste =
  match liste with
  |[] -> []
  |t::q -> (f t)::(applique f q);;
```

Éléments de réponse pour l'exercice 9

```
let rec selectionMax liste =
  match liste with
  |[] -> failwith "liste vide"
  |[x] -> x, []
  |t::q -> let max, reste = selectionMax q in
           if  t < max
           then max, t::reste
           else t, q;;
```

Éléments de réponse pour l'exercice 10

Dans la fonction `retourne` du cours on effectue un assemblage pour chaque élément de la liste : la complexité est donc n, si n est le nombre d'éléments de la liste.
Par contre, dans `renverse`, la concaténation a une complexité égale à la longueur de `renverse q`.
On a ainsi $C(n) = C(n-1) + n - 1$ donc $C(n) = \frac{n}{2}(n-1)$ ce qui est asymptotiquement plus grand que la complexité de `retourne`.

Éléments de réponse pour l'exercice 11

1.

```
let rec positifs liste =
  match liste with
  |[] -> []
  |t::q when t >= 0 -> t :: (positifs q)
  |_::q -> positifs q;;
```

2.

```
let rec filtrer f liste =
  match liste with
  |[] -> []
  |t::q when f t -> t :: (filtrer f q)
  |_::q -> filtrer f  q;;
```

3.

```
('a -> bool) -> 'a list -> 'a list
```

Éléments de réponse pour l'exercice 12

1.

```
let rec verifie f liste =
  match liste with
  |[] -> None
  |t::q when f t -> Some t
  |t::q -> verifie f q;;
```

2.

```
let rec il_existe f liste =
  match liste with
  |[] -> false
  |t::q when f t -> true
  |t::q -> il_existe f q;;
```

3.

```
let rec pour_tous f liste =
  match liste with
  |[] -> true
  |t::q when f t -> verifie f q
  |t::q -> false;;
```

Éléments de réponse pour l'exercice 13

```
let rec shuffle l1 l2 =
  match l1, l2 with
  |[], l2 -> l2
  |l1, [] -> l1
  |t1::q1, t2::q2 -> t1 :: (t2 :: (shuffle q1 q2));;
```

Éléments de réponse pour l'exercice 14

```
let rec shuffle l1 l2 =
  match l1, l2 with
  |[], l2 -> l2
  |l1, [] -> l1
  |t1::q1, t2::q2 -> t1 :: (t2 :: (shuffle q1 q2));;
```

Éléments de réponse pour l'exercice 15

```
let rec parties liste =
  match liste with
  |[] -> [[]]
  |t::q -> let p1 = parties q in
           let p2 = List.map (fun l -> t::l) p1 in
           p1@p2;;
```

Éléments de réponse pour l'exercice 16

```
type flou = Vrai | Faux | Incertain;;
```

```
let non f =
  match t1 with
  |Vrai -> Faux
  |Faux -> Vrai
  |Incertain -> Incertain;;
```

```
let et f1 f2 =
  match t1, t2 with
  |Vrai, Vrai -> Vrai
  |Faux, _ -> Faux
  |_, Faux -> Faux
  |_, _ -> Incertain;;
```

```
let ou f1 f2 =
  match t1, t2 with
  |Faux, Faux -> Faux
  |Vrai, _ -> Vrai
  |_, Vrai -> Vrai
  |_, _ -> Incertain;;
```

Éléments de réponse pour l'exercice 17

```
type nComplet = Infini|Fini of int;;

let addition n m =
  match (n,m) with
  |Infini,_ -> Infini
  |_,Infini -> Infini
  |Fini p, Fini q -> Fini (p+q);;

let produit n m =
  match (n,m) with
  |Infini,Fini 0 -> failwith "Forme indéterminée"
  |Fini 0,Infini -> failwith "Forme indéterminée"
  |Infini,_ -> Infini
  |_,Infini -> Infini
  |Fini p,Fini q -> Fini (p*q);;
```

Éléments de réponse pour l'exercice 18

```
type rationnel = Quot of int*int;;

let rec pgcd a b =
  let c = abs a and d = abs b in
  if c > d
  then pgcd d c
  else if c = 0
       then d
       else pgcd (d mod c) c;;

let simplifie (Quot(a,b)) =
  let d = pgcd a b in
  Quot (a/d,b/d);;

let sommeR (Quot(a,b)) (Quot(c,d)) = simplifie (Quot(a*d+b*c,b*d));;

let produitR (Quot(a,b)) (Quot(c,d)) = simplifie (Quot(a*c,b*d));;
```

Éléments de réponse pour l'exercice 19

```
let rec enum a b =
  if a > b then []
           else a::(enum (a+1) b);;
```

Éléments de réponse pour l'exercice 20

```
let rec oterMult k liste =
  match liste with
  |[] -> []
  |t::q -> if t mod k = 0
           then oterMult k q
           else t::(oterMult k q);;
```

Éléments de réponse pour l'exercice 21

```
let premier n =
  let rec elim liste =
    match liste with
    |[] -> []
    |t::q -> t::(elim (oterMult t q)) in
  elim (enum 2 n);;
```

À chaque nombre premier on parcourt la liste restante. Il y a au plus n nombre premiers et les listes sont de taille n au plus : on effectue au plus n^2 instructions.
N.B. On peut imaginer que cette majoration n'est pas optimale ; en fait le théorème de densité des nombres premiers dit que, si $\pi(n)$ est le nombre de premiers inférieurs à n alors $\pi(n) \sim \ln(n)$. On obtient ainsi une majoration en $n\ln(n)$.

Éléments de réponse pour l'exercice 22

```
let premier n =
  let rec elim liste =
    match liste with
    |[] -> []
    |t::q -> if t*t <= n
               then t::(elim (oterMult t q))
               else liste in
  elim (enum 2 n);;
```

On effectue maintenant au plus $n^{3/2}$ instructions.

Éléments de réponse pour l'exercice 23
Les entiers de 0 à 7 ont pour développements binaires 000, 001, 010, 011, 100, 101, 110, 111.
Un code de gray peut être 000, 010, 110, 100, 101, 111, 011, 001 (soit 0, 2, 6, 4, 5, 7, 3, 1).

Éléments de réponse pour l'exercice 24
On traduit l'énoncé : $a_0, a_1, \ldots, a_p$ est un code de Gray ($p = 2^n - 1$).
Alors $0a_0, 0a_1, \ldots, 0a_{p-1}, 0a_p, 1a_p, 1a_{p-1}, ldots, 1a_1, 1a_0$ est un code de Gray. En effet
↪ une seule lettre change entre $0a_i$ et $0a_{i+1}$ car une seule est changée entre a_i et a_{i+1},
↪ de même une seule lettre change entre $1a_{i+1}$ et $1a_i$,
↪ il reste $0a_p$ et $1a_p$ entre lesquels seule la première lettre change.

Éléments de réponse pour l'exercice 25
On décompose en fonctions simples les étapes.

```
let retourner liste =
  let rec aux l1 l2 =
    match l1 with
    |[] -> l2
    |t::q -> aux q (t::l2) in
  aux liste [];;

let ajouter char liste =
  map (fun x -> char^x) liste;;

let rec gray n =
  if n = 0
  then [""]
  else let  l = gray (n-1) in
       (ajouter "0" l)@(ajouter "1" (retourner l));;
```

Éléments de réponse pour l'exercice 26
On a toujours $g_n(0) = 0$.
$g_n(2^n - 1)$ s'obtient en ajoutant un 1 en binaire devant $g_n(0)$ à la place $n - 1$: on obtient 2^{n-1}.
$g_n(2^{n-1} - 1) = g_{n-1}(2^{n-1} - 1) = 2^{n-2}$
$g_n(2^{n-1})$ s'obtient en ajoutant un 1 en binaire devant $g_n(2^{n-1} - 1)$ à la place $n - 1$: on obtient $2^{n-1} + 2^{n-2} = 3.2^{n-2}$.
En raison de la construction par symétrie on a $g_n(k) < 2^{n-1}$ pour $k < 2^{n-1}$ et $g_n(k) \geqslant 2^{n-1}$ pour $k \geqslant 2^{n-1}$.

Éléments de réponse pour l'exercice 27
On utilise la question précédente.
Le successeur de 2^{n-2} est 3.2^{n-2} et celui de 2^{n-1} et 0.

Pour $k < 2^{n-1}$ (et $k \neq 2^{n-2}$ le successeur de k dans Γ_n est son successeur dans Γ_{n-1}.
En raison de la symétrie, pour $k > 2^{n-1}$ le successeur de k est $2^{n-1} + k'$ où k' est le prédécesseur de k_2^{n-1} dans Γ_{n-1}.
On va donc écrire deux fonctions récursives simultanées.

```
let rec grayPlus k n =
  if n = 1
  then 1-k
  else (let p = puiss2 (n-2) in
        if k = 2*p
        then 0
        else if k = p
             then 3*p
             else if k < 2*p
                  then grayPlus k (n-1)
                  else 2*p + grayMoins (k-2*p) (n-1))
and grayMoins k n =
  if n = 1
  then 1-k
  else (let q = puiss2 (n-2) in
        if k = 0
        then 2*q
        else if k = 3*q
             then q
             else if k < 2*q
                  then grayMoins k (n-1)
                  else 2*q + grayPlus (k-2*q) (n-1));;
```

Éléments de réponse pour l'exercice 28
Si la liste décroissante de ces termes est $[2^k b; 2^{k-1} b; \ldots; 2b; b]$ alors la liste des termes pour $a/2$ est $[2^{k-1} b; \ldots; 2b; b]$. On a un algorithme récursif.

```
let rec puiss2b a b =
  if a < b
  then []
  else match (puiss2b (a/2) b) with
       |[] -> [b]
       |t::q -> (2*t)::(t::q);;
```

Éléments de réponse pour l'exercice 29

```
let reste a b =
 let rec aux l petit_a =
    match l with
    |[] -> petit_a
    |t::q -> if t <= petit_a
             then aux q (petit_a - t)
             else aux q petit_a in
  aux (puiss2b a b) a;;
```

Éléments de réponse pour l'exercice 30
Le nombre d'opération est un $\mathcal{O}(\lambda)$ où λ est la longueur de la liste `puiss2b a b` que ce soit pour le calcul de cette liste ou pour son exploitation dans la fonction `reste`.
Or le nombre de termes de la forme $b2^k$ majorés par a est $p+1$ si on a $2^p < \frac{a}{b} \leqslant 2^{p+1}$.
La complexité de `reste` est donc un $\mathcal{O}(\ln(b/a)) = \mathcal{O}(\ln(b))$.

Éléments de réponse pour l'exercice 31

```
let rec est_sous_liste m1 m2 =
  match m1, m2 with
  |[], _ -> true
  |_, [] -> false
  |t1 :: q1, t2 :: q2 -> if t1 = t2
                         then est_sous_liste q1 q2
                         else est_sous_liste m1 q2;;
```

Éléments de réponse pour l'exercice 32
On sépare les sous-listes selon qu'elles contiennent t ou pas.

Éléments de réponse pour l'exercice 33

