Mes notes d'un cours d'OCaml (option informatique)

Benjamin LOISON (MPSI 1)
Fénelon
2018-2019
Professeur: Mme Faget

1 Introduction à la programmation en Caml

1.1 Introduction

On peut utiliser l'IDLE WinCaml (disponible sur `http://jean.mouric.pagesperso-orange.fr/`).
Cette année on programmera en OCaml et non CamlLight
On a l'autorisation d'ammener notre ordinateur à toute séance d'option informatique.

1.1.1 Spécificité Caml

1.1.1.1 Paradigme

Caml est un langage essentiellement fonctionnel (qui permet aussi de faire de la programmation impérative).
La programmation impérative indique au processeur les étapes à suivre.
La programmation fonctionnelle met en avant la définition et l'évaluation de fonctions et limite les affectations.
Les fonctions sont considérées comme des valeurs, elles peuvent être le paramètre ou le résultat d'autres fonctions.

Exemple: Calcul de factoriel ≥ 1

Style impératif:

```
1 fact(n):
2     res = 1 # résultat
3     pour i = 1 ... n
4         res *= i
5     return res
```

Style fonctionnel:

```
1 fact(n):
2     fact(1) = 1
3     fact(n) = n * fact(n - 1)
```

1.1.1.2 Typage

Caml est fortement type car tous les objets manupulés appartiennent à un type donné qui ne change pas. On dit que les conversions de type sont interdites.
Au contraire, Python est faiblement typé.

Exemple: $i = (i + j)$ / 2 # i peut devenir un float

En Caml c'est impossible, l'entier 1 s'écrit "1" et le réel 1 s'écrit "1.".
Le typage d'une fonction en Caml est réalisé lors de sa définition.
Caml détermine le type des arguments et du résultat.

Exemple:

Addition des entiers +
Addition des réels +.

1 + 2. génère donc une erreur, il faut écrire: (float_of_int 1) +. 2. (la conversion de type est de préférence à éviter).
1.5 + 2.5 génère aussi une erreur, il faut écrire 1.5 +. 2.5 sin 2.
3 / 5 renvoie 0 (division entière, 6 / 5 = 1)
3 /. 5 génère une erreur (3. / 5. = 2.5)

1.1.2 Variables globales

Règles pour le nom des variables:
- commence par une lettre
- peut utiliser des chiffres
- pas d'espace
- tiret du bas uniquement

1.1.2.1 Variables globales

On attribue un nom à une variable avec l'instruction let
let pi = 3.14159;;
pi *. 2.;;
Une fois crée la valeur ne change plus sauf en la modifiant directement.
let a = 1;; let b = a + 1;;
let a = 2;;

1.1.2.2 Variables locales

let a = 1 in 2 * a;;
Si on éxécute ce code cela génère une erreur (a est inconnue). (Ben: non...)

let a = 1;;
let a = 2 in 2 * a;;
a;; (* vaut 1 *)

1.1.2.3 Définitions multiples

let a = 1 and b = 2 in a * b;;
Attention on ne peut pas faire de définitions multiples liées.
On ne peut pas faire: let a = 2 and b = 2 * a in a * b;; (* erreur a est inconnue*)
let a = 2 in let b = 2 * a in a * b;;

1.1.3 Types de base

1.1.3.1 Types simples

1. int (entiers compris entre $-2^{30}+1$ et $2^{30}-1$)
   Opérateurs + * / -
   Modulo et valeur absolue: mod abs
   abs (−4);; (* utiliser des parenthèses lorsqu'on met un '−' *)
   5 mod 2;;

2. float (nombres décimaux approchés)
   Opérateurs exp log (* ln *) sqrt sin cos tan +. −. /. *. tanh
   (int_of_float ... éviter - reprendre en amont le problème si doit le faire en général)

3. Bool (valeurs booléennes true false)
   let a = 1 = 2;; (* a: bool = false *) seul un égal pour la comparaison
   Opérateurs && || not (pas and, or fonctionne mais pas and donc éviter or)
   Evaluation paresseuse: le second test n'est fait que si nécessaire
   let a = false && (1 / 0 = 1);; (* ne génère pas d'erreur (ne fait pas la division par 0 *)

4. char (caractère let a = 'z';;)
   char_of_int int_of_char (conversion code ascii)

5. string (chaîne de caractères)
   let s = "mpsi";;
   s.[0];; (* char 'm' *)
   s.[0] <- 'M';; (* function de type unit *)
   s;; (* string "Mpsi" *)
6. unit () (on note comme ça)

1.1.3.2 Couples, triplets, p-uplet

On crée les p-uplets en séparant les objets avec des virgules.
let a = (true, 1, 2.35, "bonjour");;
a: bool * int * float * string

1.1.4 Fonctions

1.1.4.1 Fonctions d'une seule varible

let f = function x -> x * x;;
let f = fun x -> x * x;; (* parfois *)
let f x = x * x;; (* préférer ceci, si *. float -> float *)
f est de type int (entrée) -> int (sortie)
let f x = x * sin x;; (* erreur, patch: *. *)
Les parenthèses sont parfois indispensables.
let f x = x + 1;; (* f: int -> int *)
f 2 * 3;; compris comme (f 2) * 3 (priorité aux fonctions) mieux vaut un peu surparenthèser. (-> 9)

Autrement dit f x y est interprété en (f x) y

1.1.4.2 Fonctions

Deux solutions pour définir une fonction à plusieurs variables. Une première possibilité consiste à définir une fonction prenant en argument un p-uplet.

