I Introduction

Définition A

Un algorithme **récursif** est un algorithme qui résout un problème d'une certaine taille en calculant des solutions de problèmes plus petits. Informatiquement, cela se manifeste par un appel de la fonction à elle-même, sur un nouvel objet.

Exemple B : Un classique de la récursivité

La fonction factorielle peut se définir mathématiquement par récurrence de la façon suivante :

- $0! = 1$.
- Pour $n \geq 1$, $n! = n \times (n-1)!$.

Informatiquement, cela s'écrit :

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

Remarque C : Cas d'arrêt

Dans une preuve ou une définition par récurrence, il est essentiel d'assurer une initialisation de la récurrence. En informatique, comme les appels sont *descendants*, on parle de **cas d'arrêt** de la fonction. Sans ces cas d'arrêts, ou s'ils sont mal implémentés, il y a un risque d'obtenir un algorithme qui ne s'arrête jamais de calculer. C'est le cas, par exemple, dans la fonction suivante :

```
let rec stupide = function
    | 0 -> 1
    | n -> n * stupide n;;
```

Cependant, il n'est pas toujours direct de détecter une erreur, comme par exemple dans le cas :

```
let rec subtil = function
    | 0 -> 1
    | n when n mod 3 = 0 -> subtil (n - 2)
    | n -> subtil (n + 1);;
```

Il existe actuellement de tels algorithmes dont la terminaison est une conjecture. C'est le cas de la conjecture de Syracuse, qui affirme que la fonction ci-dessous arrête ses calculs pour tout entier :

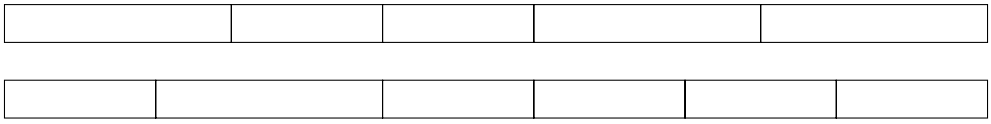
```
let rec syracuse = function
    | 0 -> 1
    | 1 -> 1
    | n when n mod 2 = 0 -> syracuse (n / 2)
    | n -> syracuse (3 * n + 1);;
```

Exercice 1

Écrire une fonction `pgcd` qui calcule le pgcd de deux entiers naturels en suivant l'algorithme d'Euclide.

Exercice 2

On dispose de briques de longueur 2 et 3, et on veut connaître le nombre de façons de construire une rangée de longueur n. Par exemple, voici deux façons de construire une rangée de longueur 13. Il en existe 16 au total :



Écrire une fonction `briques` permettant de calculer ce nombre de façons pour une longueur totale donnée. (On pourra étudier le nombre de possibilités de construire des rangées de longueur $n-2$ et $n-3$).

II Analyse d'algorithmes

II.1 Terminaison

L'étude de la terminaison d'un programme consiste à essayer de prouver que tout calcul s'arrêtera au bout d'un temps fini.

Définition D

Soit E un ensemble muni d'une relation d'ordre strict $\prec$.
Un élément x de E est dit **minimal** si et seulement si, pour tout $y \in E$, $y \not\prec x$.
L'ensemble E est dit **bien fondé** si toute partie non vide de E admet un élément minimal.

Exemple E

- L'ensemble $(\mathbb{N}, <)$ est un ensemble bien fondé. Il n'a qu'un seul élément minimal : 0.
- L'ensemble $\mathbb{N}^2$ muni de l'ordre lexicographique est bien fondé, où l'ordre lexicographique est défini par : $(a,b) \prec (c,d) \Leftrightarrow a < c$ ou $(a = c$ et $b < d)$.
  Il n'a qu'un seul élément minimal : $(0,0)$.
- L'ensemble $(\mathbb{Z}, <)$ n'est pas bien fondé.
- L'ensemble $\mathbb{N} \setminus \{0,1\}$ muni de la relation de divisibilité est un ensemble bien fondé. Ses éléments minimaux sont les nombres premiers (il en existe donc une infinité).