```
let rec sous_listes = function
  |[] -> [[]]
  | t :: q -> let e = sous_listes q in
              e @ (map (fun m -> t :: m) e);;
```

Éléments de réponse pour l'exercice 34
L'inégalité $F_{i_r} \leqslant n$ est évidente.
On montre ensuite par récurrence sur r que si on a $i_r \gg i_{r-1} \gg \cdots \gg i_1 \gg i_0 \geqslant 1$ alors $\sum_{k=0}^{r} F_{i_k} < F_{i_r+1}$.

C'est vrai pour $r = 0$ car $i \geqslant 1$ implique $F_i < F_{i+1}$.
Si la propriété est vraie pour r alors $i_{r+1} \gg i_r \gg i_{r-1} \gg \cdots \gg i_1 \gg i_0 \geqslant 1$ implique $i_{r+1} \geqslant i_r + 2$ et $\sum_{k=0}^{r} F_{i_k} < F_{i_r+1} \leqslant F_{i_{r+1}-1}$ puis $\sum_{k=0}^{r+1} F_{i_k} < F_{i_{r+1}-1} + F_{i_{r+1}} = F_{i_{r+1}+1}$.

Éléments de réponse pour l'exercice 35
On procède par récurrence.
Pour $n = 1$, la seule décomposition possible est (F_1).
On suppose que tout entier k avec $1 \leqslant k < n$ avec $n \geqslant 2$ admet une décomposition unique.
D'après la question précédente, si $n \geqslant 2$ admet une décomposition le premier terme doit être p tel que $F_p \leqslant n < F_{p+1}$.
On a $2 = F_2 \leqslant n < F_{p+1}$ donc $p + 1 > 2$ puis $p \geqslant 2$.
On a alors $n - F_p < F_{p+1} - F_p = F_{p-1}$.
Si $n = F_p$ alors (F_p) est une décomposition et c'est la seule car elle doit contenir F_p.
Si $n > F_p$ alors $n - F_p$ admet une décomposition unique $(F_{i_r}, F_{i_{r-1}}, \ldots, F_{i_0})$. D'après la question précédente on a $F_{i_r} \leqslant n - F_p < F_{i_r+1}$ et on a vu qu'on a $n - F_p < F_{p-1}$. On a donc $i_r < p - 1$ donc $p \gg i_r$.
Ainsi $(F_p, F_{i_r}, F_{i_{r-1}}, \ldots, F_{i_0})$ est une décomposition de $F_p + n - F - p = n$ et c'est la seule en raison de l'unicité du premier terme et de l'unicité de la décomposition de $n - F_p$.

Éléments de réponse pour l'exercice 36
On a $F_9 = 55 \leqslant 67 < 89 = F_{10}$, $F_5 = 8 \leqslant 67 - 55 = 12 < 13 = F_6$, $F_3 = 3 \leqslant 12 - 8 = 4 < 5 = F_4$ et $4 - 3 = 1 - F_1$ donc la décomposition de 67 est (55,8,3,1).

Éléments de réponse pour l'exercice 37

```
let suivant liste =
  match liste with
  |t::s::q -> t+s
  |_ -> failwith "erreur dans suivant" ;;

let listeFibo n =
  let rec construction liste =
    let suiv = suivant liste in
    if suiv <= n
    then construction (suiv::liste)
    else liste in
  construction [1; 1];;
```

Éléments de réponse pour l'exercice 38
On applique l'algorithme :

```
let rec decompose n =
  if n = 0
  then []
  else let k = List.hd (listeFibo n) in
       k::(decompose (n - k));;
```

La fonction ci-dessus calcule la suite de Fibonacci pour chaque nouvelle valeur.
On peut améliorer avec

```
let decompose n =
  let rec aux k liste =
    match liste with
    |[] -> []
    |t::q -> if t > k then aux k q else t::(aux (k-t)) q in
    aux n (listeFibo n);;
```

Éléments de réponse pour l'exercice 39
On ajoute récursivement à une représentation en différenciant les cas en fonction du premier terme de la représentation.
Comme on va utiliser $p-2$ il faut gérer le cas $p \leqslant 1$.
Pour prouver que le résultat est une représentation n utilise la propriété, que l'on montre par récurrence, que si le premier terme de la représentation est majoré par p alors le premier terme du résultat est majoré par $p+1$.

```
let rec somme1 p listeF =
  match listeF with
  |[] -> [p]
  |q::reste when q < p-1 -> p::listeF
  |q::reste when q = p-1 -> (p+1)::reste
  |q::reste when q = p && p= 1 -> [1]
  |q::reste when q = p && p= 2 -> [3; 1]
  |q::reste when q = p -> (p+1)::(somme1 (p-2) reste)
  |q::reste -> somme1 q (somme1 p reste);;
```

Éléments de réponse pour l'exercice 40
On ajoute récursivement chaque terme de la liste.

```
let rec somme liste1 liste2 =
  match liste1 with
  |[] -> liste2
  |p::reste -> somme1 p (somme reste liste2)::
```

3 Analyse d'algorithmes, tris simples

Éléments de réponse pour l'exercice 1

```
let recherche x tableau =
  let n = Array.length tableau in
  let reponse = ref false in
  for i = 0 to (n-1) do
    if tableau.(i) = x then reponse := true done;
  !reponse;;
```

1. L'itération est effectuée par une boucle **for** donc l'algorithme termine.

2. On note $P(i)$ la propriété : `reponse` vaut `true` si et seulement si il existe k compris entre 0 et $i-1$ tel que `tableau.(k)` vaut `x`.
   $P(0)$ est vraie car il n'y a rien entre 0 et -1 et `reponse` vaut initialement `false`.
   On suppose que $P(i)$ est vérifiée.
   Si `tableau.(i)` vaut `x` alors `reponse` vaut `true` et il y a bien un entier entre 0 et i en lequel x est atteint.
   Si `tableau.(i)` ne vaut pas `x` alors la valeur de `reponse` est inchangée et elle a la vérité de l'appartenance de x dans le tableau entre 0 et $i-1$ d'après l'hypothèse de récurrence donc d son existence entre 0 et i.
   Dans les deux cas $P(i+1)$ est vérifiée.
   Enfin $P(n)$ signifie que l'algorithme fait bien ce qui est attendu.

3. On va compter le nombre de comparaisons avec un terme du tableau : il y en a n.

Éléments de réponse pour l'exercice 2

```
let rec somme liste =
  match liste with
  |[] -> 0
  |t::q -> t + somme q;;
```

1. La taille de la liste diminue à chaque appel : c'est un variant.

2. On montre par récurrence sur la taille que la fonction renvoie bien la somme des termes.

3. À chaque appel on fait une addition de plus sauf pour la liste vide, le nombre d'additions est la taille de la liste.

Éléments de réponse pour l'exercice 3

a. On suppose qu'on a $0 \leqslant p \leqslant n$.

```
let rec binomSomme n p =
  if p = 0 || p = n
  then 1
  else (binomSomme (n-1) p) + (binomSomme (n-1) (p-1));;
```

1. n décroît strictement lors de chaque appel donc l'algorithme termine.

2. La correction de l'algorithme est la conséquence de la formule de récurrence avec les conditions initiales.

3. On note $a(n,p)$ le nombre d'additions qu'exécute `binomSomme n p`. On a $a(0,n) = a(n,n) = 0$ et $a(n,p) = a(n-1,p) + a(n-1,p-1) + 1$.
   Si on pose $b(n,p) = a(n,p) + 1$ on obtient
   $b(0,n) = b(n,n) = 1$ et $b(n,p) = b(n-1,p) + b(n-1,p-1)$.
   On retrouve la définition de $\binom{n}{p}$ donc la complexité est $\binom{n}{p} - 1$.

b. Avec les mêmes contraintes ($0 \leqslant p \leqslant n$) on peut écrire

```
let rec binomProd n p =
  if p = 0
  then 1
  else (binomProd (n-1) (p-1))*n/p;;
```

Comme les coefficients binomiaux la division entière est exacte.

1. n décroît strictement lors de chaque appel donc l'algorithme termine.

2. La correction de l'algorithme est la conséquence de la formule de récurrence avec les conditions initiales.

3. On note $a(n,p)$ le nombre de multiplications et divisions.
   On a $a(n,0) = 0$ et $a(n,p) = 2 + a(n-1,p-1)$ d'où $a(n,p) = 2p$.

c. Le second algorithme est meilleur dans le cas général.

Éléments de réponse pour l'exercice 4
On introduit une fonction auxiliaire qui reçoit en paramètres : le reste de la liste, la somme des termes vus et le maximum des tranches vues.

```
let maxTete liste =
  let rec aux_maxT reste debut t_max =
    match reste with
    |[] -> t_max
    |t::q -> let deb = debut + t in
             aux_maxT q deb (max t_max deb)
  in aux_maxT liste 0 0 ;;
```

- La taille de la liste appelée décroît strictement à chaque appel : l'algorithme termine.
- On note S_k la somme des k premiers termes de la liste avec $S_0 = 0$.
  On considère l'invariant : si le reste contient $n - k$ termes alors `debut` vaut S_k et `max` vaut $\max\{S_i\ ; 0 \leqslant i \leqslant k\}$.
  On remarque qu'alors la fonction est écrite pour conserver cet invariant.
- On effectue une somme et une comparaison pour chaque terme de la liste : la complexité est $2n$ où n est la taille de la liste.

Éléments de réponse pour l'exercice 5

```
let rec trancheMax liste =
  match liste with
  |[] -> 0
  |t::q -> let m1 = maxTete liste in
           let m2 = trancheMax q in
           max m1 m2;;
```

- `maxTete` termine et la taille de la liste appelée décroît strictement à chaque appel récursif de `trancheMax` : l'algorithme termine.
- La preuve de l'algorithme découle de l'indication de l'énoncé : la tranche maximale est soit une tranche de `q`, soit une tranche commençant par `t`.
- Si $C(n)$ est le nombre d'opérations effectuées lors du calcul d'une tranche maximale d'une liste de taille n on a $C(0) = 0$ et $C(n+1) = 2(n+1) + C(n)$ donc $C(n) = n.(n+1)$: la complexité est quadratique.

Éléments de réponse pour l'exercice 6

- Ici encore la taille de la liste appelée décroît strictement à chaque appel récursif de `aux` : l'algorithme termine.

- On commence par prouver que la valeur de `fin` est la valeur maximale des tranches de la partie vue de la liste qui sont vides ou qui contiennent le dernier élément vu.
  En effet cette valeur maximale est 0 ou contient le dernier élément et, dans ce cas, on ajoute à ce dernier élément une tranche finale maximale des derniers éléments.
  Il suffit alors de voir que la tranche maximale parmi les k premiers éléments est, soit une tranche maximale parmi les $k-1$ premiers éléments, soit une tranche terminale.

- La complexité est linéaire : 3 opérations pour chaque terme.
  On a bien un algorithme plus efficace.

Éléments de réponse pour l'exercice 7
On doit changer la séparation pour trouver un minimum.

```
let rec selectionMin liste =
  match liste with
  |[] -> failwith "selectionMin d'une liste vide"
  |[x] -> x, []
  |t::q -> let min, reste = selectionMin q in
           if  pluspetit t min then t, q
                               else min, t::q;;
```

```
let triSelection liste =
  match liste with
  |[] -> []
  |_ -> let min, reste = selectionMin liste in
        min::(triSelection reste);;
```

Éléments de réponse pour l'exercice 8
On remarque que la complexité vérifie $C(t::q) = C(q) + C'(t,q')$ où q' est le résultat du tri de q et $C'(a,\ell)$ est le coût de l'insertion de a dans la liste ℓ.

1. Pour l_σ on a $t = \sigma(1)$, $q = l'_\sigma = [\sigma(2);\ldots;\sigma(n)]$.
   On a alors $q' = L_{\sigma(1)}) = [1;2;\ldots;\sigma(1)-1;\sigma(1)+1;\ldots;n]$.
   Si on note $C'_k = C'(k, L_k)$ on a donc $C(l_\sigma) = C(l'_\sigma) + C'_{\sigma(1)}$ d'où
   $\overline{C}(n) = \dfrac{1}{n!}\displaystyle\sum_{i=1}^{n}\left(\sum_{\sigma(1)=i} C(l'_\sigma) + C'_i\right) = \dfrac{1}{n!}\sum_{i=1}^{n}\sum_{\sigma(1)=i} C(l'_\sigma) + \sum_{i=1}^{n}\left(\dfrac{1}{n!}\sum_{\sigma(1)=i} C'_i\right).$

2. Pour chaque i, il y a $(n-1)!$ permutations telles que $\sigma(1)=i$ donc $\frac{1}{n!}\sum_{\sigma(1)=i} C'_i = \frac{1}{n!}.(n-1)!.C'_i = \frac{1}{n}C'_i$.
   De plus l'insertion de a dans une liste de taille p effectue de 1 à p comparaisons.
   Il y a 1 comparaison si a est placé en tête,
   2 comparaisons si a est placé en deuxième position,
   p comparaisons si a est placé en p-ème position
   et p aussi si a est placé en dernière position.
   Ainsi, quand on insère i dans une liste triée contenant tous les autres éléments de 1 à n, on fait i comparaisons, sauf pour $i=n$ pour lequel on fait $n-1$ comparaisons d'où $\sum_{i=1}^{n} C'_i = \sum_{i=1}^{n-1} i + n - 1 = \frac{n(n-1)}{2} + n - 1 = \frac{(n+2)(n-1)}{2}$.
   On en déduit $\sum_{i=1}^{n}\left(\frac{1}{n!}\sum_{\sigma(1)=i} C'_i\right)\frac{(n+2)(n-1)}{2n}$.

3. Pour chaque i, il y a $(n-1)!$ permutations telles que $\sigma(1)=i$
   et les $(n-1)!$ listes l'_σ correspondent à toutes les permutations de $(1,2,\ldots,i-1,i+1,\ldots,n)$
   donc $\sum_{\sigma(1)=i} C(l'_\sigma) = (n-1)!\overline{C}(n-1)$ puis $\frac{1}{n!}\sum_{i=1}^{n}\sum_{\sigma(1)=i} C(l'_\sigma) = \overline{C}(n-1)$.
   On en déduit que $\overline{C}(n) = \overline{C}(n-1) + \frac{(n+2)(n-1)}{2n} = \overline{C}(n-1) + \frac{n}{2} + 1 - \frac{1}{n}$.

4. On a $\overline{C}(0)=0$ donc $\overline{C}(n) = \sum_{k=1}^{n}\left(\frac{k}{2}+1-\frac{1}{k}\right) = \frac{n(n+1)}{4} + n - \sum_{k-1}^{n}\frac{1}{k}$.
   On en conclut $\overline{C}(n) \sim \frac{n^2}{4}$.

Éléments de réponse pour l'exercice 9

Une stratégie possible est de créer une liste aléatoire de taille $n-1$, de choisir un entier k et d'augmenter de les éléments de la listes supérieurs ou égaux à k. On peut alors placer k en tête.