Exemple:

On veut écrire une fonction "divise" qui dit si m divise n.

let divise (m, n) = n mod m = 0;;
divise: (int * int) -> bool

Le type est déterminé automatiquement quand c'est possible. Parfois le type n'est pas connu à l'avance, on parle de fonctions polymorphes.

let fst (a, b) = a;; (* snd (second) f: ('a * 'b) -> 'a *)

On pouvait changer le b par '_' si on ne s'en sert pas.

Le type des arguments peut aussi être une fonction.

Exemple:

Fonction "acc" qui calcule le taux d'accroissement d'une fonction entre a et b ?

```
1 let acc(f, a, b) =
2     (f(b) -. f(a)) /. (b -. a);; (* float -> float, float, float) -> float *)
```

Si on a:
let acc2(f, a, b) = f(b) -. f(a);; (* ('a -> float * 'a * 'a) -> float, on lit "alpha" pour 'a *)
De même on peut renvoyer une fonction:
let double f = function x -> 2 * (f x);; (* ('a -> int) -> 'a -> int *)
argument résultat
let g = double somme;;

1.1.4.3 Curryfication et décurryfication

Exemple:

$(x, y) \in \mathbb{Z}^2$ et $x + y \in \mathbb{Z}$
On veut définir f: x, y -> $x + y$
~~let somme(x, y) = x+ y;; (* int * int) -> int (moyen) *)~~
let somme x y = x + y;; (* int -> int -> int *)
La première version est une fonction a une seule variable qui est un couple.
La seconde version est la version curryfiée et c'est celle qu'on préfère car elle permet de définir des applications partielles.
D'un point de vue mathématique, on utilise (l'existence d'une bijection entre les deux ensembles):
$(\mathbb{Z} * \mathbb{Z} ->\ \mathbb{Z})$ et $(\mathbb{Z} ->\ F(\mathbb{Z}, \mathbb{Z}))$
$((x, y) ->\ x + y)$ et $(x ->\ \mathbb{Z} ->\ \mathbb{Z})$

let somme2 = somme 2;;(*int -> * int, n'y a t'il pas une faute ? *)

Syntaxe équivalente:

```
1    let mul = function m -> function n -> m * n;; (* nul *)
2    let mul m = function n -> m * n;; (* nul *)
3    let mul m n = m * n;;
4    (* int -> int -> int
5    mul(3, 2) génère une erreur *)
```

Attention à l'ordre des arguments.
let ascii c n = int_of_char c = n;; (* char -> int -> bool ('=' compare des choses de même type) *)
ascii 'a' 97 (* true ? *)
let g = ascii 97;; (* erreur *)
let g = function c -> ascii c 97;;
(* préférer: *) let g c = ascii c 97;;

1.2 Programmation impérative

On donne une suite d'instructions (affectation en mémoire, tests) qui après l'éxécution donne le résultat attendu.
Le contenu d'une fonction impérative est une suite d'instruction de type unit à l'exception de la dernière qui est du type du résultat.

1.2.1 Ref, vec, mat

- Si une variable est destinée à être modifié au cours de l'éxécution, on crée une ref.
Une ref est un emplacement mémoire de taille fixe pour le stockage d'une valeur d'un type donné.

let a = ref 1;; (* création d'un ref, type int ref *)
!a;; (* déférencement (accès à la valeur), type int, lu "bang a" *)
a := 0;; (* modification, type unit *)

- Un vec est un ensemble de places mémoires de taille fixée au moment de la création.
Les places mémoires contiennent des objets du même type.
On ne modifie ni la taille, ni le type d'un vec.
let a = [|1; 2; 3|];; (* int array(3), [|1, 2, 3|] type: int * int *int array(1) *)
a.(0);; (* 1 *)
a.(0) <- 0;; (* string: chaîne.[0] <-, type unit*)

let a = Array.make 5 0;;(* taille, valeur par défaut *)
Taille du vecteur: Array.length
Copie de vecteur: Array.copy, exemple: let vec1 = Array.copy vec0

- Les matrices sont des tableaux de tableaux.
let m = Array.make_matrix 3 3 0;;(*n: nombre de lignes, p: nombre de colonnes, valeur, type de m: int array array *)
m.(0);; (* 1ère ligne *)
m.(0).(0) <- 1;; (* type unit *)

1.2.2 Boucles

Syntaxe boucle for:

```
for i = ... to ... do
    ...; (* instruction *)
    ...; (* de type unit *)
done;

while (test) do
    ...; (* unit *)
    ...; (* unit *)
done;
```

Exemple:

Afficher les valeurs d'un tableau (vec) d'entiers:

```
let affiche t = (* int array -> unit *)
    let n = Array.length t in
        for i = 0 to n - 1 do
            print_int t.(i);
            print_newline();
        done;;
```

En Caml les indentations sont facultatives, s'il n'y a qu'une instruction on peut ne mettre qu'un ou aucun ';'.