Théorème F : Induction structurelle

Soit $(E, \prec)$ un ensemble bien fondé et M l'ensemble de ses éléments minimaux. Soit $\mathcal{P}$ un prédicat sur E (une fonction de E dans {Vrai,Faux}). On suppose :

- Pour $m \in M$, $\mathcal{P}(m)$ est vrai.
- Pour $x \in E \setminus M$, il existe $y_1, \dots, y_n \in E$ tels que $y_i \prec x$ et si tous les $\mathcal{P}(y_i)$ sont vrais, alors $\mathcal{P}(x)$ est vrai.

Alors, pour tout $x \in E$, on a $\mathcal{P}(x)$.

Preuve

Démontrons ce résultat par l'absurde. Soit X l'ensemble des éléments de E tels que $\mathcal{P}(x)$ est faux. Si on suppose X non vide, alors X admet un élément minimal x_0 (car E est bien fondé). Par la première hypothèse, nécessairement, $x_0 \notin M$. Par la deuxième hypothèse, il existe des y_i vérifiant les conditions. Cependant, comme $\mathcal{P}(x)$ est faux, nécessairement l'un des $\mathcal{P}(y_i)$ est faux, ce qui contredit la minimalité de x_0.

Remarque G

Ce théorème est la base du raisonnement par récurrence, lorsque $(E, \prec) = (\mathbb{N}, <)$. Il permet de montrer le théorème de terminaison suivant :

Corolaire H

Soit f une fonction d'ensemble de définition E, un ensemble bien fondé dont les éléments minimaux sont dans M. On suppose :

- Pour $m \in M$, $f(m)$ termine.
- Pour $x \in E \setminus M$, $f(x)$ fait un nombre fini d'appels à f, en les valeurs y_1, y_2, …, y_n, et de plus, pour tout $i \in [\![1; n]\!]$, $y_i \prec x$.

Alors $f(x)$ termine pour tout $x \in E$.

Exercice 3

Montrer que l'algorithme d'Euclide termine.

Exercice 4

On définit la fonction d'Ackermann de la façon suivante :

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

Montrer que la fonction d'Ackermann termine.
Attention : `ackermann 4 2` *dépasse déjà le nombre d'atomes dans l'univers.*

Exercice 5

Montrer que la fonction de MacCarthy définie ci-dessous termine.

```
let rec f n =
   if n > 100 then n - 10
   else f(f(n + 11));;
```

II.2 Correction

L'étude de la correction d'un algorithme consiste à prouver qu'il calcule bien ce qu'on attend de lui. Le théorème d'induction structurelle fournit directement ce qui est nécessaire, en utilisant le prédicat « La fonction f calcule la bonne valeur pour x ».

Exercice 6

Prouver la correction de la fonction suivante, qui supprime les n premiers éléments d'une liste ℓ.

```
let rec supprime n l = match n, l with
   | 0, _ -> l
   | _, [] -> []
   | _, _ :: q -> supprime (n - 1) q;;
```

II.3 Complexité

II.3.1 Introduction

L'étude de la complexité (ou coût) d'un algorithme consiste à étudier d'une part le temps qu'il va mettre à être exécuté (complexité temporelle) et d'autre part l'espace mémoire qu'il va occuper au cours de son exécution (complexité spatiale). Ces valeurs dépendent :

- De la **taille** de l'argument, qui dépend de la structure : le nombre de chiffre d'un entier, le nombre d'éléments dans un tableau, le nombre de nœuds dans un arbre, ... En pratique, on considèrera l'espace mémoire qu'il occupe.
- De la manière dont est implémenté l'algorithme.

Un calcul de complexité consiste à découper un algorithme en une suite d'instructions élémentaires supposées à coût constant, notamment :

- Une opération arithmétique (addition, multiplication, ...)
- Une comparaison de valeurs (test d'égalité, de supériorité, ...)
- Une écriture de la mémoire (affectation, modification, ...)

Plutôt que de déterminer un nombre exact de ces opérations, on préfère en trouver un ordre de grandeur, et on utilise pour cela des notations mathématiques :

Définition I : Notations de Landau

Soient f et g deux fonctions $\mathbb{N} \longrightarrow \mathbb{R}$. On note :

- $f(n) = O(g(n))$ s'il existe $A \in \mathbb{R}^+$ tel que $f(n) \leq Ag(n)$.
- $f(n) = \Omega(g(n))$ si $g(n) = O(f(n))$.
- $f(n) = \theta(g(n))$ si $f(n) = O(g(n)) = \Omega(g(n))$.