```
let rec augmentation k liste =
  match liste with
  |[] -> []
  |t::q when t >= k -> (t+1)::(augmentation k q)
  |t::q -> t::(augmentation k q);;
```

```
let rec listeAlea n =
  match n with
  |0 -> []
  |n -> let k = Random.int n in
        k::(augmentation k (listeAlea (n-1)));;
```

4 Diviser pour régner

Éléments de réponse pour l'exercice 1
On note n le degré maximal des deux polynômes `p1` et `p2`.

```
let rec somme p1 p2 =
  match (p1, p2) with
  |[], _ -> p2
  |_, [] -> p1
  |t1::q1, t2::q2 -> (t1 +. t2) :: (somme q1 q2);;
```

La complexité est majorée par n.

```
let rec difference p1 p2 =
  match (p1, p2) with
  |_, [] -> p1
  |[],t2::q2 -> (-.t2)::(difference [] q2)
  |t1::q1, t2::q2 -> (t1 -. t2) :: (difference q1 q2);;
```

La complexité est majorée par n.
On aura besoin de multiplier un polynôme par une constante.

```
let rec fois c p =
  match p with
  |[] -> []
  |t::q -> (c *. t) :: (fois c q);;
```

La complexité est majorée par n.
Cela permet de ré-écrire la différence :

```
let difference p1 p2 = somme p1 (fois (-.1.0) p2;;
```

La complexité est alors majorée par $2n$.

```
let rec produit p1 p2 =
  match p1 with
  |[] -> []
  |t::q -> add (fois t p2) (0.0::(mult q p2));;
```

On effectue tous les produits possibles entre un coefficient de P_1 et un coefficient de p_2 : il y en a au plus $(n+1)^2$. Chacun de ces produits est ajouté à un coefficient : il y a donc au plus $(n+1)^2$ additions.

Éléments de réponse pour l'exercice 2

```
let decoupe m =
  let n = (Array.length m)/2 in
  let m1 = Array.make_matrix n n 0.0 in
  let m2 = Array.make_matrix n n 0.0 in
  let m3 = Array.make_matrix n n 0.0 in
  let m4 = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      m1.(i).(j) <- m.(i).(j);
      m2.(i).(j) <- m.(i).(j+n);
      m3.(i).(j) <- m.(i+n).(j);
      m4.(i).(j) <- m.(i+n).(j+n) done done;
  m1, m2, m3, m4;;
```

```
let assemblage m1 m2 m3 m4 =
  let n = Array.length m1 in
  let m = Array.make_matrix (2*n) (2*n) 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      m.(i).(j) <- m1.(i).(j);
      m.(i).(j+n) <- m2.(i).(j);
      m.(i+n).(j) <- m3.(i).(j);
      m.(i+n).(j+n) <- m4.(i).(j) done done;
  m;;
```

```
let sum a b =
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      c.(i).(j) <- a.(i).(j) +. b.(i).(j) done done;
  c;;
```

```
let sub a b =
  let n = Array.length a in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      c.(i).(j) <- a.(i).(j) -. b.(i).(j) done done;
  c;;
```

Éléments de réponse pour l'exercice 3

```
let majorant2 n =
  let m = ref 1 in
  while !m < n do m := 2 * !m done;
  !m;;
```

```
let borde mat n =
  let p = Array.length mat in
  let q = Array.length mat.(0) in
  let c = Array.make_matrix n n 0.0 in
  for i = 0 to (p-1) do
    for j = 0 to (q-1) do
        c.(i).(j) <- mat.(i).(j) done done;
  c;;
```

```
let extraction mat p q =
  let c = Array.make_matrix p q 0.0 in
  for i = 0 to (p-1) do
    for j = 0 to (q-1) do
        c.(i).(j) <- mat.(i).(j) done done;
  c;;
```

Éléments de réponse pour l'exercice 4

```
let separation tab =
  let n = (Array.length tab)/2 in
  let t1 = Array.make n Complex.zero in
  let t2 = Array.make n Complex.zero in
  for i = 0 to (n-1) do
    t1.(i) <- tab.(2*i);
    t2.(i) <- tab.(2*i+1) done;
  t1, t2;;
```

```
let rec fft_inv coef =
  let n = Array.length coef in
  if n = 1
  then [|coef.(0)|]
  else let coef1, coef2 = separation coef in
       let p1 = fft_inv coef1 in
       let p2 = fft_inv coef2 in
       let p = Array.make n Complex.zero in
       let m = n/2 in
       let pi = 4.0*.(atan 1.0) in
       let demi = {Complex.re = 0.5; Complex.im = 0.0} in
       for k = 0 to (m-1) do
         let e = Complex.polar  1.0 (-.pi*.(float_of_int k)
                                        /.(float_of_int m)) in
         p.(k) <- Complex.mul  demi
                               (Complex.add p1.(k)
                                            (Complex.mul e p2.(k)));
         p.(k+m) <- Complex.mul demi
                                (Complex.sub (Complex.mul e p2.(k))
                                             p1.(k)) done;
       p;;
```

Éléments de réponse pour l'exercice 5

```
let augmentation tableau n =
  let tab_c = Array.make n Complex.zero in
  let m = Array.length tableau in
  for i = 0 to (m-1) do
    tab_c.(i) <- {Complex.re = tableau.(i); Complex.im = 0.0} done;
  tab_c;;
```

```
let produit_fft p1 p2 =
  let m = (Array.length p1) + (Array.length p2) - 1 in
  let n = majorant2 m in
  let q1 = augmentation p1 n in
  let q2 = augmentation p2 n in
  let c1 = fft q1 in
  let c2 = fft q2 in
  let c = Array.make n Complex.zero in
  for i = 0 to (n-1) do
    c.(i) <- Complex.mul c1.(i) c2.(i) done;
  let q = fft_inv c in
  let p = Array.make m 0.0 in
    for i = 0 to (m-1) do
    p.(i) <- q.(i).re done;
  p;;
```

Éléments de réponse pour l'exercice 6

On commence par calculer le point le plus proche d'un point dans une liste.

```
let rec point_proche p liste =
   match liste with
   |[] -> failwith "point-proche : la liste est vide"
   |[q] -> q, (distance p q)
   |q::reste -> let m, d = point_proche p reste in
                let d1 = distance p q in
                if d1 < d
                then q, d1 else m, d;;
```

On calcule une distance par point de la liste.

```
let rec points_proches liste =
  match liste with
  |[] -> failwith "points_proches : la liste est vide"
  |[p] -> failwith "points_proches : il n'y a qu'un seul point"
  |[p; q] -> p, q, (distance p q)
  |p::reste -> let q, r, d = points_proches reste in
               let p1, d1 = point_proche p reste in
               if d1 < d
               then p, p1, d1 else q, r, d ;;
```

On note $C(n)$ le nombre de calculs de distance lors de l'appel de `points_proches` avec une liste de taille n. On a $C(2) = 1$

Pour une liste de taille n on invoque la fonction `point_proche` avec le reste de la liste : on effectue alors $n-1$ calculs de distance.
On a donc $C(n) = C(n-1) + (n-1)$ d'où $C(n) = \frac{n(n-1)}{2}$.

Éléments de réponse pour l'exercice 7

```
let rec decoupage k liste =
   if k = 0
   then [], liste
   else match liste with
        |[] -> failwith "La liste est trop courte"
        |t::q -> let l1, l2 = decoupage (k-1) q in
                 (t::l1), l2;;

let moities liste =
   let n = List.length liste in
   decoupage (n/2) liste;;
```

Éléments de réponse pour l'exercice 8

```
let rec bande liste p0 delta =
   let x0, y0 = p0 in
   match liste with
   |[] -> []
   |(x, y):: q when x < x0 -. delta || x > x0 +. delta -> bande q p0
delta
   |(x, y):: q -> (x, y) :: (bande q p0 delta);;
```

Éléments de réponse pour l'exercice 9
On commence par chercher le point proche au-dessus.

```
let rec point p liste delta =
   let (a, b) = p in
   match liste with
   |[] -> None
   |(x, y)::reste when x > a +. delta -> None
   |q::reste when distance p q > delta -> point p reste delta
   |q::reste -> begin
                  let d = distance (a, b) q in
                  match point p reste delta with
                   |Some (q1,d1) when d1 < d -> Some (q1, d1)
                  |_ -> Some (q, d)
                end;;
```

On va comparer plusieurs fois des résultats sous forme de triplets optionnels.

```
let meilleur triplet1 triplet2 =
   match triplet1, triplet2 with
   |None, _ -> triplet2
   |_, None -> triplet1
   |Some (p1, q1, d1), Some (p2, q2, d2) -> if d1 < d2 then triplet1
else triplet2;;
```

```
let rec points_bande liste delta =
   match liste with
   |[] -> None
   |p::reste -> begin
                  let triplet = points_bande reste delta in
                  match point p reste delta with
                  |None ->  triplet
                  |Some (q, d) -> meilleur (Some (p, q, d)) triplet
                end;;
```

Éléments de réponse pour l'exercice 10

```
let delta triplet =
   match triplet with
   |None -> max_float
   |Some(p, q, d) -> d;;
```

```
let points_proches points  =
   let liste_x = List.sort compare_x points in
   let n0 = List.length points in
   let rec aux liste n =
      match liste with
      |[] -> None
      |[p] -> None
      |[p; q] -> Some (p, q, distance p q)
      |p::reste -> let gauche, droite = moities liste n in
                   let p0 = List.hd droite in
                   let triplet_g = aux gauche (n/2) in
                   let triplet_d = aux droite (n - n/2)in
                   let triplet_e =  meilleur triplet_g triplet_d in
                   let d = delta triplet_e in
                   let b1 = bande liste p0 d in
                   let b = List.sort compare_y b1 in
                   let triplet_c = points_bande b d in
                   meilleur triplet_e triplet_c
  in aux liste_x n0;;
```

Éléments de réponse pour l'exercice 11

Il suffit de renvoyer la liste triée selon les ordonnées à chaque appel de la fonction auxiliaire et de remplacer le tri par la fusion des deux listes triées. La complexité ajoutée reste linéaire et on aboutit au résultat souhaité.