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

for i = n downto 1 do (* ou n - 1 downto 0 *)
    ...

let fact n =
    let res = ref 1 and i = ref 1 in
        while !i <= n do
            res := !res * !i
            i := !i + 1 (* incr i, argument: int ref *)
        done; !res;;
```

1.2.3 Branchements

```
if (cdt) then
    (...;
     ...;)
else
    (...;
     ...;)
```

Les parenthèses et ';' sont facultatifs s'il n'y a qu'une seule instruction avec le then ou le else.
La dernière ligne de chaque bloc doit être de même type et toutes les autres de type unit.
S'il n'y a pas de else, la dernière ligne du bloc est de type unit (on sous-entend else ()).

```
let signe x = (* int -> string *)
    if x > 0 then "pos"
    else "neg";;
```

```
4
5  let signe2 x =
6      if x > 0 then "pos";; (* génère une erreur *)
7
8  let signe2 x =
9      if x > 0 then print_str "pos";; (* est la bonne version *)
```

1.2.4 Les filtrages

Syntaxe de branchelent qui consiste à reconnaître la forme d'une variable.

```
1  (* filtrage explicite *)
2  let t x = match x with
3  | motif 1 -> expr 1
4  ...
5  | motif n -> expr n;;
6
7  (* filtrage implicite *)
8  let f = function
9  | motif 1 -> expr 1
10 ...
11 | motif n -> expr n;;
```

Lors du calcul d'une telle fonction, l'interprète va essayer de faire concorder l'argument avec les motifs successifs.

Exemple:

Définition de la fonction sinc (sinus cardinale):

```
1  let sinc x = match x with
2  | 0. -> 1. (* le premier '|' n'est pas obligatoire *)
3  | _ -> (sin x) /. x;; (* si on inverse les deux lignes, on a un "warning" car le second cas *)
4  (* n'est pas pris en compte, il y a un warning si le filtrage n'est pas exhaustif *)
5  | x -> (* on peut utiliser une des deux lignes suivantes au lieu de la précédente *)
6  | y -> (* renomme la variable en y ? *)
7
8  let f m n = match (m, n) with
9  | (_, 0) -> m
10 | (0, _) -> n
11 | (_, _) -> m * m + n * n;;
```

Attention à ne pas confondre concordances des motifs et égalité des noms.

```
1  let est_egal x y = match y with
2  | x -> true
3  | y -> false;;
```

Le 1^{er} test n'est pas interprété comme $x = y$ mais comme le cas général où l'on a renommé y en x.

Motif gardés:

Les filtres peuvent êtres combinés à des tests avec la syntaxe:

```
1  (* | motif when cdt -> *)
2
3  let est_egal x y = match y with
4  | y when y = x -> true (* on peut mettre des expressions complexes au lieu de "true" *)
5  | _ -> false;;
```

1.3 Programmation récursive

1.3.1 Principe

- Donner le résultat dans le cas d'arrêt - Donner la formule permettant le calcul dans les autres cas (formule de récurrence) - Laisse la machine faire

```
let rec fact n = match n with
| n when n < 0 -> failwith "entrée négative"
| 0 -> 1
| _ -> n * fact (n - 1);; (* les parenthèses sont ici primordiales *)

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

1.4 Listes

Un vec est une structure de taille fixe dans laquelle on peut accéder à chacun des éléments en temps constant.
Une list est une structure de taille variable mais pour accéder à un élement autre que le 1^{er}, on doit parcourir la liste.
let a = [1; 2; 3];; (* int list *)
Pour mettre un élément en tête de liste, on utiliser ::
let b = 2::a
b: int list , b = [2; 1; 2; 3]
On peut faire des listes de n'importe quel type mais les éléments d'une liste ont tous le même type.
On note la liste vide []

Fonctions sur les listes:

List.hd renvoie le 1^{er} élément (ne modifie pas la liste)
List.tl renvoie tous les éléments sauf le 1^{er}
@ permet la concaténation de 2 listes.
List.rev renvoie la liste renversée.
List.mem prend x et l et renvoie true si $x \in l$ false sinon (et est de coût linéaire).

On utilise beaucoup les listes dans les programmes récursifs, on les manipule avec des filtrages.