Remarque J

On pourrait être tenté de penser « Pourquoi améliorer mon algorithme ? Les ordinateurs seront bientôt tellement puissants qu'il s'exécutera instantanément ! » C'est faux ! Voici un tableau récapitulatif des temps d'exécution en fonction de la complexité des fonctions et de la taille des entrées. On supposera faire les calculs sur un ordinateur d'1 GHz (10^9 opérations par seconde).

Nom	Logarithmique	Linéaire	Semi-linéaire	Quadratique	Cubique	Exponentiel
Taille	$\log n$	n	$n \log n$	n^2	n^3	2^n
10	3 ns	10 ns	30 ns	100 ns	1 µs	1 µs
100	7 ns	100 ns	700 ns	10 µs	1 ms	4 000 milliards d'années
10 000	13 ns	10 µs	133 µs	100 ms	17 s	-
1 000 000	20 ns	1 ms	20 ms	17 min	32 ans	-

Remarque K : Type de complexité

Un algorithme peut ne pas mettre le même temps d'exécution sur deux objets de même taille. C'est par exemple le cas lorsqu'on cherche une valeur dans un tableau :

- si la valeur est la première case du tableau étudiée,
- si la valeur n'est pas présente dans le tableau.

On distinguera alors trois types de complexité, dont les noms sont explicites :

- Complexité dans le pire des cas.
- Complexité dans le meilleur des cas.
- Complexité en moyenne.

Sans précision supplémentaire, la complexité par défaut sera celle dans le pire des cas.

II.3.2 En récursif

Comme les autres éléments d'analyse, on utilise une propriété de récurrence pour déterminer un ordre de grandeur de la complexité, qu'elle soit spatiale ou temporelle.

Exemple L

Analysons la complexité temporelle de la fonction `fact`. En notant $C(n)$ le temps d'exécution pour calculer $n!$, on a :

- $C(0) = O(1) \leq k_0$ où k_0 est une constante.
- Si $n > 0$, $C(n) = C(n-1) + O(1) \leq C(n-1) + k_1$ où k_1 est une constante.

On en déduit que $C(n) \leq k_0 + n \times k_1$, soit $C(n) = O(n)$.

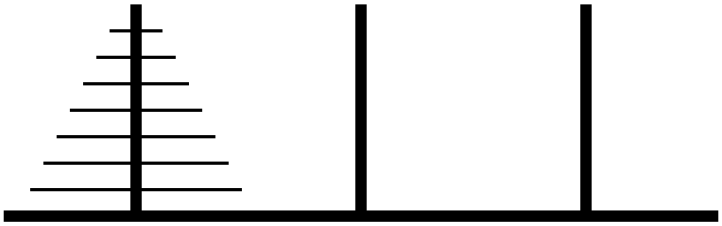
Exercice 7

Déterminer la complexité temporelle de `briques` de l'exercice 2.

Exercice 8

Les tours de Hanoï est un jeu formé de 3 colonnes et n disques de diamètres croissants. Au début du jeu, tous les disques sont empilés par taille croissante sur la première colonne et l'objectif est de les déplacer sur la troisième colonne, sachant que :

- On ne peut déplacer qu'un disque à la fois.
- Un disque doit toujours être placé sur une colonne vide, ou sur un disque de diamètre plus grand.



1. Écrire une fonction `hanoi : int -> unit` affichant une solution du jeu, en écrivant tous les mouvements nécessaires de la forme `a->b` où `a` est la tour d'origine et `b` la tour d'arrivée, en utilisant la fonction suivante :

```
let prt_tp (a, b) =
   print_int a; print_string "->";
   print_int b; print_string "\n";;
```

   Par exemple, on a l'exécution suivante :

```
# hanoi 3;;
0->2
0->1
2->1
0->2
1->0
1->2
0->2
- : unit = ()
```

2. Déterminer le nombre exact de déplacements lors de l'exécution de la fonction `hanoi`.
3. On ajoute une règle supplémentaire : un disque ne peut être déplacé que d'une colonne à la colonne suivante dans l'ordre modulo 3 (0 vers 1, 1 vers 2 et 2 vers 3). Modifier la fonction `hanoi` et déterminer le nombre exact de déplacements dans ce cas.
   Indication : la complexité temporelle est en $O((1+\sqrt{3})^n)$

III Utilisation de la mémoire

III.1 Pile d'appels

On se propose d'étudier l'exécution de la fonction `fact` (rappelée ci-dessous) sur l'entier 4.

