Chapitre 1

De Python à OCaml

Option informatique MP1, MP2 & MP3

I Introduction

I.1 Où en sommes nous ?

I.1.a 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.

I.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 fiables : les langages modernes contiennent souvent une partie fonctionnelle.
C'est cette méthode de programmation que nous allons étudier avec le langage OCaml.

I.2.a Propriétés de OCaml

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.

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 "+."

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`.

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

II Utilisation de OCaml

II.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`.

II.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 4.10.2
#
```

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.

II.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`, `try.ocaml`, `emacs`) les instructions sont ajoutées dans une boucle interactive.
- `emacs` permet d'écrire directement dans la console.
- Au contraire `gedit, visual studio code` 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.

III 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.

III.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`.

III.2 Types simples

III.2.a 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`.

III.2.b 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.

III.2.c Booléens

Les booléens (type `bool`) sont des variables qui peuvent prendre les valeurs `true` et `false` avec les opérateurs

- `et` noté `&&`
- `ou` noté ||
- `négation` notée `not`

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

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

III.2.d 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
```

III.2.e 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.

III.2.f 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 = ()
```

III.3 Types composés

III.3.a 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
```

III.3.b 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
```

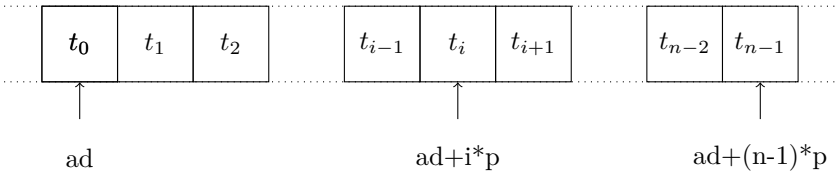
III.3.c Tableaux

Un tableau (type `array`) 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.
- On définit un tableau
  - en l'écrivant en extension

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

  - en créant un tableau avec une valeur initiale

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

- On accède aux éléments par leur indice

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

- On peut modifier un élément, c'est une instruction qui ne renvoie rien, de résultat `unit`.

```
# a.(1) <- 3;;
- : unit = ()
# a;;
- : int array = [|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 `:=`.
- La longueur d'un tableau est accessible par `Array.length`.
- 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.

III.3.d Listes

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

III.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 construire explicitement ce comportement :

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

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>
```

III.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.

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
```

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;;
```

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;;
```

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.

III.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`.

- 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;;
```

- 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; (* On peut aussi écrire incr i *)
   !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`.

IV Exercices

IV.1 Traductions

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

Exercice 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 telle que `f x` calcule $f(x)$.

Exercice 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.

- 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.
- Écrire un programme `hauteurCollatz a` qui détermine l'entier maximal atteint par la suite de Collatz de terme initial a.

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

Exercice 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.

Exercice 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.

Exercice 6 - Recherche par dichotomie
écrire une fonction qui détermine si une valeur appartient à un tableau **trié**.

Exercice 7 - Logarithme entier
Écrire une fonction `log_entier : int -> int` qui renvoie l'entier p tel que $2^p \leqslant n < 2^{p+1}$, sans utiliser la fonction logarithme.

Exercice 8 - Somme fixée
Écrire une fonction `somme : int -> int array -> bool * in * int` telle que `somme k t` renvoie `(true, a, b)` s'il existe deux termes a et b dans le tableau dont la somme est k et `(false, 0, 0)` sinon.
Donner une écriture plus rapide si le tableau est trié.

IV.2 Autres exercices

Exercice 9 - 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;;
```

Exercice 10 - 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
```

Exercice 11 - Composition
Écrire une fonction `comp f g` qui reçoit deux fonctions et qui retourne la composée $f \circ g$.
Quelle est sa signature ?

Exercice 12 - 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 ?

Exercice 13 - 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.

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

IV.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 tableau `perm` peut s'énoncer.

- On cherche le dernier indice j tel que `perm.(j) < perm.(j+1)`.
- S'il n'existe pas on est arrivé au dernier terme.
- 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)`.
- On échange les termes d'indices k et j dans `perm`.
- On retourne les termes de `perm` entre $j + 1$ et $n - 1$.

Exercice 14 - Permutation
Écrire la fonction définie ci-dessus.

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

Solutions

Solution de l'exercice 1 - Suite récurrente

```
let calculer_u n u0 f =
   let u = ref u0 in
   for i = 1 to n do
      u := (f !u) done;
   !u;;
```

Solution de l'exercice 2 - Suite de Collatz

```
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;;
```

Solution de l'exercice 3 - Somme

```
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;;
```

Solution de l'exercice 4 - Maximum

```
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;;
```

Solution de l'exercice 5 - Occurrences

```
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;;
```

Solution de l'exercice 6 - Recherche par dichotomie

```
let appartient x t =
   let a  = ref 0 in
   let b = Array.length t in
   while a < b - 1 do
      let c = (!a + !b)/2 in
      if t.(c) > x
      then b := c
      else a:= c done;
   t.(!a) = x;;
```

Solution de l'exercice 7 - Logarithme entier

```
let log_entier n =
   let p = ref 0 in
   let puiss2 = ref 1 in
   while !puiss2 <= n do
      incr p;
      puiss2 := 2 * !puiss2 done;
   !p - 1;;
```

Solution de l'exercice 8 - Somme fixée

```
let somme k t =
   let n = Array.length t in
   let a = ref 0 in
   let b = ref 0 in
   let trouve = ref false in
   for i = 0 to (n-1) do
      for j = i to (n-1) do
         if t.(i) + t.(j) = k
         then begin a := t.(i);
              b := t.(j);
              trouve := true end done done;
   !trouve, !a, !b;;
```

```
let somme k t =
   let n = Array.length t in
   let i = ref 0 in
   let j = ref 0 in
   let a = ref 0 in
   let b = ref 0 in
   let trouve = ref false in
   while !i < n && not !trouve do
      if t.(!i) + t.(!j) = k
      then begin a := t.(!i);
           b := t.(!j);
           trouve := true end;
      incr j;
      if !j = n then (j := 0; incr i) done;
   !trouve, !a, !b;;
```

```
let somme k t =
   let n = Array.length t in
   let i = ref 0 in
   let j = ref (n - 1) in
   let a = ref 0 in
   let b = ref 0 in
   let trouve = ref false in
   while !i <= !j && not !trouve do
      if t.(!i) + t.(!j) = k
      then begin a := t.(!i);
           b := t.(!j);
           trouve := true end;
      else if t.(!i) + t.(!j) < k
           then incr i
           else decr j done;
   !trouve, !a, !b;;
```

Solution de l'exercice 9 - Parenthésage

```
somme (somme (somme 2 3) 4) (somme 2 (somme 3 4));;
```

Solution de l'exercice 10 - Tableau unimodal

```
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;;
```

Solution de l'exercice 11 - Composition

```
let comp f g =
   let h x = f (g x) in
   h;;

#comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Solution de l'exercice 12 - Fonction mystère
La fonction calcule l'itérée n-ième de f.

```
#itere : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Solution de l'exercice 13 - Nombres de Hamming

```
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
```

Solution de l'exercice 14 - Permutation

```
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 begin 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 end;;
```