Exemple:

```
let tete l = match l with (* <=> List.hd *)
| [] -> failwith "Liste vide"
| t::q -> t;;

let queue l = match l with (* <=> List.tl *)
| [] -> failwith "Liste vide"
| t::q -> q;;

let longueur l = match l with (* <=> List.length, les deux fonctions sont à éviter *)
| [] -> 0 (* car la complexité est de l'ordre de n (la taille de la liste) *)
| t::q -> 1 + longueur q;;

let rec appartient x l = match l with (* pire cas: n *)
| [] -> false
| t::q when t = x -> true
| t::q -> appartient x q;;

let rec appartient x l = match l with
| [] -> false
| t::q -> t = x || appartient x q;;

let rec dernier l = match l with
| [] -> failwith "Liste vide"
```

```
| [t] -> t
| t::q -> dernier q;;
```

1.5 Compléments sur les types:

1.5.1 Types construits

On peut construire de nouveaux types à partir de types existants.
On distingue les types produits et les types sommes.

- Les types produits:
Un type produit est un ensemble de champs. Deux types de produits distincts ne peuvent pas avoir de nom de champs en commun.

type newType = champ 1: type 1;...; champ n: type n

Exemple on définit un point de $\mathbb{R}^2$:

```
type point = {x: float; y: float};;
let p1 = {x = 2.; y = 3.};;
```

Accéder à un champ: p1.x
Les valeurs des champs ne sont pas modifiables à moins de les déclarer comme telles avec le mot mutable.

```
type point = {mutable x: float; mutable y: float};;
let p1 = {x = 2.; y = 3.};;
(* modifier un champ: *)
p1.x <- 1.5;; (* par exemple *)

let milieu a b = {x = (a.x +. b.x) /. 2.; y = a.y +. b.y /. 2.};;
```

Fonction de type (point -> point -> point)

On peut définir des types polymorphes exemple:

```
type('a, 'b) couple = {x: 'a; y: 'b};;
let test = {x = "cold"; y = 75};;
(* test: (string * int) *)

type point = {mutable x: float; mutable y: float};;
type vect = {dx: float; dy: float};;

let translation p v =
    p.x <- p.x +. v.dx;
    p.y <- p.y +. v.dy;;
```

- Types sommes :

```
type reeletendu = Reel of float | Plusinfini | Moinsinfini;;

let pi = Reel 3.14;;
let etendu_of_float x = Reel x;;
```

Pour manipuler des types sommes on va utiliser des filtrages:

```
let logetendu x = match x with
| x when x = Reel 0. -> Moinsinfini
| Plusinfini -> Plusinfini
| Reel x when x > 0. -> Reel (log x)
| _ -> failwith "x est negatif";;

let addition r s = match (r, s) with
```

```
| Reel x, Reel y -> Reel (x +. y)
| Moinsinfini, Reel x -> Moinsinfini
| Plusinfini, Reel x -> Plusinfini
| Reel x, Plusinfini -> Plusinfini
| Reel x, Moinsinfini -> Moinsinfini
| a, b when a = b -> a
| _ -> failwith "non defini";;
```

1.6 Les exceptions

Le principe de l'exception est de créer un comportement brutal de l'algorithme afin d'interrompre son exécution, exemple: exception division par zéro.
Utiliser avec l'instruction raise, exemple:

```
let div a b =
    if b = 0 then raise divparzero else a /. b;; (* sure ? *)
```

Une exception peut etre ratrappée pendant une exception:

```
try
    (* bloc d'instructions *)
with
    (*filtrage *)
exception trouvee;; (* ? *)

let cherche t x =
    try
        for i = 0 to (Array.length t) - 1 do
            if t.(i) = x then raise trouve;;
        done;
    false;
    (* with ? *)
```

2 Analyse d'algorithmes

2.1 Introduction

Analyser un algorithme c'est trouver sa terminaison (montrer qu'il se terminera en un temps fini). Sa correction (le résultat fournis est-il celui attendu) et étudier sa complexité temporelle (se termine-t-il en un temps raisonnable).

2.2 La notion de complexité

En pratique, pour comparer des algorithmes, résolvant le même problème, on introduit une mesure appelée complexité. La complexité temporelle $T(n)$ est le nombre d'opérations élémentaires que va effectuer l'algorithme lors de son éxécution en fontion de la taille n des données en entrée.
Recherche dans un tableau trié.
Recherche naive: le nombre d'opérations dans le pire des cas (valeur absente du tableau) est égale à la taille du tableau -> linéaire. Recherche dichotomique: $log_2\ n = \frac{log(n)}{log(2)}$ -> complexité logarithmique.

Notations de Landeau:

f, g: $\mathbb{N}$ -> RENVOIE

On note que $f = O(g(n))$ s'il existe $M > 0$, N appartenant à $\mathbb{N}$, tel que pour tout $n > N$, $f(n) <= M * g(n)$

$F(n) = \Theta(g(n))$, s'il existe $M > 0$ et $m > 0$, tel qu'il existe N, tel que pour tout $n > N$, on a: $m * g(n) < f(n) < M * g(n)$

Les classes de complexité

Dénomination	Définition
Constant	$\Theta(1)$
Logarithmique	$\Theta(log(n))$
Linéaire	$\Theta(n)$
Quasi-linéaire	$\Theta(nlog\ n)$
Quadratique	$\Theta(n^2)$
Polynomiale	$\Theta(n^k)$ avec $k > 1$
Exponentielle	$\Theta(a^n)$ avec $a > 1$