```
let rec fusion l1 l2 =
   match l1, l2 with
   |[], _ -> l2
   |_, [] -> l1
   |t1::q1, t2::q2 when compare_y t1 t2 = 1
                   -> t2 :: (fusion l1 q2)
   |t1::q1, _ -> t1 :: (fusion q1 l2);;
```

```
let points_proches2 points  =
   let liste_x = List.sort compare_x points in
   let n0 = List.length points in
   let rec aux liste n =
      match liste with
      |[] -> None, []
      |[p] -> None, [p]
      |[p; q] when compare_y p q = 1 -> Some (p, q, distance p q), [q;
p]
      |[p; q]-> Some (p, q, distance p q), [p; q]
      |p::reste -> let gauche, droite = moities liste n in
                   let p0 = List.hd droite in
                   let triplet_g, tri_g = aux gauche (n/2) in
                   let triplet_d, tri_d = aux droite (n - n/2)in
                   let triplet_e =  meilleur triplet_g triplet_d in
                   let d = delta triplet_e in
                   let liste_t = fusion tri_g tri_d in
                   let b = bande liste_t p0 d in
                   let triplet_c = points_bande b d in
                   meilleur triplet_e triplet_c, liste_t
  in fst (aux liste_x n0);;
```

5 Tris plus rapides

Éléments de réponse pour l'exercice 1

a. Si n est pair `separer` donne deux parties de même taille égale à $\frac{n}{2} = \left\lfloor \frac{n}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil$.
Si n est impair `separer` donne deux parties de taille $\frac{n+1}{2} = \left\lceil \frac{n}{2} \right\rceil$ et $\frac{n-1}{2} = \left\lfloor \frac{n}{2} \right\rfloor$.
La fusion demande au plus $n-1$ comparaisons car on peut parvenir à deux liste avec un élément avec $n-2$ comparaisons et il reste une comparaison pour ces deux termes.
On aboutit bien à $C_n = C_{\lfloor n/2 \rfloor} + C_{\lceil n/2 \rceil} + n - 1$. De plus on a $C_1 = 0$.

b. $D_n = C_{n+1} - C_n = C_{\lfloor (n+1)/2 \rfloor} + C_{\lceil (n+1)/2 \rceil} + n + 1 - 1 - (C_{\lfloor n/2 \rfloor} + C_{\lceil n/2 \rceil} + n - 1)$.
On a vu que $\lceil (n+1)/2 \rceil = \lfloor n/2 \rfloor + 1$ et $\lfloor (n+1)/2 \rfloor = \lceil n/2 \rceil$ donc
$D_n = C_{\lfloor n/2 \rfloor + 1} - C_{\lfloor n/2 \rfloor} + 1 = D_{\lfloor n/2 \rfloor} + 1$.
$D_1 = C_2 - C_1 = C_1 + C_1 + 2 - 1C_1 = C_1 + 1 = 1$.

c. $D_2 = D_1 + 1 = 2$, $D_3 = D_1 + 1 = 2$. On calcule $D_4 = D_5 = D_6 = D_7 = 3$.
On prouve par récurrence sur p que, pour $2^p \leqslant n < 2^{p+1}$, $D_n = p + 1$.
On vient de voir que cela est vrai pour $p = 0$, $p = 1$ et $p = 2$.
Si la propriété est vraie pour p, on considère n tel que $2^{p+1} \leqslant n < 2^{p+2}$.
On a $D_n = D_{\lfloor n/2 \rfloor} + 1$ et $2^p \leqslant \lfloor n/2 \rfloor < 2^{p+1}$ donc, d'après l'hypothèse de récurrence,
$D_{\lfloor n/2 \rfloor} = p + 1$ puis $D_n = p + 2$.
La propriété est vraie pour tout p.

On a $C_n = C_n - C_1 = \sum_{k=1}^{n-1} C_{k+1} - C_k = \sum_{k=1}^{n-1} D_k$.
On suppose qu'on a $2^q \leqslant n < 2^{q+1} - 1$. On a, en séparant aux 2^p,
$C_n = \sum_{p=0}^{q-1} \sum_{k=2^p}^{2^{p+1}-1} D_k + \sum_{k=2^q}^{n-1} D_k = \sum_{p=0}^{q-1} \sum_{k=2^p}^{2^{p+1}-1} (p+1) + \sum_{k=2^q}^{n-1} (q+1)$.
En comptant le nombre d'éléments égaux on obtient
$C_n = \sum_{p=0}^{q-1} 2^p(p+1) + (n-2^q)(q+1) = \sum_{p=0}^{q-1} 2^p p + \sum_{p=0}^{q-1} 2^p + (n-2^q)(q+1)$
$C_n = \sum_{p=0}^{q-1} 2^p p + 2^q - 1 + (n-2^q)(q+1) = \sum_{p=0}^{q-1} 2^p p + n - 1 + q(n-2^q)$.
On a déjà calculé $\sum_{p=0}^{q-1} 2^p p = (q-2)2^q + 2$ d'où
$C_n = (q-2)2^q + 2 + n - 1 + (n-2^q)q = nq + n - 2^{q+1} + 1$ avec $q = \lfloor \log_2(n) \rfloor$.

Éléments de réponse pour l'exercice 2
Si $|A| = k-1$ alors $\sigma(1) = k$ pour tout $\sigma \in S_n(A)$.
Chaque élément de $S_n(A)$ est défini de manière unique par sa restriction à A (à valeurs dans $\{1, 2, ..., k-1\}$) et à $\widehat{A}$ (à valeurs dans $\{k+1, ..., n\}$).
Ces restrictions sont des bijections de A vers $\{1, 2, ..., k-1\}$ et de $\widehat{A}$ vers $\{k+1, ..., n\}$.
Inversement tout couple de bijection permet de définir une bijection σ de $A \cup \widehat{A}$ vers $E_{1,n}$ puis, en prolongeant par $\sigma(k) = 1$ en une permutation de $\{1, 2, ..., n\}$.
$S_n(A)$ est donc de cardinal $(k-1)!(n-k)!$, produit des cardinaux des ensembles de bijections.

Éléments de réponse pour l'exercice 3
Pour A fixé, chaque permutation de $[a_1; ...; a_{k-1}]$ est obtenue $(n-k)!$ fois par une permutation de $S_n(A)$, les différentes bijections de $\widehat{A}$ vers $\{k+1, ..., n\}$ définissant les mêmes permutations.
On a donc $\displaystyle\sum_{\sigma \in S_n(A)} C(\ell_{\sigma,g}) = (n-k)! \sum_{\ell_g} C(\ell_g) = (n-k)!(k-1)!\overline{C}(k-1)$.
De même $\displaystyle\sum_{\sigma \in S_n(A)} C(\ell_{\sigma,d}) = (k-1)! \sum_{\ell_d} C(\ell_d) = (k-1)!(n-k)!\overline{C}(n-k)$.
Comme il y a $\binom{k-1}{n-1}$ partie de $E_{1,n}$ à $k-1$ éléments on a

$$\begin{aligned}
\sum_{\sigma \in S_n,\ \sigma(1)=k} C(\ell_{\sigma,g}) &= \sum_{A \subset E_{1,n}, |A|=k-1} \sum_{\sigma \in S_n(A)} C(\ell_{\sigma,g}) \\
&= \sum_{A \subset E_{1,n}, |A|=k-1} (n-k)!(k-1)!\overline{C}(k-1) \\
&= \frac{(n-1)!}{(k-1)!(n-1-(k-1))!}(n-k)!(k-1)!\overline{C}(k-1) \\
&= (n-1)!\overline{C}(k-1)
\end{aligned}$$

De même $\displaystyle\sum_{\sigma \in S_n,\ \sigma(1)=k} C(\ell_{\sigma,d}) = (n-1)!\overline{C}(n-k)$.

$$\begin{aligned}
\overline{C}(n) &= \frac{1}{n!} \sum_{\sigma \in S_n} C(\ell_\sigma) = \frac{1}{n!} \sum_{\sigma \in S_n} \left((n-1) + C(\ell_{\sigma,g}) + C(\ell_{\sigma,d})\right) \\
&= n-1 + \sum_{k=1}^{n} \sum_{\sigma \in S_n,\ \sigma(1)=k} \frac{C(\ell_{\sigma,g}) + C(\ell_{\sigma,d})}{n!} \\
&= n-1 + \frac{1}{n!} \sum_{k=1}^{n} \sum_{\sigma \in S_n,\ \sigma(1)=k} C(\ell_{\sigma,g}) + \frac{1}{n!} \sum_{k=1}^{n} \sum_{\sigma \in S_n,\ \sigma(1)=k} C(\ell_{\sigma,d}) \\
&= n-1 + \frac{1}{n!} \sum_{k=1}^{n} (n-1)!\overline{C}(k-1) + \frac{1}{n!} \sum_{k=1}^{n} (n-1)!\overline{C}(n-k) \\
&= n-1 + \frac{1}{n} \sum_{k=1}^{n} \overline{C}(k-1) + \frac{1}{n} \sum_{k=1}^{n} \overline{C}(n-k)
\end{aligned}$$

Éléments de réponse pour l'exercice 4

$$\begin{aligned}
\overline{C}(n) &= n-1+\sum_{k=1}^{n}\frac{\overline{C}(k-1)}{n}+\sum_{k=1}^{n}\frac{\overline{C}(n-k)}{n}\\
&= n-1+\sum_{p=0}^{n-1}\frac{\overline{C}(p)}{n}+\sum_{q=0}^{n-1}\frac{\overline{C}(q)}{n} \text{ avec } p=k-1,\ q=n-k\\
&= n-1+\frac{2}{n}\sum_{k=0}^{n-1}\overline{C}(k) = n-1+\frac{2\overline{C}(n-1)}{n}+\frac{2}{n}\sum_{k=0}^{n-2}\overline{C}(k)\\
&= n-1+\frac{2\overline{C}(n-1)}{n}+\frac{2}{n}\frac{\overline{C}(n-1)-(n-2)}{\frac{2}{n-1}}\\
&= n-1-\frac{(n-1)(n-2)}{n}+\frac{\overline{C}(n-1)}{n}(2+n-1)\\
&= \frac{2(n-1)+\overline{C}(n-1)(n+1)}{n}
\end{aligned}$$

Pour déterminer $\overline{C}(n)$ vérifiant $n\overline{C}(n) = 2(n-1)+(n+1)\overline{C}(n-1)$ on commence par déterminer les suites vérifiant $nu_n = (n+1)u_{n-1}$, la relation sans second membre.
On trouve $u_n = \frac{n+1}{n}\frac{n}{n-1}\cdots\frac{3}{2}\frac{2}{1}u_0 = (n+1)u_0 : u_n = K.(n+1)$.
Puis on fait varier la constante, $\overline{C}(n) = (n+1)v_n$:
on obtient $v_n = v_{n-1}+\dfrac{2(n-1)}{n(n+1)} = v_{n-1}-2\dfrac{1}{n}+4\dfrac{1}{n+1}$.
On a $C(1) = 0$ donc $v_1 = 0$ puis $v_n = v_1+\displaystyle\sum_{k=2}^{n}\frac{2(k-1)}{k(k+1)} = -2\sum_{k=2}^{n}\frac{1}{k}+4\sum_{k=2}^{n}\frac{1}{k+1}$.
Si on pose $H_n = \displaystyle\sum_{k=1}^{n}\frac{1}{k} = \ln(n)+\gamma+\mathcal{O}(\frac{1}{n})$ où γ est la constante d'Euler on obtient
$u_n = -2(H_n-1)+4\left(H_n+\dfrac{1}{n+1}-1-\dfrac{1}{2}\right) = 2H_n-\dfrac{4n}{n+1}$
puis $\overline{C}(n) = 2(n+1)H_n-4n \sim 2n\ln(n)$.

Éléments de réponse pour l'exercice 5
Dans la fonction d'insertion de x dans une liste, chaque fois qu'il y a besoin de continuer à insérer c'est qu'il y avait une inversion dont un des termes est x. On va donc modifier la fonction en ajoutant un décompte.

```
let rec insertionC x liste =
  match liste with
  |[] -> [x], 0
  |t::q when t < x -> let q', n = insertionC x q in t::q', (n+1)
  |_ -> x::liste, 0;;
```

Il suffit alors de sommer les inversions

```
let nbInversions liste =
  let rec triInsertionC liste =
    match liste with
    |[] -> [], 0
    |t::q -> let triee, n = triInsertionC q in
             let l, k = insertionC t triee in
             n + k;;
```

Éléments de réponse pour l'exercice 6
Pour compter les inversions lors d'un tri-fusion on compte les inversions dans chacun des deux tris puis on compte les inversions qui apparaissent lors de la fusion. Mais on doit pour cela séparer en prenant les premiers éléments puis les autres. On aura besoin de connaître la taille des listes que l'on ajoute aux paramètres.
On commence par séparer une liste en passant le nombre d'éléments dans la première partie.