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

Lors de l'appel à `fact 4`, la condition d'arrêt n'est pas vérifiée. Le calcul doit donc être mis en attente, le temps de calculer `fact 3`, puis de même pour calculer `fact 2`, ...
Lorsque finalement `fact 0` est calculé et prend la valeur 1, on peut alors calculer `fact 1`, puis `fact 2` et ainsi de suite jusqu'à pouvoir calculer `fact 4`.
Toutes ces informations doivent être mise en mémoire pour permettre le calcul du résultat final. Elles sont stockées dans ce qui est appelé la **pile d'appels**.

Il existe en `caml` une fonction permettant de suivre ces appels :

```
# #trace fact;;
fact is now traced.

# fact 4;;
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact <-- 0
fact --> 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
- : int = 24
```

On constate ci-contre qu'un appel à la fonction est placé dans la pile d'appels avec son argument, avec la flèche `<--`. Lorsque une valeur est finalement calculée, le résultat est sorti de la pile d'appels avec la flèche `-->`.

Ainsi, même si ce n'est pas visible directement dans la fonction, l'utilisation d'une fonction récursive nécessite de la mémoire supplémentaire.
La complexité spatiale de la fonction `fact` suit les mêmes relations de récurrence que sa complexité temporelle. On en déduit que cette fonction occupe un espace $O(n)$.

III.2 Récursivité terminale

Comme vu précédemment, la taille de la pile d'appels augmente à chaque appel récursif. Cependant, cette pile a une taille limitée, pour éviter d'occuper trop d'espace mémoire. Il est possible d'écrire une petite fonction pour effectuer ce test :

```
let n = ref 0;;

let rec limite () =
   incr n;
   1 + limite ();;
```

Cette fonction ne va évidemment pas s'arrêter, mais `caml` va forcer le programme à stopper. On étudie alors la valeur de la référence :

```
# limite ();;
Stack overflow during evaluation (looping recursion?).
# !n;;
- : int = 262068
```

Ainsi, toute fonction qui nécessite d'avoir plus de 262 068 appels en attente simultanément ne pourra pas terminer.

Étudions un autre exemple, une façon naïve d'écrire la fonction identité sur $\mathbb{N}$:

```
let rec identite1 = function
   | 0 -> 0
   | n -> 1 + identite (n - 1);;
```

En accord avec ce qui a été dit précédemment, on a :

```
# identite1 262067;;
- : int = 262067
# identite1 262068;;
Stack overflow during evaluation (looping recursion?).
```

Modifions alors la fonction pour ne pas effectuer d'opérations en plus de l'appel récursif, en utilisant un accumulateur :

```
let identite2 n =
   let rec aux = function
      | (acc, 0) -> acc
      | (acc, n) -> aux (acc + 1, n - 1) in
   aux (0, n);;

# identite2 262068;;
- : int = 262068
# identite2 1000000;;
- : int = 1000000
```

Pourquoi cette nouvelle version ne présente plus la même limite, alors qu'a priori elle nécessite le même nombre d'appels récursifs ? La réponse est dans la définition suivante :

Définition M

Soit f une fonction d'ensemble de définition E, un ensemble bien fondé dont les éléments minimaux sont dans M. f est dite **récursive terminale** si pour $x \notin M$, le dernier calcul effectué par $f(x)$ est un appel récursif.

