Algorithmes itératifs

1 Boucle inconditionnelle

Une boucle `for` est une succession d'instructions dépendant d'un compteur parcourant un ensemble ordonné fini. Généralement, le compteur parcourt les entiers de la valeur `debut` à la valeur `fin`.
En pseudo-langage : `Pour i de <debut> à <fin> ...`
En OCaml :
- Une seule instruction dans la boucle : `for i = debut to fin do instruction done`
Contrainte de typage : `debut` et `fin` doivent être de type `int` et `instruction` de type `unit`.
- Plusieurs instructions : `for i = debut to fin do <instr 1> ; <instr2> ; ... ; <instr n> done`

- Boucle parcourue à l'envers : `for i = fin downto debut do ...done`

Exemples.

Afficher les multiples de 3 de 1 à 30.

```
for n = 1 to 10 do print_int (3*n); print_string " " done ;;
```

Calcul du n-ième terme d'une suite récurrente $u_{n+1} = f(u_n)$

La fonction `suite` prend en argument une fonction f, une valeur initiale u_0 et un entier n et calcule u_n :

```
let suite f u0 n =
 let valeur = ref u0 in
 for k = 1 to n do valeur := f(!valeur) done;
 !valeur ;;
```

Signature de la fonction `suite : ('a -> 'a) -> 'a -> int -> 'a = <fun>`

La fonction `suite2` imprime à l'écran les valeurs (en flottants) de la suite de 0 à n :

```
let suite2 f u0 n =
 let valeur = ref u0 in
 for k = 1 to n do valeur := f(!valeur); print_float !valeur; print_string " " done ;;
```

Signature de la fonction `suite2 : (float -> float) -> float -> int -> unit = <fun>`

Test sur la fonction $x \mapsto 1 - x^2$.

Nombre d'occurrences d'un caractère donné dans une chaîne

```
let nb_occ c s =
 let nb = ref 0 in
 for i = 0 to String.length s - 1 do if s.[i] = c then incr nb done;
 !nb ;;
```

Signature de la fonction `nb_occ : char -> string -> int = <fun>`

Invariant de boucle

Un invariant de boucle est une propriété qui, si elle est vraie avant l'exécution d'un tour de boucle, reste vraie après.
La boucle étant parcourue un nombre fini de fois, si la propriété est vraie avant d'exécuter la boucle, grâce au principe de récurrence, elle sera vraie à la sortie de la boucle.
Exhiber un invariant de boucle permet de démontrer la correction d'un programme.

Exemple : pour la fonction `nb_occ`, l'invariant de boucle est la propriété P(`i`) : Après traitement de `i`, `nb` est égal au nombre d'occurrences de `c` dans la chaîne `s.[0]s.[1]...s.[i]`.

2 Boucle conditionnelle

Une boucle `tant que` est une succession d'instructions s'exécutant tant qu'une certaine condition est satisfaite.
En pseudo-langage : `Tant que <condition>, faire ...  fin tant que`
En OCaml : `while condition do instruction`
Contrainte de typage : `condition` doit être de type `bool` et `instruction` de type `unit`.
Bloc d'instructions : `while <condition> do <instr 1> ; <instr2> ; ...  <instr n> done`

Une boucle conditionnelle est intéressante lorsqu'on ne sait pas à l'avance combien de fois la boucle sera exécutée.
Une boucle `for` est un cas particulier de boucle `while` :
`for i = debut to fin do <instructions> done`
est équivalent à :

```
i:= debut;
while !i <= fin do <instructions>; i:= !i+1 done
```

Comme pour les boucles `for`, on peut définir la notion d'invariant de boucle.

Exemples.

1. Dichotomie

`dichotomie f a b eps` retourne un encadrement à ε près du zéro de la fonction f compris entre a et b. Le programme suppose que f admet un zéro unique entre a et b.

```
let dichotomie f a b eps =
  let u = ref a and v = ref b in
  while !v -. !u > eps do
    let m = (!u +. !v)/.2. in
  if f m = 0. then  (v := m; u := m)
  else if f m *. f(!u) < 0. then v := m else u := m
  done;
  !u , !v ;;
```

```
dichotomie :  (float -> float) -> float -> float -> float -> float * float = <fun>
```

La propriété "le zéro de f est compris entre u et v" est un invariant de boucle.