```
let rec separer liste nb =
  match nb, liste with
  |0, _ -> [], liste
  |k, [] -> failwith "Liste trop courte"
  |k, t::q -> let l1, l2 = separer q (k-1) in t::l1, l2;;
```

Lors de la fusion il y a p inversions lorsque le premier élément de la seconde liste est plus petit que les p éléments de la première.

```
let rec fusionC l1 l2 n1 = (* n1 est la taille de l1 *)
  match l1,l2 with
  |[],_ -> l2, 0
  |_,[] -> l1, 0
  |t1::q1,t2::q2 -> if t1 < t2
                   then let l, k = fusionC q1 l2 (n1-1)
                   in t1::l, k
                   else let l, k = fusionC l1 q2 n1
                   in t2::l, (k+n1);;
```

On peut alors combiner

```
let nbInversions liste =
  let rec triFusionC liste n = (* n est la taille de l *)
    match liste with
    |[] -> [], 0
    |[x] -> [x], 0
    |_ -> let l1, l2 = separer liste (n/2) in
          let lt1, i1 = triFusionC l1 (n/2) in
          let lt2, i2 = triFusionC l2 (n - n/2) in
          let l, k = fusionC lt1 lt2 (n/2) in
          l, k + i1 + i2 in
  snd (triFusionC liste (list_length liste));;
```

Éléments de réponse pour l'exercice 7

On commence par le découpage, on ajoute le décompte du premier sous-ensemble.

```
let rec filtrerC liste pivot =
  match liste with
  |[] -> [], [], 0
  |t::q -> let l1, l2, k = filtrerC q pivot in
           if t < pivot
             then t::l1, l2, k+1
             else l1, t::l2, k;;
```

On traduit ensuite l'algorithme décrit.

```
let rec selection liste p =
  match liste with
  |[] -> failwith "il n'y a pas assez d'éléments dans la liste"
  |t::q -> let l1, l2, k = filtrerC q t in
           if k = p -1 then t
                       else (if k < p-1
                               then selection l2 (p-1-k)
                               else selection l1 p);;
```

Éléments de réponse pour l'exercice 8

On va reproduire l'étude du tri pivot.

On note $C_p(n)$ la complexité moyenne de recherche du p-ième élément dans une liste de taille n et $C(n) = \frac{1}{n}\sum_{p=1}^{n} C_p(n)$ la complexité moyenne de recherche d'un élément dans une liste de taille n.

$C_p(n) = n - 1 + \frac{1}{n}\left(\sum_{k=0}^{p-1} C_{p-1-k}(n-k-1) + 0 + \sum_{k=p}^{n-1} C_p(k)\right)$ d'où

$nC(n) = \sum_{p=1}^{n}(n-1) + \frac{1}{n}\sum_{p=1}^{n}\sum_{k=0}^{p-2} C_{p-1-k}(n-k-1) + \frac{1}{n}\sum_{p=1}^{n}\sum_{k=p}^{n-1} C_p(k).$

On a $\sum_{p=1}^{n}(n-1) = n(n-1)$, $\sum_{p=1}^{n}\sum_{k=p}^{n-1} C_p(k) = \sum_{k=1}^{n-1}\sum_{p=1}^{k} C_p(k) = \sum_{k=1}^{n-1} kC(k)$

et, en posant $q = p-1-k$ puis $l = n-1-k$,

$\sum_{p=1}^{n}\sum_{k=0}^{p-2} C_{p-1-k}(n-k-1) = \sum_{k=0}^{n-2}\sum_{p=k+2}^{n} C_{p-1-k}(n-k-1) = \sum_{k=0}^{n-2}\sum_{q=1}^{n-1-k} C_q(n-k-1)$

$\sum_{p=1}^{n}\sum_{k=0}^{p-2} C_{p-1-k}(n-k-1) = \sum_{k=0}^{n-2}(n-1-k)C(n-k-1) = \sum_{l=1}^{n-1} lC(l)$

Ainsi $nC(n) = n(n-1) + \frac{2}{n}\sum_{k=1}^{n-1} kC(k)$ donc $n^2C(n) = n^2(n-1) + 2\sum_{k=1}^{n-1} kC(k)$.

En calculant la différence de termes consécutifs
$(n+1)^2C(n+1) - n^2C(n) = (n+1)^2 n + 2\sum_{k=1}^{n} kC(k) - n^2(n-1) - 2\sum_{k=1}^{n-1} kC(k)$
on obtient $(n+1)^2C(n+1) - n^2C(n) = n(3n-2) + 2nC(n)$
d'où $(n+1)^2C(n+1) = n(3n-2) + n(n+2)C(n)$ avec $C(1) = 0$.
On a $n^2 + 2n \leqslant n^2 + 2n + 1 = (n+1)^2$ et $n(3n-2) \leqslant 3n^2 + 6n + 3 = 3(n+1)^2$ donc $C(n+1) \leqslant 3 + C(n)$ puis $C(n) \leqslant 3(n-1)$.
On obtient un complexité linéaire, légèrement meilleure que celle d'un tri.
On pouvait faire plus précis en posant $u_n = \frac{n}{n+1}C(n)$. On a alors, en divisant par $(n+1)(n+2)$, $u_{n+1} = u_n + 3 + \dfrac{5}{n+1} - \dfrac{16}{n+2}$ et $u_1 = 0$.
On a donc $u_n = 3(n-1) + 5\sum_{k=2}^{n} \dfrac{1}{k} - 16\sum_{k=3}^{n+1} \dfrac{1}{k} = 3n - 9H_n + 4 - \dfrac{16}{n+1}$

6 Types de données

Éléments de réponse pour l'exercice 1

```
let resize t =
  let n = Array.length t in
  let tt = Array.make (2*n) t.(0) in
  for i =0 to (n-1) do tt.(i) <- t.(i) done;
  tt;;
```

On doit définir le type en rendant mutable le contenu.

```
type 'a stack = {mutable taille : int;
                 mutable contenu : 'a array};;
```

Seule la fonction `push` doit être changée

```
let push pile x =
  let n = pile.taille in
  if n = Array.length pile.contenu
    then pile.contenu <- (resize pile.contenu);
  pile.contenu.(n) <- x;
  pile.taille <- n+1;;
```

Éléments de réponse pour l'exercice 2

1. `isEmptyQueue (createQueue u)` renvoie `true`.
2. `isEmptyQueue (enque f x)` renvoie `false`.
3. `first (createQueue u)` renvoie une erreur.
4. `first (enque f x)` renvoie `x` **si `f` est la file vide**
   et sinon renvoie la même valeur que `first f`.
5. `deque (createQueue u)` renvoie une erreur.
6. `deque (enque f x)` renvoie une file vide **si `f` est la file vide**
   et sinon renvoie la même valeur que `enque (deque f) x`.

Éléments de réponse pour l'exercice 3

```
let nMax = 500;;

type 'a queue = {contenu : 'a array;
                 mutable premier : int;
                 mutable suivant : int;
                 mutable vide : bool};;

let createQueue u = {contenu = Array.make nMax u;
                     premier = 0;
                     suivant = 0;
                     vide = true};;

let isEmptyQueue file = file.vide;;
```

```
let enque file x =
  let n = Array.length file.contenu in
  let p = file.premier in
  let s = file.suivant in
  if s = p && not file.vide
  then failwith "La file est pleine"
  else begin file.contenu.(s) <- x;
             file.suivant <- (s+1) mod n;
             file.vide <- false end;;
```

```
let first file =
  if isEmptyQueue file
  then failwith "La file est vide"
  else let p = file.premier in
       file.contenu.(p);;
```

```
let deque file =
  let n = Array.length file.contenu in
  if isEmptyQueue file
  then failwith "La file est vide"
  else begin let p = file.premier in
             file.premier <- (p+1) mod n;
             if file.premier = file.suivant
             then file.vide <- true end;;
```

Éléments de réponse pour l'exercice 4

```
let createQueue u = None ;;

let isEmptyQueue f = f = None;;

let enque f x =
  match f with
  |None -> let rec c = {valeur = x; suivant = c} in Some c
  |Some c -> let c' = {valeur = x; suivant = c.suivant} in
  c.suivant <- c';
  Some c';;

let first f =
  match f with
  |None -> failwith "La file est vide"
  |Some c -> c.suivant.valeur;;

let deque f =
  match f with
  |None -> failwith "La file est vide"
  |Some c when c.suivant = c -> None
  |Some c -> c.suivant <- c.suivant.suivant;
                      Some c;;
```

Éléments de réponse pour l'exercice 5

```
let parentheses ch =
  let n = String.length ch in
  let p = createStack 0 in
  let par = ref [] in
  for i = 0 to (n-1) do
    if ch.[i] = '('
    then push p i
    else if ch.[i] = ')'
         then (par := (i, top p) :: !par; pop p)
  !par;;
```

Éléments de réponse pour l'exercice 6

```
let listeBinaire n =
  let f = createQueue "" in
  enque f "1";
  let rec aux k liste =
    if k = 0
    then List.rev liste
    else begin
           let ch  = first f in
           enque f (ch^"0");
           enque f (ch^"1");
           deque f;
           aux (k-1) (ch::liste)
         end in
  aux n [];;
```

Éléments de réponse pour l'exercice 7

```
let dicoVide u = {contenu = []};;
```

```
let ajouter dico cle valeur =
  let rec auxPlus liste =
    match liste with
    |[] -> [(cle, valeur)]
    |(k,x)::q when k = cle -> (cle, valeur) :: q
    |t::q -> t :: (auxPlus q) in
  dico.contenu <- (auxPlus dico.contenu);;
```

```
let defini dico cle =
  let rec cherche liste =
    match liste with
    |[] -> false
    |(k,x)::q when k = cle -> true
    |t::q -> cherche q in
  cherche dico.contenu;;
```

```
let association dico cle =
  let rec auxLire liste =
    match liste with
    |[] -> failwith "Clé non trouvée"
    |(k,x)::q when k = cle -> x
    |t::q -> auxLire q in
  auxLire dico.contenu;;
```

```
let estVide dico = dico.contenu = [];;
```

```
let enlever dico cle =
  let rec auxMoins liste =
    match liste with
    |[] -> []
    |(k,x)::q when k = cle -> q
    |t::q -> t :: (auxMoins q) in
  dico.contenu <- (auxMoins dico.contenu);;
```

```
let dict_to_list dico = dico.contenu;;
```

La complexité des fonctions est un $\mathcal{O}(n)$ où n est le nombre d'éléments dans le dictionnaire.

Éléments de réponse pour l'exercice 8

```
let tailleDico = 500;;
```

```
let dicoVide u = Array.make tailleDico None;;
```

```
let ajouter dico cle valeur =
  dico.(cle) <- Some(valeur);;
```

```
let defini dico cle =
  dico.(cle) <> None;;
```

```
let association dico cle =
  match dico.(cle) with
    |None -> failwith "Clé non trouvée"
    |Some(x) -> x;;
```

```
let estVide dico =
  let vide = ref true in
  let i = ref 0 in
  while !i < nMax && !vide do
    if dico.(!i) <> None then vide := false;
    i := !i + 1 done;
  !vide;;
```

```
let enlever dico cle =
  dico.(cle) <- None;;
```

```
let dict_to_list dico =
  let liste = ref [] in
  for i = 0 to nMax -1 do
    match dico.(i) with
    |None -> ()
    |Some(x) -> liste := (i,x) :: !liste done;
  !liste;;
```

Ici la complexité des premières fonctions est constante ; les fonctions supplémentaires sont en $\mathcal{O}(n)$.

Éléments de réponse pour l'exercice 9

```
let estVide dico =
  let vide = ref true in
  let i = ref 0 in
  while !i < nMax && !vide do
    if dico.contenu.(!i) <> [] then vide := false;
    i := !i + 1 done;
  !vide;;

let enlever dico cle =
  let i = dico.h cle in
  let rec auxMoins liste =
    match liste with
    |[] -> []
    |(k,x)::q when k = cle -> q
    |t::q -> t :: (auxMoins q) in
  dico.contenu.(i) <- (auxMoins dico.contenu.(i));;

let dict_to_list dico =
  let liste = ref [] in
  for i = 0 to nMax -1 do
    liste := dico.contenu.(i) @ !liste done;
  !liste;;
```

Éléments de réponse pour l'exercice 10

1.