3 Preuves de programmes impératifs

La preuve d'un programme a pour but de justifier qu'un programme se termine et que le résultat (ou l'action) du programme est bien celle attendue.
La preuve d'un programme impératif passe par un invariant de boucle.
Qu'il s'agisse d'une boucle for ou while, le schéma est le suivant:

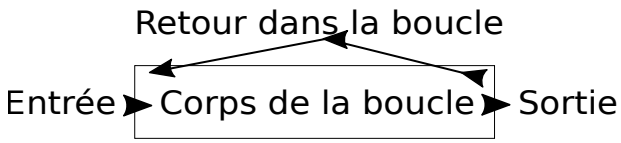

Figure 1:

Def: un invariant de boucle est un prédicat en i où i est le nombre de retours de boucle depuis le début de l'exécution de l'algorithme. Il décrit l'état de la mémoire juste après la i-ème boucle (pour $i = 0$ les conditions initiales)

La preuve de programme est alors:

1. on définit un invariant de boucle et on vérifie qu'il est vrai avant d'entrer dans la boucle
2. on prouve par récurrence que l'invariant reste vrai pour tout i jusqu'à la fin de l'exécution
3. on se sert de l'invariant pour prouver que le résultat est celui attendu.

Ex: Fonction factorielle

```
1 let fact n =
2     let res = ref 1 and m = ref n in
3         while !m > 0 do
4             res := !res * !m; decr m
5         done; !res;;
```

On note res_i et m_i les valeurs stockées dans res et m à la i-ème itération.
L'invariant est: res_i * $m_i!$ = $n!$
$i = 0$ $res_0 = 1$ $m_0! = n!$
vrai $i = 0$
H_i => H_{i+1} ?
$res_{i+1} = res_i$ * m_i
$m_{i+1} = m_i$ - 1
res_{i+1} * $(m_{i+1})!$ = res_i * m_i * $m_{i-1}!$ = res_i * $m_i!$ = n!
On sort de la boucle pour $m_j = 1$ et res_j * $m_j!$ = $res_j = res_0$ * $m_0!$ = $n!$

Ex: Algorithme du drapeau hollandais

On dispose d'un tableau dont les n cases ne peuvent contenir que les valeurs -1, 0 ou 1. On veut trier le tableau en temps linéaire et en place, c'est-à-dire sans créer de structure supplémentaire.

Description de l'algorithme:
On utilise 3 variables a, b, c initialisées à $a = b = 0$ et $c = n$ - 1
On maintient l'invariant suivant:

à la fin de la i-ème boucle
les cases d'indices:
- 0 à a - 1 contiennent -1
- a à $a+b$ - 1 contiennent 0
- c à n - 1 contiennent 1

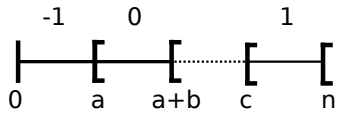

Figure 2:

Dans le cours de la boucle, on regarde le contenu de la case d'indice $a+b$ on l'échange selon qu'il s'agit d'un -1, 0 ou 1 et on met à jour les indices. On s'arrête lorsque $a+b > c$

```
let echange t i j =
    let c = t.(i) in
        t.(i) <- t.(j);
        t.(j) <- c;;

let drapeau t =
    let n = Array.length t in
        let a = ref 0 and b = ref 0 and c = ref (n - 1) in
            while !a + !b <= !c do
                match t.(!a + !b) with
                | 0 -> incr b
                | -1 -> echange t (!a + !b) !a; incr a
                | 1 -> echange t (!a + !b) !c; decr c;
            done;
        t;;
```

4 Preuves de programmes récursifs

4.1 Rappel sur les fonctions récursives

Une fonction récursive travaille avec une pile de strockage d'appels récursifs.
Il n'y a aucune création de variable.

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

Pour le calcul de fact 3, la pile évolue de la manière suivante:

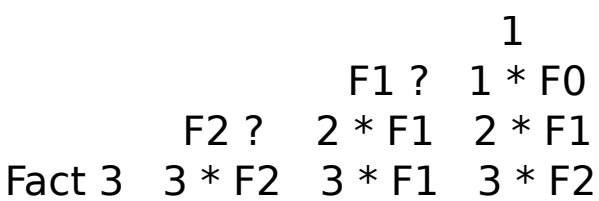

Figure 3:

4.2 Récursivité terminale

Def: On dit qu'une fonction est récursive terminale si pour chaque argument elle n'effectue qu'un appel récursif et si cet appel est la dernière instruction.