La propriété "Après le i-ème tour de boucle, $v-u = \frac{b-a}{2^i}$" est un invariant de boucle. Le programme termine car $\frac{b-a}{2^i}$ tend vers 0 quand i tend vers $+\infty$ donc devient $\leqslant \varepsilon$ au bout d'un nombre fini d'étapes, ce qui permet de sortir de la boucle.

2. PGCD de 2 entiers naturels

```
let pgcd a b =
  let x = ref (max a b) and y = ref (min a b) in
  while !y <> 0 do if !x > !y then x := !x - !y else y := !y - !x done;
  !x ;;
```

Si les valeurs contenues dans `x` et `y` sont égales, on renvoie cette valeur. Sinon, la valeur maximale contenue dans les références `x` et `y` est un entier naturel diminuant strictement à chaque passage dans la boucle donc les valeurs contenues dans `x` et `y` sont égales au bout d'un nombre fini d'étapes, d'où la terminaison du programme.

La propriété $\text{PGCD}(x, y) = \text{PGCD}(a, b)$ est un invariant de boucle.

Le PGCD de (x, y) vaut $\text{PGCD}(a, b)$ à l'entrée de la boucle et $\text{PGCD}(x, 0) = x$ à la sortie, donc le programme calcule bien le PGCD de a et b.

3 Tableaux

3.1 Structure de données composée

Un tableau est une structure de données composée d'éléments auxquels on accède directement par un numéro, appelé l'indice. Sa taille est fixée à sa création.

La taille maximale d'un tableau se calcule par `Sys.max_array_length`. Sur une machine 64 bits, elle est égale à 18014398509481983.
Il est représenté dans l'espace mémoire par une zone de cases contigues.
L'avantage est qu'on peut **accéder directement** à tout élément du tableau par son indice.
L'inconvénient est sa rigidité : on fige une zone mémoire allouée au tableau, on ne peut **pas modifier la taille d'un tableau, ni insérer ou supprimer un élément**.
On peut créer un tableau à plusieurs dimensions (tableau de tableaux). Un tableau à deux dimensions est une matrice.

3.2 Implémentation Caml

Un tableau de longueur n est indexé de 0 à $n-1$. Tous ses éléments doivent être du même type.
Le module `Array` de OCaml permet d'implémenter des tableaux et fournit des fonctions s'appliquant à des tableaux. Un tableau dont les éléments sont de type `'a` possèdera le type `'a array`.

- Création d'un tableau : `let tab = [|elt_1;elt_2;...;elt_n|]`
- Création d'un tableau de longueur n initialisé à une valeur donnée :
  `let tab = Array.make n value`
- Création du tableau `[|f(0);...;f(n-1)|]` où f est une fonction :
  `let tab = Array.init n f`
- Accès à l'élément d'indice `i` : `tab.(i)`
- Modification de la case d'indice `i` : `tab.(i) <- val`
- Taille du tableau : `Array.length tab`

Mutabilité

Un tableau est une structure **mutable** repérée par son adresse : on peut modifier le contenu de ses cases.
Important : si on crée un double du tableau par l'instruction `let tp = t` , le tableau copié `tp` aura la même adresse mémoire, donc toute modification de l'un entraîne la même modification de l'autre.

```
let t = [|1;2;3|] ;; let tp = t ;; t.(0) <- 10; tp ;; tp.(1) <- 20; t ;;
```

Pour effectuer une copie indépendante, il faut copier case par case ou utiliser la fonction `Array.copy`.

```
let t = [|1;2;3|] ;;
let t2 = Array.copy t ;;
t.(0) <- 10 ; t2 ;;
```

Egalité de deux tableaux

Il faut distinguer égalité structurelle et égalité physique :
`t = t2` teste si les contenus des deux tableaux sont identiques.
`t == t2` teste si les tableaux ont la même adresse mémoire (dans ce cas, il s'agit du même objet, avec des identificateurs différents).
La 2ème égalité entraîne la 1ère.

```
let t = [|1;2;3|] ;;
let tp = t ;;
let t2 = Array.copy t ;;
t = tp, t == tp, t = t2, t == t2 ;;
t.(0) <- 10 ;;
t2.(0) <- 10 ;;
t = tp, t = t2, t == t2 ;;
```