```
let trouver_paires tab x =
  let n = Array.length tab in
  let paires = ref [] in
  for i = 0 to (n-2) do
    for j = (i+1) to (n-1) do
      if tab.(i) + tab.(j) = x
      then paires := (i,j) :: !paires done done;
  !paires;;
```

La complexité est quadratique

2.

```
let trouver_paires2 tab x =
  let n = Array.length tab in
  tri tab;
  let rec aux i j paires =
    if i < j
    then if tab.(i) + tab.(j) = x
         then aux (i+1) (j-1) ((i,j)::paires)
         else if tab.(i) + tab.(j) > x
              then aux i (j-1) paires
              else aux (i+1) j paires
    else paires
  in aux 0 (n-1) [];;
```

La complexité est maintenant linéaire pour la fonction mais il faut ajouter la complexité quasi-linéaire du tri.

3. `mod` peut renvoyer un entier négatif, on doit le modifier.

```
let modulo a b =
if a >= 0
then a mod b
else b + (a mod b);;
```

```
let trouver_paires3 tab x =
  let n = Array.length tab in
  let nMax = 2*n in
  let f x = modulo x nMax in
  let d = dicoVide f nMax in
  let liste = ref [] in
  for i = 0 to (n-1) do
    let k = x - tab.(i) in
    if defini d k
    then liste := (association d k, i) :: !liste
    else ajouter d tab.(i) i done;
  !liste;;
```

La complexité est linéaire.

7 Programmation dynamique

Éléments de réponse pour l'exercice 1
Pour calculer C_n, on calcule deux fois chaque C_k pour $0 \leqslant k < n$ on fait ensuite n produit et n somme.
On a $y(n) = u_n - u_{n-1} = 2u_{n-1} + 2n$, le résultat se trouve par récurrence.
$y(n) = u_n - u_{n-1} = 3^n - 1$.

Éléments de réponse pour l'exercice 2
On calcule chaque C_k une seule fois.
Pour n pair on calcule $n/2$ produits qu'on multiple par et qu'on additionne.
Pour n impair on le fait $(n-1)/2$ fois et on ajoute 2 opérations.

Éléments de réponse pour l'exercice 3
La nombre d'additions et de multiplications est $\sum_{p=1}^{n} 2p = n(n+1)$.

Éléments de réponse pour l'exercice 4

```
let rec produit liste1 liste2 =
  match liste1, liste2 with
  |t1::q1, t2::q2 -> (t1*t2) + (produit q1 q2)
  |_ -> 0;;

let catalan n =
  let rec aux_cat n =
    match n with
    |0 -> [1]
    |n -> let l = aux_cat (n-1) in
          (produit l (List.rev l)) :: l
 in List.hd (aux_cat n);;
```

Comme la complexité de `produit` est $2n$ et celle de `List.rev` est n pour une liste de taille n, la complexité est donc $\sum_{p=1}^{n} 3p = \frac{3n(n+1)}{2}$.

Éléments de réponse pour l'exercice 5
1. La fonction `minFin` sert à déterminer l'indice et la fin d'une demande de fin minimale dans une liste.

```
let rec finMinimale demandes =
  match demandes with
  |[] -> failwith "Pas de minimum"
  |[a] -> (a.num, a.fin)
  |t::q -> let (i, f) = finMinimale q in
           if t.fin < f
           then (t.num, t.fin)
           else (i, f);;
```

`filtrer` ne conserve d'une liste que les éléments qui vérifient une condition

```
let rec filtrer condition liste =
  match liste with
    |[] -> []
    |t::q -> if condition t
             then t::(filtrer condition q)
             else filtrer condition q;;
```

```
let rec planning demandes =
  match demandes with
    |[] -> []
    |_ -> let (i, f) = finMinimale demandes in
          let condition demande = demande.debut >= f in
             i::(planning (filtrer condition demandes));;
```

2. La preuve de l'algorithme se fait par récurrence sur la taille de la liste de demandes.
   Pour une liste vide la solution optimale est vide, c'est ce que renvoie l'algorithme.
   On suppose que la solution est optimale pour toute liste de taille n au plus.
   On dispose d'une liste de $n+1$ demandes et on a une solution optimale $(i_1, i_2, ..., i_p)$.
   La solution renvoyé par l'algorithme est $(j_1, j_2, ..., j_q)$ et on veut montrer qu'on a $q = p$.
   La demande d'indice j_1 finit la première donc elle finit avant toutes les demandes de la solution optimale. Toutes les demandes d'indices i_2 à i_p sont donc filtrées (par la fonction `filtrer`) et elles forment une solution optimale pour l'ensemble des demandes filtrées car, sinon on aurait, i_1 une solution de taille supérieure à p.
   Les demandes filtrées forment un ensemble de demande de taille n au plus donc la solution de l'algorithme est optimale : elle comporte $p-1$ demandes.
   On a donc trouvé une solution de taille p, elle est optimale.
3. Pour une liste de demandes de taille n la solution fait au plus n appels récursifs.
   À la k-ième étape on traite une liste filtrée de taille $n+1-k$ au plus dans laquelle on recherche un minimum puis on filtre : on a une complexité en $\mathcal{O}(n)$ à chaque fois.
   La complexité totale est donc un $\mathcal{O}(n^2)$.

Éléments de réponse pour l'exercice 6
On lit les demandes en choisissant celles qui peuvent suivre.

```
let planning demandes =
  let rec auxPlan liste temps =
    match liste with
    |[] -> []
    |t::q -> if t.debut < temps
             then auxPlan q temps
             else t.num::(auxPlan q t.fin)
  in auxPlan (tri demandes) 0;;
```

La complexité du tri est en $O(n\log_2(n))$. La suite parcourt la liste donc sa complexité est en $O(n)$. on aboutit à une complexité en $O(n\log_2(n))$

Éléments de réponse pour l'exercice 7
L'algorithme est celui vu dans le cours : on sépare, on trie les morceaux, on fusionne

```
let rec tri tab compare =
  match Array.length tab with
  |0 -> [| |]
  |1 -> [|tab.(0)|]
  |n -> let p = n/2 in
        let t1 = tri (Array.sub tab 0 p) compare in
        let t2 = tri (Array.sub tab p (n-p)) compare in
        fusion t1 t2 compare;;
```

La fusion est, elle aussi, classique. On suppose que les tableaux à fusionner sont triés et non vides (le premier surtout).

```
let fusion t1 t2 compare =
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let t = Array.make (n1+n2) t1.(0) in
  let i = ref 0 in
  let i1 = ref 0 in
  let i2 = ref 0 in
  while !i1 < n1 && !i2 < n2 do
    if compare t1.(!i1) t2.(!i2)
    then (t.(!i) <- t1.(!i1); i1 := !i1 + 1)
    else (t.(!i) <- t2.(!i2); i2 := !i2 + 1);
    i := !i +1 done;
  for k = !i1 to (n1-1) do t.(!i) <- t1.(k); i := !i +1 done;
  for k = !i2 to (n2-1) do t.(!i) <- t2.(k); i := !i +1 done;
  t;;
```

Éléments de réponse pour l'exercice 8

On veut la dernière position dont la date de fin précède le début, il suffit de parcourir le tableau.

```
let calculRho tab =
  let n = Array.length tab in
  let rho = Array.make n None in
  (* La première demande n'a pas de prédécesseur *)
  for k = 1 to (n-1) do
    for i = 0 to (k-1) do
      if tab.(i).fin <= tab.(k).debut
      then rho.(k) <- (Some i) done done;
  rho;;
```

Une double boucle donne une complexité quadratique qui dépasse celle du tri.

Éléments de réponse pour l'exercice 9

```
let optimum tab =
  let n = Array.length tab in
  let tabOrd = tri tab inferieur in
  let rho = calculRho tabOrd in
  let opt = make_vect n 0 in
  opt.(0) <- tabOrd.(0).valeur;
  for i = 1 to (n-1) do
    let plus = match rho.(i) with
      |None -> 0
      |Some k -> opt.(k) in
    opt.(i) <- max opt.(i-1) (plus + tabOrd.(i).valeur) done;
  opt.(n-1);;
```

La complexité est celle de `rho` : $\mathcal{O}(n^2)$.

Éléments de réponse pour l'exercice 10

```
let optimumListe tab =
  let n = vect_length tab in
  let tabOrd = tri tab inferieur in
  let rho = calculRho tabOrd in
  let opt = make_vect n (0, []) in
  opt.(0) <- (tabOrd.(0).valeur, [tabOrd.(0).num]);
  for i = 1 to (n-1) do
    let plus, vu = match rho.(i) with
                   |None -> (0, [])
                   |Some k -> opt.(k) in
    let v, l = opt.(i-1) in
    if v > plus + tabOrd.(i).valeur
    then opt.(i) <- opt.(i-1)
    else opt.(i) <- (plus + tabOrd.(i).valeur, (tabOrd.(i).num)::vu)
done;
  opt.(n-1);;
```

Éléments de réponse pour l'exercice 11

↪ Si on avait $x_n = y_m$ avec $z_p \neq x_n$ alors on pourrait ajouter x_n à Z pour obtenir une sous-séquence commune de X et Y de longueur plus grande ce qui contredirait l'hypothèse de maximalité. On a donc $z_k = x_n = y_m$.
Z' est une sous-séquence commune de longueur p de X' et Y'.

Toute PLSSC de X' et Y' de longueur q permet de construire, avec x_n, une PLSSC de X et Y de longueur $q+1$: on doit avoir $q \leqslant p$ donc Z' est bien une PLSSC de X' et Y'.
↪ Si on a $x_n \neq y_m$ alors on doit avoir $z_p \neq x_n$ ou $z_p \neq y_m$.
Si on a $z_p \neq x_n$ alors Z est une sous-séquence commune de X' et de Y. S'il existait une sous-séquence commune plus longue elle serai une sous-séquence commune de X et de Y ce qui est contradictoire. Z est bien une PLSSC de X' et Y.
De même si $z_k \neq y_m$ alors Z est bien une PLSSC de X et Y'.

Éléments de réponse pour l'exercice 12
On va maintenir une matrice qui contient une PLSSC des chaînes extraites.

```
let plssc mot1 mot2 =
   let n1 = String.length mot1 in
   let n2 = String.length mot2 in
   let sousSuites = Array.make_matrix (n1+1) (n2+1) "" in
   for i = 1 to n1 do
      for j = 1 to n2 do
         if mot1.[i-1] = mot2.[j-1]
         then sousSuites.(i).(j) <- sousSuites.(i-1).(j-1)^
                                                    (String.make 1 mot1.[i-1])
         else if String.length sousSuites.(i-1).(j)
                    < String.length sousSuites.(i).(j-1)
               then sousSuites.(i).(j) <- sousSuites.(i).(j-1)
               else sousSuites.(i).(j) <- sousSuites.(i-1).(j)
       done
   done;
   sousSuites.(n1).(n2);;
```

Éléments de réponse pour l'exercice 13

```
let sacAdos objets wMax =
  let n = Array.length objets in
  let opt = Array.make_matrix (n+1) (wMax+1) 0 in
  for k = 1 to n do
    let l = k - 1 in
    let poids, valeur = objets.(l) in
    for w = 0 to wMax do
      if poids <= w
      then let v = opt.(l).(w-poids) + valeur in
           opt.(k).(w) <- max opt.(l).(w) v
      else opt.(k).(w) <- opt.(l).(w) done done;
  opt.(n).(wMax);;
```

Éléments de réponse pour l'exercice 14

```
let sacAdos objets wMax =
  let n = Array.length objets in
  let opt = Array.make_matrix (n+1) (wMax+1) (0,[]) in
  for k = 1 to n do
    let l = k - 1 in
    let poids, valeur = objets.(l) in
    for w = 0 to wMax do
      if poids <= w
      then let (v,ls) =  opt.(l).(w-poids) in
           if v + valeur > fst (opt.(l).(w))
           then opt.(k).(w) <- (v + valeur, l::ls)
           else opt.(k).(w) <- opt.(l).(w)
      else opt.(k).(w) <- opt.(l).(w) done done;
  opt.(n).(wMax);;
```

Éléments de réponse pour l'exercice 15