Cette définition a du sens, car si un seul appel récursif est effectué pour calculer $f(x)$, c'est qu'il existe $y \in E$, $y \prec x$ tel que $f(x) = f(y)$. Par conséquent, inutile de garder en mémoire tous les appels précédents, il suffit de continuer jusqu'à trouver un élément de M. La fonction de traçage permet d'illustrer ce phénomène :

```
# #trace identite1;;
identite1 is now traced.
# identite1 3;;
identite1 <-- 3
identite1 <-- 2
identite1 <-- 1
identite1 <-- 0
identite1 --> 0
identite1 --> 1
identite1 --> 2
identite1 --> 3
- : int = 3
```

```
# #trace aux;;
aux is now traced.
# aux (0,3);;
aux <-- (0, 3)
aux <-- (1, 2)
aux <-- (2, 1)
aux <-- (3, 0)
aux --> 3
aux --> 3
aux --> 3
aux --> 3
- : int = 3
```

On constate en effet que la valeur retournée est toujours la même une fois que les retours commencent.

Remarque N

Le compilateur `caml` est optimisé pour gérer la récursivité terminale. Ce n'est pas le cas en `python`, qui limitera de toute façon le nombre d'appels récursifs. Ainsi, il n'est pas nécessaire d'écrire des fonctions récursives terminales en `python`.
Il n'est jamais exigé aux concours d'écrire des fonctions récursives terminales, notamment car leur écriture alourdit souvent la compréhension de l'algorithme.

Exercice 9

Écrire la fonction `fact` de manière récursive terminale.

III.3 Mémoïsation

La fonction ci-dessous calcule les termes de la classique suite de Fibonacci définie par :

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_{n+2} = f_{n+1} + f_n \text{ si } n \geq 0 \end{cases}$$

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

On constate que la relation de récurrence suivi par la complexité temporelle est : $C(0) = O(1)$, $C(1) = O(1)$ et $C(n+2) = C(n+1) + C(n)$ pour $n \geq 0$. On en déduit facilement que $C(n) \geq f_n = \Theta(\varphi^n)$.

Le traçage de la fonction permet de mieux comprendre ce qui se passe :

```
# #trace fibo;;
fibo is now traced.
# fibo 5;;
fibo <-- 5
fibo <-- 3
fibo <-- 1
fibo --> 1
fibo <-- 2
fibo <-- 0
```

```
fibo --> 0
fibo <-- 1
fibo --> 1
fibo --> 1
fibo --> 2
fibo <-- 4
fibo <-- 2
fibo <-- 0
fibo --> 0
```

```
fibo <-- 1
fibo --> 1
fibo --> 1
fibo <-- 3
fibo <-- 1
fibo --> 1
fibo <-- 2
fibo <-- 0
```

```
fibo --> 0
fibo <-- 1
fibo --> 1
fibo --> 1
fibo --> 2
fibo --> 3
fibo --> 5
- : int = 5
```

On constate ici que f_3 est calculé 2 fois, f_2 est calculé 3 fois, f_1 est appelé 5 fois et f_0 est appelé 3 fois. Le programme n'a aucune notion de « mémoire » de ce qu'il a déjà calculé.

Améliorons ça en calculant les termes deux par deux :

```
let fibo2 n =
   let rec aux (a, b) = function
      | 0 -> a
      | n -> aux (b, a + b) (n - 1) in
   aux (0, 1) n;;
```

Dans cette version (qui est récursive terminale), on évite de relancer des appels en calculant les termes précédents. En effet, une suite récurrente d'ordre 2 s'itère en connaissant deux termes consécutifs.

On perd cependant l'esprit de l'écriture intuitive de la suite de Fibonacci. Proposons alors une dernière version utilisant de la mémoire pour conserver les valeurs déjà calculées.

```
let t = Array.make 100 (-1);;
t.(0) <- 0; t.(1) <- 1;;

let rec fibo_mem n =
   if t.(n) = -1 then
      t.(n) <- fibo_mem (n - 1) + fibo_mem (n - 2);
   t.(n);;
```

Cette version est dite **mémoïsée**. On stocke dans la mémoire les calculs déjà effectués pour éviter de les refaire. La fonction obtenue a alors une complexité linéaire.

Nous étudierons au prochain chapitre la structure de dictionnaire qui permet de stocker uniquement les valeurs déjà calculées, avec d'autres arguments que des entiers, sans avoir à déterminer la taille de la structure au préalable.

Exercice 10

1. Écrire une version naïve de fonction calculant les coefficients binomiaux.
2. Déterminer sa complexité temporelle.
3. Améliorer la fonction précédente en écrivant une version mémoïsée.
4. Quelle est sa complexité ?