Dans ce cas la pile de calculs en attente reste vide, la complexité en mémoire est constante.

```
let rec nonterm n =
    if n = 0 then ()
    else (nonterm (n - 1);
        print_string "";
        print_int n);;

(* B: ou *)

let rec nonterm n =
    if n != 0 then
    (
        nonterm (n - 1);
        print_string " ";
        print_int n
    );;

let rec terminale n =
    if n = 0 then ()
    else ( print_int n; print_string " ";
        terminale (n - 1));;

(* B: ou *)

let rec terminale n = (* show numbers in the contrary order *)
    if n != 0 then
    (
        print_int n;
        print_string " ";
        terminale (n - 1)
    );;
```

On peut souvent transformer une fonction récursive en fonction récursive terminale. {demande jamais écrire la fonction récursive terminale mais c'est parfois mieux en terme de complexité}
On s'en sort en définissant une fonction auxiliaire prenant des arguments supplémentaires.

```
let term n =
    let rec aux p q =
        if p > q then ()
        else
    print_int p; print_string " "; (* without parenthesis print_string is infinitively called *)
    aux (p + 1) q
    in aux 1 n;;

    (* or B: *)

let term n =
    let rec aux p q =
        if p <= q then
        (
            print_int p;
            print_string " ";
            aux (p + 1) q
        )
    in aux 1 n;;
```

Ex: Fibonacci

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

$T_n = T_{n-1} + T_{n-2} + 1$
$T_n = 0\ (\Phi^n)$ {nombre d'or}

```
let fiboterm n =
    let rec aux (x, y) n = match n with
    | 1 -> y
    | n -> aux (y, x + y) (n - 1)
in aux (0, 1) n;;
```

$T_n = T_{n-1} + 1$

```
let factorielle n =
    let rec aux res n = match n with
        | 0 -> res
        | n -> aux (n * res) (n - 1)
in aux 1 n;;
```

4.3 Terminaison et correction d'une fonction récursive

4.3.1 Fonction à un argument entier

Pour justifier la terminaison et la correction d'un algo récursif, il suffit d'appliquer le principe de récurrence avec le prédicat $P(n)$: le programme se termine et donne la bonne solution pour la valeur n.

4.3.2 Cas général

Def: Soit $(E, <=)$ un ensemble ordonné.
L'ordre $<=$ sur E est dit bien fondé s'il n'existe pas de suite strictement décroissante d'éléments de E.

Prop: Un ordre $<=$ sur un ensemble E est bien fondé ssi toute partie non vide de E admet un élément minimal.
$\forall A \in E, \exists a \in A, \forall m \in A, m \leq a \Rightarrow m = a, A \neq \emptyset$

Ex Sur $\mathbb{N}^2$, les ordres suivants sont bien fondés:
- $(a_1, b_1) \Leftarrow (a_2, b_2) \iff a_1 <= a_2$ et $b_1 \Leftarrow b_2$

- $(a_1, b_1) \leq \iff a_1 + b_1 < a_2 + b_2$ ou $(a_1 + b_1 = a_2 + b_2$ et $a_1 \leq a_2)$
Le principe de récurrence se généralise alors:

Thm: (principe d'induction)

Soit $(E, \leq)$ un ensemble bien fondé, $A \in E$, $A \neq 0$ et Φ:E ?
A, $\Phi(x) < x$.
On considère un prédicat P défini sur E:
- $\forall a \in A$, $P(a)$ est vrai - $\forall x \in E$
$AP(\Phi(x)) \Rightarrow P(x)$

Alors $P(x)$ est vrai $\forall x \in E$.

Rq Avec $E = \mathbb{N}$, $A = \{0\}$, Φ: $n \longmapsto n - 1$ on retrouve le principe de récurrence.

Ex La fonction de Ackerman est définie par:
$\text{Ack}(0, n) = m + 1$
$\text{Ack}(n, 0) = \text{Ack}(n - 1, 1)$
$\text{Ack}(n, m) = \text{Ack}(n - 1, \text{Ack}(n, m - 1))$

Pour justifier la terminaison de cette fonction, on munit $\mathbb{N}^2$ de l'ordre lexicographique et on prouve par induction que $\forall (n, p) \in \mathbb{N}^2$ le calcul de $\text{A}(n, p)$ se termine.

Posons $A = \{(0, p), p \in \mathbb{N}\}$
Pour tout $(n,p) \in A$, le calcul se termine.
Si $(n,p) \notin A$, on suppose que le calcul se termine pour tout $(n',p') < (n,p)$
Si $p = 0$ alors $A(n, 0) = A(n-1, 1)$ et $(n-1, 1) < (n, 0)$
donc par hypothèse le calcul se termine.

Si $p > 0$ alors $(n, p\text{ - }1) < (n,p)$ donc le calcul de $A(n,p-1)$ se termine et $(n-1, A(n, p-1)) < (n,p)$ donc le calcul de $A(n-1, A(n,p-1))$ se termine.

5 Diviser pour régner

5.1 Principes et premiers exemples

Principe: Pour traiter un problème de taille n, on peut procéder la manière suivante:
- On divise le problème en deux problèmes de tailles $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$
- On résoud récursivement le problème sur une ou plusieurs de ces parties
- On combine les résultats afin d'obtenir le résultat du problème initial

L'intérêt de cette approche est que l'on peut améliorer la complexité temporelle lorsque les opérations de division et de fusion ne sont pas trop coûteuse.