```
let dim m =
  let n = Array.length m in
  if n = 0
  then (0, 0)
  else (n, Array.length m.(0));;
```

```
let prod m1 m2 =
  let n1, p1 = dim m1
  and n2, p2 = dim m2 in
  if p1 <> n2 then failwith "Tailles non compatibles";
  let m =Array. make_matrix n1 p2 0 in
  for i = 0 to (n1-1) do
    for j = 0 to (p2-1) do
      for k = 0 to (p1-1) do
        m.(i).(j) <- m.(i).(j) + m1.(i).(k) * m2.(k).(j)
      done done done;
  m;;
```

La fonction effectue $n_1 p_1 p_2$ produits d'entiers.

Éléments de réponse pour l'exercice 16
Si M_1 et M_3 sont de taille $2xn$ et M_2 de taille $nx2$ alors

↪ $(M_1.M_2).M_3$ demande $(2.n.2).2.n = 8n^2$ multiplications
↪ $M_1.(M_2.M_3)$ demande $2.n.(n.2.n) = 4n^3$ multiplications

Le rapport entre les coûts des deux possibilités est $n/2$.

Éléments de réponse pour l'exercice 17
Pour calculer $M_i.M_{i+1}.\cdots.M_{j-1}.M_j$ la dernière opération est un produit de deux blocs $(M_i.M_{i+1}.\cdots.M_k).(M_{k+1}.\cdots.M_{j-1}.M_j)$ avec $k \in \{i, i+1, \ldots, j-1\}$.
Le nombre optimal d'opérations pour chaque bloc est $p_{i,k}$ et $p_{k+1,j}$ et les tailles sont $d_{i-1} \times d_k$ et $d_k \times d_j$ donc $p_{i,k} + p_{k+1,j} + d_{i-1}d_kd_j$ est le nombre d'opérations optimal pour cette décomposition.
On calcule ensuite le minimum selon k.

Éléments de réponse pour l'exercice 18
On commence par une fonction `min_fun` prenant pour arguments une fonction `f` de type `int -> int` et deux entiers `i <= j` et qui calcule l'entier `k` entre `i` et `j` tel que `f k` soit minimal.

```
let rec min_fun f i j =
  if i = j
  then i
  else let k = min_fun f (i + 1) j in
       if f i < f k
       then i
       else k;;

let min_fun f i j =
  let rep = ref i in
  for k = (i+1) to j do
    if f k < f !rep
    then rep := k done;
  !rep;;
```

On construit la matrice triangulaire des $p(i,j)$ $(i \leqslant j)$ en diagonale en augmentant l'écart entre i et j.
Quand $i = j$ (l'écart h vaut 0) $p(i,i) = 0$ qui est la valeur initiale

```
let optprod d =
  let n = Array.length d -1  in
  let p = Array.make_matrix (n+1) (n+1) 0 in
  for h = 1 to (n-1) do
    for i = 1 to (n-h) do
      let j = h + i in
      let f k =  p.(i).(k) + p.(k+1).(j) + d.(i-1)*d.(k)*d.(j) in
      let k = min_fun f i (j-1) in
      p.(i).(j) <- f k done done;
  p.(1).(n);;
```

Deux boucles de longueur au plus n imbriquées et `min_fun` qui effectue au plus n comparaisons : la complexité est en $O(n^3)$.

Éléments de réponse pour l'exercice 19
On calcule les deux matrice puis on construit récursivement la chaîne correspondant.

```
let optprodS d =
  let n = Array.length d -1  in
  let p = Array.make_matrix (n+1) (n+1) 0 in
  let coupe = Array.make_matrix (n+1) (n+1) 0 in
  for h = 1 to (n-1) do
    for i = 1 to (n-h) do
      let j = h + i in
      let f k =  p.(i).(k) + p.(k+1).(j) + d.(i-1)*d.(k)*d.(j) in
      let k = min_fun f i (j-1) in
      p.(i).(j) <- f k;
      coupe.(i).(j) <- k done done;
  let rec ecrire i j =
    if i = j
    then "M"^(string_of_int i)
    else let k = coupe.(i).(j) in
         "("^(ecrire i k)^"."^(ecrire (k+1) j)^")" in
  let s = ecrire 1 n in
  let n = String.length s in
  sub_string s 1 (n-2);;
```

8 Arbres

Éléments de réponse pour l'exercice 1
On va avoir besoin d'envoyer la profondeur comme paramètre : on utilise une fonction auxiliaire.

```
let profondeur arbre =
   let rec aux_prof a h =
      match a with
      |F -> F
      |Noeud(g, r, d) -> Noeud(aux_prof g (h+1), (r, h), aux_prof d
(h+1))
   in aux_prof arbre 0;;
```

Éléments de réponse pour l'exercice 2

```
let rec prefixe a =
  match a with
  |F -> ()
  |Noeud (g,r,d) -> traiter r;
                    prefixe g;
                    prefixe d;;
```

```
let rec postfixe a =
  match a with
  |F -> ()
  |Noeud (g,r,d) -> postfixe g;
                    postfixe d;
                    traiter r;;
```

Éléments de réponse pour l'exercice 3

1. Si $h = -1$, l'arbre est vide et il n'y a rien à prouver.
   On suppose $h \geqslant 0$ et on utilise les notations et les résultats de l'exercice **8**.
   Un arbre est complet si et seulement si le nombre de feuille de profondeur k, F_k, vérifie $F_k = 0$ pour $k \leqslant h$.
   Comme on a $N_{k+1} + F_{k+1} = 2N_k$ pour tout $k \geqslant 0$ on en déduit qu'un arbre est complet si et seulement si $N_{k+1} = 2N_k$ pour tout $k \leqslant h-1$. Comme on a $N_0 = 1$ c'est donc équivalent à $N_k = 2^k$ pour $k \leqslant h$.
2. On sait qu'on a $N_k \leqslant 2^k$ donc la taille n vérifie $n = \sum_{k=0}^{h} N_k \leqslant \sum_{k=0}^{h} 2^k = 2^{h+1} - 1$ avec égalité si et seulement si $N_k = 2^k$ pour tout k, c'est-à-dire si et seulement si l'arbre est complet.

3. On va procéder par récurrence sur la hauteur de l'arbre.
   $\mathcal{P}(h)$ est la propriété : un arbre de hauteur h est complet si et seulement si, pour tout nœud de l'arbre, les deux fils ont la même hauteur.
   $\mathcal{P}(-1)$ est vérifiée car tous les arbres de hauteur -1 sont des feuilles donc sont complets et vérifient toute propriétés sur leurs fils car ils n'ont pas de fils.
   On suppose que $\mathcal{P}(h)$ est vérifiée avec $h \geqslant -1$.
   a est un arbre de hauteur $h+1$, ses fils droit et gauche sont notés d et g respectivement.
   a. Si a est complet alors ses feuilles, donc les feuilles de g et d, sont à la profondeur $h+2$. Ainsi les feuilles de g sont à la profondeur $h+1$ dans g donc g est complet ; de même d est complet.
      g et d sont de hauteur h au plus mais ils admettent des feuilles à la profondeur $h+1$ donc ils sont de hauteur h au moins. Ainsi g et h sont complets de hauteur h.
      On peut appliquer l'hypothèse de récurrence : tous les nœuds de g et d ont des fils de même hauteur. Comme g et d, fils de la racine de a, ont même hauteur on en déduit que tous les nœuds de a ont des fils de même hauteur.
   b. Si tous les nœuds de a ont des fils de même hauteur alors g et d ont même hauteur h et ont des nœuds qui ont des fils de même hauteur. D'après l'hypothèse de récurrence g et d sont complets : leurs feuilles sont à la profondeur $h+1$.
      Les feuilles de a sont les feuilles de g et d et sont à la profondeur augmentée de 1 : $h+2$. On en déduit que a est complet.
   Ainsi $\mathcal{P}(h+1)$ se déduit de $\mathcal{P}(h)$ donc $\mathcal{P}(h)$ est vraie pour tout h.

Éléments de réponse pour l'exercice 4
On suppose $h \geqslant 0$ et on utilise les notations et les résultats de l'exercice **8**.
Un arbre est complet si et seulement si le nombre de feuille de profondeur k, F_k, vérifie $F_k = 0$ pour $k < h$. On en déduit qu'on a $N_k = 2^k$ pour $k < h$.

Éléments de réponse pour l'exercice 5
On reprend les notations et résultats de l'exercice **8**.
Comme dans l'exercice ?? le nombre de nœuds à la profondeur k est $N_k = 2^k$ pour $k \leqslant h-1$.
Pour la profondeur h on a $N_h + F_h = 2.2^{h-1} = 2^h$.
Pour la profondeur $h+1$ on a $N_{h+1} = 0$ (car la hauteur est h) donc $F_{h+1} = 2N_h$.
Le nombre de feuilles est donc $F_h + F_{h+1} = 2^h - N_h + 2N_h = 2^h + N_h$.
De plus on a $N_h \leqslant 2^h$ et, comme la hauteur est h, on a $N_h \geqslant 1$.
Ainsi le nombre de feuilles vérifie $2^h + 1 \leqslant F_h + F_{h+1} = 2^h + N_h \leqslant 2^h + 2^h = 2^{h+1}$.
La taille est égale au nombre de feuille diminué de 1 donc
$2^h \leqslant n = F_h + F_{h+1} - 1 \leqslant 2^{h+1} - 1 < 2^{h+1}$.
Le logarithme donne $h \leqslant \log_2(n) < h-1$ d'où $h = \lfloor \log_2(n) \rfloor$.

Éléments de réponse pour l'exercice 6
a est un arbre quasi-complet de hauteur $h \geqslant 0$, il n'est pas réduit à une feuille.
Ses fils on des hauteurs majorées par $h-1$.

De plus les feuilles de a sont à la profondeur h ou $h+1$ donc les feuilles des fils sont à la profondeur $h-1$ ou h. 2 cas sont alors possibles pour chaque fils :
1. soit il n'a que des feuilles à la profondeur $h-1$ donc il est complet de hauteur $h-2$, cela n'est possible que pour un seul des fils,
2. soit il admet des feuilles à la profondeur h donc il est quasi-complet de hauteur $h-1$.

Un des fils est de hauteur $h-1$, l'autre est de hauteur $h-1$ ou $h-2$ et les deux sont quasi-complets.
L'inégalité de l'énoncé est donc valide pour les fils de la racine et est vraie pour les autres nœuds par récurrence sur la hauteur.

Éléments de réponse pour l'exercice 7
On note $\mathcal{P}(h)$ la propriété :
tout arbre équilibré de hauteur h contient au moins $F_{h+2}-1$ nœuds.
Un arbre de hauteur 0 est un nœud de fils vides donc contient 1 nœud et $F_2-1=1$.
Un arbre de hauteur 1 est un nœud dont au moins un fils est non vide : il contient au moins 2 nœuds et $F_3-1=2$. Ainsi $\mathcal{P}(0)$ et $\mathcal{P}(1)$ sont vraies.
On suppose que $\mathcal{P}(h-1)$ et $\mathcal{P}(h)$ sont vraies avec $h \geqslant 1$.
a est un arbre équilibré de hauteur $h+1$.
Ses deux fils sont de hauteur h au plus et au moins l'un des deux est de hauteur h.
Comme l'arbre est équilibré l'autre fils est de hauteur $h+\varepsilon$ avec $|\varepsilon| \leqslant 1$ donc il est de hauteur $h-1$ au moins.
De plus les deux fils sont équilibrés donc l'un contient au moins $F_{h+2}-1$ nœuds et l'autre contient au moins $F_{h-1+2}-1$ nœuds.
On en déduit que la taille de a est minorée par $1+F_{h+2}-1+F_{h+1}-1=F_{h+3}-1$.
Ainsi $\mathcal{P}(h+1)$ est vrai donc la propriété est vraie pour tout arbre.

Les racines de X^2-X-1 sont $\alpha=\frac{1+\sqrt{5}}{2}$ et $\beta=\frac{1-\sqrt{5}}{2}$ et $F_n=\frac{1}{\sqrt{5}}(\alpha^{n+1}-\beta^{n+1})$ donc, pour un arbre équilibré on a $n \geqslant \frac{1}{\sqrt{5}}(\alpha^{h+3}-\beta^{h+3})-1$. On en déduit
$$\frac{n}{\alpha^h} \geqslant \frac{\alpha^3}{\sqrt{5}} - \left(\frac{\beta}{\alpha}\right)^h \frac{\beta^3}{\sqrt{5}} - \frac{1}{\alpha^h} \geqslant \frac{\alpha^3}{\sqrt{5}} - \left(\frac{\beta}{\alpha}\right)^h \frac{\beta^3}{\sqrt{5}} - 1 = F_2 - 1 = 1$$
d'où $\alpha^h \leqslant n$ puis $h \leqslant \dfrac{1}{\log_2(\alpha)} \log_2(n)$.

Éléments de réponse pour l'exercice 8
Chaque nœud de la profondeur k a deux fils qui sont à la profondeur $k+1$ et qui sont des nœuds ou des racines d'où $N_{k+1}+F_{k+1}=2.N_k$.
Si l'arbre est réduit à une feuille alors $F_0=1$ et $N_0=0$, si l'arbre admet au moins un nœud alors la racine est un nœud donc $F_0=0$ et $N_0=1$.

Si la hauteur est h on a $N_{h+1}=0$. De plus $N_k=2N_{k-1}-N_k$ pour $1 \leqslant k \leqslant h+1$ d'où

$$\sum_{k=0}^{h+1} F_k 2^{-k} \& = F_0 + \sum_{k=1}^{h+1} (2N_{k-1} - N_k)2^{-k} = F_0 + \sum_{k=1}^{h+1} N_{k-1}2^{-k+1} - \sum_{k=1}^{h+1} N_k 2^{-k}$$
$$\& = F_0 + \sum_{k=0}^{h} N_k 2^{-k} - \sum_{k=1}^{h+1} N_k 2^{-k} = F_0 + N_0 - N_{h+1}2^{-h-1} = F_0 + N_0 = 1$$

Éléments de réponse pour l'exercice 9

1. La longueur de cheminement des nœuds d'un fils est celle de ces nœuds dans le fils augmentée de 1 pour tout nœud car on doit ajouter la longueur entre la racine du fils et la racine de l'arbre. On va employer une fonction auxiliaire qui renvoie la longueur de cheminement et la taille.