Tableau de tableaux

Un tableau de tableaux est appelé une matrice. Par exemple : `let m = [|[|1;2;0|];[|0;1;1|]|] ;;`
On accède à l'élément rangé dans la case `(i,j)` par `t.(i).(j)`.
Le nombre de lignes du tableau est `Array.length t`, le nombre de colonnes est `Array.length t.(0)`
On peut créer un tableau de tableaux par l'instruction `let m = Array.make 4 (Array.make 3 0) ;;`
L'inconvénient est que toute modification d'une case d'une ligne affectera les autres lignes :

```
m.(0).(0) <- 10 ;;
m ;;
```
La fonction `Array.copy` ne résout pas ce problème (même problème qu'en Python) :
```
let n = Array.copy m ;;
m.(0).(0) <- 20 ;;
n ;;
```
Pour le résoudre, on utilise la fonction `Array.make_matrix` :
```
let m = Array.make_matrix 4 3 0 ;; (* crée une matrice à 4 lignes et 3 colonnes initialisée à 0 *)
m.(0).(0) <- 30 ;;
m ;;
```

Des fonctions supplémentaires du module `Array` sont présentées dans le manuel de référence (chapitre 24.2)

Par exemple : `Array.sub t deb long` renvoie un tableau extrait de `t` commençant à l'indice `deb` et de longueur `long`.
`Array.append t1 t2` renvoie la concaténation des tableaux `t1` et `t2`.

3.3 Algorithmes utilisant des tableaux

3.3.1 Maximum dans un tableau

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

Invariant de boucle : après traitement de `i` dans la boucle, `m` contient le maximum de `t.(0),...t.(i)`

3.3.2 Renversement des valeurs d'un tableau

Deux approches sont possibles :

1. Modification du tableau en place

La fonction `renverse` modifie le tableau passé en paramètre et le renvoie.

```
let renverse t =
 let n = Array.length t in
 for i = 0 to n/2 - 1 do
  let aux = t.(i) in t.(i) <- t.(n-1-i); t.(n-1-i) <- aux
 done;
 t ;;
```

Que devient le tableau `t` après traitement de l'indice `i` dans la boucle ?

```
let t= [|1;2;3;4;5|] ;;
let t2 = renverse t ;;
t2 == t ;;
renverse t ;;
```

```
t : int array = [|1; 2; 3; 4; 5|]
#t2 : int array = [|5; 4; 3; 2; 1|]
#- : bool = true
#- : int array = [|1; 2; 3; 4; 5|]
```

2. Création d'un nouveau tableau sans changer l'ancien

```
let miroir t =
 let n = Array.length t in let tt = Array.make n t.(0) in
 for i = 0 to n-1 do tt.(i) <- t.(n-1-i) done;
 tt ;;
```

3.3.3 Suite récurrente simple ou double

On veut renvoyer les valeurs de 0 à n d'une suite récurrente $(u_n)_{n\in\mathbb{N}}$.
Dans le cas d'une suite récurrente simple, u_k ne dépend que de k et de u_{k-1}, donc pour remplir la case k, on a juste besoin du contenu de la case $k-1$.
Dans le cas d'une suite récurrente double, u_k dépend de k, u_{k-1} et u_{k-2}, donc pour remplir la case k, on a besoin du contenu des cases $k-1$ et $k-2$.
Ceci se généralise, même aux récurrences fortes dans lesquelles u_k dépend de toutes les valeurs précédentes.

Exemple de la suite de Fibonacci : $u_n = u_{n-1} + u_{n-2}$, u_0 et u_1 sont des paramètres de la fonction.
`fibo u0 u1 n` renvoie le tableau `[|`$u_0; u_1; \ldots; u_n$`|]`

```
let fibo u0 u1 n =
 let t = Array.make (n+1) 0 in
 t.(0) <- u0;
 t.(1) <- u1;
 for k = 2 to n do t.(k) <- t.(k-1) + t.(k-2) done;
 t ;;
```

```
fibo :  int -> int -> int -> int array = <fun>
```

Invariant de boucle : après traitement de k dans la boucle, `t` est le tableau `[|`$u_0; u_1; \ldots; u_k; 0; \ldots; 0$`|]`

3.3.4 Histogramme

On considère un tableau de valeurs entières comprises entre 0 et $n-1$. On veut compter le nombre des occurrences de chaque valeur, ce qui permet facilement de construire le tableau des fréquences.

```
let nb_occ tab n =
  let occ = Array.make n 0 in
  for i = 0 to Array.length tab -1 do
    occ.(tab.(i)) <- occ.(tab.(i)) + 1
  done;
  occ ;;
let frequences tab n =
  let occ = nb_occ tab n and sum = Array.length tab in
  let freq = Array.make n 0. in
  for i = 0 to n-1 do freq.(i) <- (float_of_int occ.(i)) /. (float_of_int sum) done;
  freq ;;
```