```
let longChem arbre =
  let rec auxLC a =
    match a with
    |Feuille _ -> 0, 0
    |Noeud(g,_,d) -> let lg, ng = auxLC g in
                     let ld, nd = auxLC d in
                     lg + ld + ng + nd, ng + nd + 1
  in fst (auxLC arbre);;
```

2. On note $h_1, h_2, \ldots, h_n$ les profondeurs des n nœuds de l'arbre en parcourant ceux-ci par le parcours en largeur. Ces profondeurs forment donc une suite croissante : $h_i \leqslant h_{i+1}$ pour $1 \leqslant i \leqslant n-1$.
   Il y a au moins un nœud à chaque profondeur donc l'accroissement entre deux profondeurs successives est majoré par 1 : $h_{i+1} \leqslant h_i + 1$. On en déduit, par récurrence sur k, que $h_k \leqslant h_1 + k - 1 = k - 1$.
   Ainsi $LC = \sum_{k=1}^{n} h_k \leqslant \sum_{k=1}^{n} k - 1 = \dfrac{n(n-1)}{2}$.
   On atteint le cas maximal quand $h_k = k - 1$ pour tout k, c'est-à-dire s'il y a un seul nœud par profondeur ; c'est le cas des arbres réduit à une liste.

   On sait de plus qu'il y a au plus 2^k nœuds à la profondeur k.
   On en déduit que h_{i+2^k} vaut au moins $k+1$ si $h_i = k$.
   On a $h_1 = 0$ donc $h_2 = h_{1+1} = h_{1+2^0} \geqslant 1$ (en fait h_1 vaut 1).
   Si on suppose $h_{2^p} \geqslant p$ alors
   ↪ soit $h_{2^p} = p$ donc $h_{2^{p+1}} = h_{2^p+2^p} \geqslant p+1$,
   ↪ soit $h_{2^p} \geqslant p+1$ donc $h_{2^{p+1}} \geqslant h_{2^p} \geqslant p+1$.
   Dans les deux cas on a $h_{2^{p+1}} \geqslant p+1$ donc, par récurrence sur p, $h_{2^p} \geqslant p$ pour tout $p \in \mathbb{N}$.
   On a alors, pour $2^p \leqslant k < 2^{p+1}$, $h_k \geqslant h_{2^p} \geqslant p = \lfloor \log_2(k) \rfloor$ donc
   $LC = \sum_{k=1}^{n} h_k \geqslant \sum_{k=1}^{n} \lfloor \log_2(k) \rfloor$. Le cas d'égalité sera vu plus tard.

Éléments de réponse pour l'exercice 10

```
let rec listePref a =
  match a with
  |F -> []
  |Noeud (g,r,d) -> r ::((listePref g)@(listePref d));;
```

```
let rec listeInf a =
  match a with
  |F -> []
  |Noeud (g,r,d) -> (listeInf g)@(r ::(listeInf d));;
```

```
let rec listePost a =
  match a with
  |F -> []
  |Noeud (g,r,d) -> ((listePost g)@(listePost d))@[r];;
```

Éléments de réponse pour l'exercice 11

Pour un arbre quasi-complet de taille n, la hauteur est $h = \lfloor \log_2(n) \rfloor$ et il y a 2^k nœuds à la profondeur k pour $k < h$. Il reste $n - \sum_{k=0}^{h-1} 2^k = n - 2^h + 1$ nœuds à la profondeur h.

La longueur de cheminement est ainsi $LC = \sum_{k=0}^{h-1} 2^k.k + h(n - 2^h + 1)$.

On remarque que $\sum_{k=0}^{h-1} 2^k.k = \sum_{k=1}^{h-1} 2^k.k = P(2)$ avec $P(X) = \sum_{k=1}^{h-1} kX^k = X.Q(X)$

où $Q(X) = \sum_{k=1}^{h-1} kX^{k-1}$ est la dérivée de $\sum_{k=0}^{h-1} X^k = \frac{X^h - 1}{X - 1}$.

Ainsi $P(X) = X\frac{hX^{h-1}(X-1) - (X^h - 1)}{(X-1)^2}$ donc

$LC = P(2) + h(n - 2^h + 1) = 2h2^{h-1} - 2(2^h - 1) + hn - h2^h + h = (n+1)h - 2^{h+1} + 2.$

Si h_1, h_2, ..., h_n sont les profondeurs des n nœuds de l'arbre (quasi-complet) parcouru en largeur. La complétude de l'arbre implique

$h_1 = 0$, $h_2 = h_3 = 1$, ..., $h_{2^k} = h_{2^k+1} = \cdots = h_{2^{k+1}-1} = k$, ...donc $h_p = \lfloor \log_2(p) \rfloor$.

On en déduit que $LC = \sum_{k=1}^{n} h_k = \sum_{k=1}^{n} \lfloor \log_2(k) \rfloor$.

Ainsi les arbres quasi-complets réalisent le minimum de la longueur de cheminement.

Inversement si un arbre vérifie $LC = \sum_{k=1}^{n} h_k = \sum_{k=1}^{n} \lfloor \log_2(k) \rfloor$ alors, comme on a $h_k \geqslant \lfloor \log_2(k) \rfloor$, on doit avoir $h_k = \lfloor \log_2(k) \rfloor$ pour tout k.

On en déduit que $h_k = p$ pour $2^p \leqslant k < 2^{p+1}$ et $k \leqslant n$ donc qu'il y a 2^p nœuds de profondeur p pour toute profondeur atteinte sauf la dernière. Cela caractérise les arbres quasi-complets.

Éléments de réponse pour l'exercice 12
Si $N(i)$ et $N(i+1)$ ont chacun deux nœuds pour fils ces 4 nœuds se suivent dans le parcours en largeur : si les nœuds fils de $N(i)$ sont $N(p)$ et $N(p+1)$ alors ceux de $N(i+1)$ sont $N(p+2)$ et $N(p+3)$.
Comme les fils de $N(1)$ sont $N(2)$ et $N(3)$ on en déduit, par récurrence sur i, que les fils de $N(i)$ sont $N(2i)$ et $N(2i+1)$.
Ainsi $N(i)$ a deux fils qui sont des nœuds si on a $2i+1 \leqslant n$ et $N(i)$ n'admet qu'un seul fils qui est un nœud (le fils gauche) si $2i = n$.
On en déduit que le père de $N(i)$ est $N(j)$ avec $j = \left\lfloor \frac{i}{2} \right\rfloor$.

Éléments de réponse pour l'exercice 13
On a besoin de la hauteur, plutôt que la calculer à chaque étape (ne pas répéter) on va écrire une fonction qui renvoie la hauteur et le caractère équilibré.

```
let test_equilibre a =
  let rec aux_equ a =
    match a with
    |F -> true, -1
    |Noeud(g, _, d) -> let eg, hg = aux_equ g in
                       let ed, hd = aux_equ d in
                       eg && ed, 1 + max hg hd
  in fst (aux_equ a);;
```

Éléments de réponse pour l'exercice 14

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

Éléments de réponse pour l'exercice 15
↪ Si h_n est la hauteur de fibo(n) on a $h_{n+2} = 1 + \max(h_{n+1}, h_n)$ avec $h_0 = -1$ et $h_1 = 0$.
On en déduit, par récurrence sur n que $h_n = n - 1$. fibo(n) admet donc une feuille à la profondeur n.
Ainsi les hauteurs des fils différent de 1 pour tout nœud, sauf ceux qui sont des fibo(1) dont les deux fils ont la même hauteur -1 ; un arbre de Fibonacci est donc équilibré.
↪ Si n_p est la taille de fibo(p) on a $n_{p+2} = 1 + n_{p+1} + n_p$ avec $n_0 = 0$ et $n_1 = 1$.

Si on pose $N_p = n_p + 1$ on a $N_{p+2} = N_{p+1} + N_p$ avec $N_0 = 1$ et $N_1 = 2$; on retrouve la récurrence de la suite de Fibonacci avec $N_0 = F_1$ et $N_1 = F_2$ donc $N_p = F_{p+1}$ et $n_p = F_{p+1} - 1$.

↪ Si F_n a une feuille à la profondeur $k = \lceil \frac{n}{2} \rceil$ alors F_{n+2}, qui admet F_n comme fils droit, a une feuille à la profondeur $k + 1 = \lceil \frac{n}{2} \rceil + 1 = \lceil \frac{n+2}{2} \rceil$.
On en déduit le résultat souhaité par récurrence sur n car fibo(1) a une feuille à la profondeur $1 = \lceil \frac{1}{2} \rceil$ et fibo(2) a une feuille à la profondeur $1 = \lceil \frac{2}{2} \rceil$.

Éléments de réponse pour l'exercice 16

```
let rec taille_gen a =
  let rec aux_taille liste =
    match liste with
    |[] -> 0
    |t::q -> taille_gen t + (aux_taille q) in
  match a with
  |N(a, fils) -> 1 + aux_taille fils;;
```

On peut aussi utiliser la double récursivité du type.

```
let rec taille_gen a =
  let N(k, fils) = a in
  1 + taille_foret fils
and taille_foret liste =
  match liste with
  |[] -> 0
  |t::q -> (taille_gen t) + (taille_foret q);;
```

Mêmes constructions pour la hauteur.

```
let rec hauteur_gen a =
  let rec aux_hauteur liste =
    match liste with
    |[] -> -1
    |t::q -> max (hauteur_gen t) (aux_hauteur q) in
  match a with
  |N(a, fils) -> 1 + aux_hauteur fils;;
```

```
let rec hauteur_gen a =
  let N(k, fils) = a in
  1 + hauteur_foret fils
and hauteur_foret liste =
  match liste with
  |[] -> -1
  |t::q ->  max (hauteur_gen t) (hauteur_foret q);;
```

Éléments de réponse pour l'exercice 17

```
let rec foret2bin f =
  match f with
  |[] -> Vide
  |N(r,fils)::ff -> Noeud(foret2bin fils, r, foret2bin ff);;

let rec bin2foret a =
  match a with
  |F -> []
  |Noeud(g,r,d) -> N(r, bin2foret g) :: (bin2foret d);;
```

Liste des programmes

Liste des définitions

Liste des figures