5.1.1 Recherche dans un tableau trié

Pour rechercher un élément dans un tableau trié, l'algorithme naïf consiste à comparer chaque élément avec l'élément cherché.
Dans le pire des cas, on est en $O(n)$.
On dispose d'une méthode plus rapide: la recherche dichotomique

L'opération élement que l'on cherche à minimiser est la comparaison entre deux éléments.

```
1 let recherche_dicho x t =
2     let rec cherche_entre i j =
3         if j < i then false
4         else match (i + j) / 2 with
5         | k when t.(k) = x -> true
6         | k when t.(k) < x -> cherche_entre (k + 1) j
7         | k -> cherche_entre i (k - 1)
8     in cherche_entre 0 (Array.length t - 1);;
```

Notons c_n le nombre de comparaisons nécessaires entre x et un élément du tableau dans le pire des cas.

$c_n = 2 + c_{\lfloor n/2 \rfloor} \qquad n \geqslant 1$
$c_0 = 0$
On en déduit $c_n = 2\ \lceil log_2\ n \rceil + 2$
D'où une complexité en O(log n).

5.1.2 Exponentiation rapide

Pour calculer la puissance n d'une quantité on a un algo linéaire évident:

```
1 let rec puiss x n = match n with
2 | 0 -> 1
3 | n -> x * (puiss x (n - 1));;
```

On fait n multiplications dans tous les cas.
On peut faire mieux avec l'exponentiation rapide:

$$\begin{cases} & = & 1 \text{ si } n = 0 \\ x^n & = & x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} \text{ si } n \text{ pair} \\ & = & x * x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} \text{ sinon} \end{cases}$$

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

let rec exprap x = function (* il y a une variable supplementaire *)
| 0 -> 1
| n -> let a = exprap x (n / 2) in
(if n mod 2 = 0 then 1 else x) * a * a;;
```

Rq: Dans ces deux exemples on a fait un seul appel récursif (sur une moitié de problème)

5.2 Tri fusion

On appelle algorithme de tri tout algo permettant de trier les éléments d'un tableau (liste ou array) ou d'une liste selon un ordre déterminé.
Les tris par comparaison se basent sur la comparaison entre paires d'éléments.
Le coût de l'algorithme est alors le nombre de comparaisons.
Les algos naifs ont une complexité en $O(n^2)$
Il est possible de faire mieux.
Le tri fusion est une application de DPR.
- On partage la liste en deux parties de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$
- On trie récursivement les deux listes
- On fusionne les listes triées

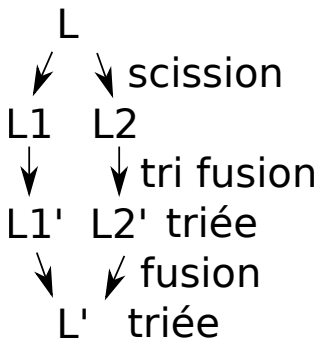

Figure 4:

La complexité est améliorée car les opérations de division et de fusion sont en temps linéaires.
On va écrire trois fonctions:
1. scission
2. fusion
3. trifusion

```
let rec scission l = match l with
| [] -> [], []
| [a] -> [a], []
| a::b::q -> let (l1, l2) = scission q in a::l1, b::l2;;
```

La fonction suivante prend en argument deux listes triées et renvoie la listée triée contenant les éléments des deux listes.

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

Il reste à écrire l'algorithme lui-même:

```
let rec trifusion l = match l with
| [] -> []
| [a] -> [a]
| l -> let (l1, l2) = scission l in fusion (trifusion l1) (trifusion l2);;
```

Complexité scission = chaque élément n'est vu qu'une seule fois -> O(n)
fusion = O($n1 + n2$)
trifusion $c_n = c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + O(n)$

D'après le théorème qui va suivre, on trouve une complexité en O($n$log n)

5.3 Un résultat général de complexité

On note $T(n)$ le coût maximal pour traiter une donnée de taille n, $S(n)$ le coût maximal pour séparer une donnée de taille n en deux parties équivalentes, $F(n)$ le coût maximal pour combiner les résultats des traitements de deux parties de tailles n / 2.
(on rassemble S et F en une seule fonction $f(n)$)

Thm S'il existe $p \geqslant 1$ tq
S(n) = O(n^p) et F(n) = O(n^p)
et si T(n) $\leqslant$ S(n) + 2 T(n/2) + F(n) alors:
T(n) = O($n$log n) si $p = 1$ (complexité semi logarithmique)
O(n^p) sinon

Plus généralement si T(n) $\leqslant$ S(n) + qT(n / 2) + F(n) $q \geqslant 1$, alors on a:

$$\begin{cases} & = & O(n^{log_2\ q}) \text{ si } 2^p < q \\ I(n) & = & O(n^{log_2\ q}\ log\ n) \text{ si } 2^p = q \\ & = & O(n^p) \text{ si } 2^p > q \text{ si } 2^p > q \end{cases}$$

(dans le 2^p, le p est celui du O(n^p) qui majore S et F)

On retrouve bien l'exemple de la dichotomie $q = 1$ et $p = 0$.

Preuve : En règle général, on prouve la complexité pour n une puissance de 2.
Un résultat mathématique admis permet de passer à n quelconque.

$n = 2^m$ et $u_m = c_n = c_{2^m}$
La relation de récurrence devient
$u_m = 2u_{m-1} + d_{2^n}$ avec d_n le coût de partage et de recomposition.
On a $\frac{u_m}{2^m} = \frac{u_{n-1}}{2^{n-1}} + \frac{d_{2^m}}{2^m}$
et donc $u_m = 2^n(u_0 + \sum_{j=1}^{n} \frac{d_{2^j}}{2^j})$
On rappelle que $2^m = n$ et $m = log_2 n$ - si d_n = O(n) alors $\exists K$ tq $d_{2^j} \leqslant K2^j$ d'où $\sum_{j=1}^{log_2 n} 1 \leqslant K log_2 n$ et finalement $c_n = O(n log_2 n)$
- si $d_n = O(n^p) p > 1$ on a
$u_m \leqslant K2^m(u_0 + \sum_{j=1}^{m} 2^{p-1} j \leqslant K2^n 2^{(p-1)m} \leqslant K2^{pm} = O(n^p)$

5.4 Distance minimale dans un nuage de points

On s'intéresse au problème de la recherche de deux points les plus proches dans un nuage $p_1 = (x_1, y_1); ...; p_n = (x_n, y_n)$ de points distincts du plan.
L'algorithme naïf consiste à calculer la distance entre toutes les paires possibles et d'en extraire le minimum.
Sachant qu'il y a $\binom{n}{2} = \frac{n(n-1)}{2}$ paire, cet algo a un coût quadratique.
Peut on améliorer la complexité à l'aide d'une stratégie DPR.
Algorithme:
- si le nuage contient moins de 4 points, on calcule toutes les distances.
- sinon: prétraitement: on trie le nuage par ordre d'abscisses croissantes d'une part et d'ordonnées croissantes.
a) On sépare le nuage en deux parties G et D de tailles comparables séparées par une droite d'abscisse $x_{lim} = x_{n/2}$
b) On calcule récursivement la distance minimale dans les nuages G et D (et les points correspondants).
c) On calcule la plus petite distance entre un point de G donné et un point de D donné (et les points correspondants).
d) On renvoie le minimum des 3 distances.

La difficulté consiste à obtenir la distance minimale entre les points des 2 demi-plans.
Soit $d = min(d_G, d_D)$. On cherche s'il existe un couple de points ayant une extrémité dans G et l'autre dans D, tel que: $d(P_G, P_D) \subset d$

Beaucoup d'algorithmes sont plus efficaces lorsqu'ils travaillent avec une liste triée donc autant la triée puis on en parle plus.

Si ce couple existe, on le cherche dans une bande T délimitée par les droites d'abscisses verticales $x_{lim} - d$ et $x_{lim} + d$.

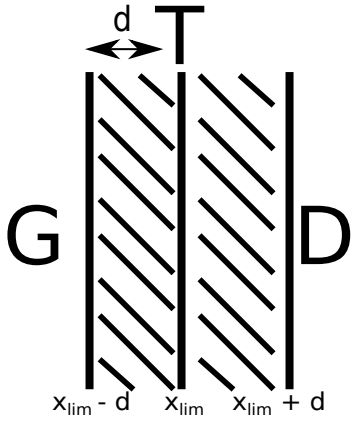

Figure 5:

Soit $M \in T$, on cherche les points $P \in T$ d'ordonnées supérieures, tel que d$(M, P) < d$

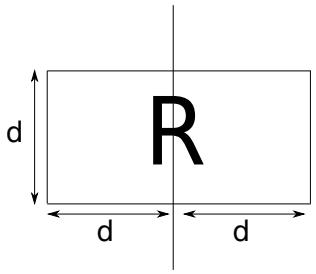

Figure 6:

Ces points appartiennent à un rectangle R de hauteur d.
On divise le rectangle R en 8 carrées de côté d / 2.

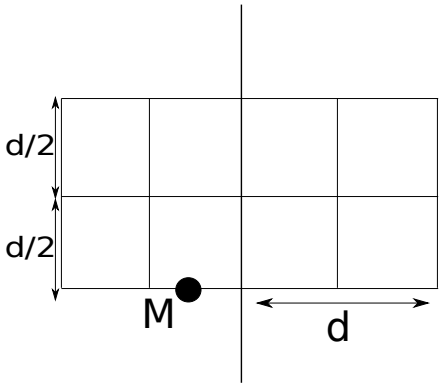

Figure 7:

Si deux points appartiennent au même carré alors leur distance est $\leqslant \frac{d}{\sqrt{2}} < d \Rightarrow$ imp

Donc chaque carré contient au plus un point.
Donc chaque rectangle R contient au plus sept autres poits du nuage.
D'où pour trouver la distance minimale entre un point de G et un point de D
a) On fait un parcours de P_{ord} en ne gardant que les points dont l'abscisse appartient à $[x_{lim} - d, x_{lim} + d]$
-> on a déterminé la bande T (O(n))
b) pour chaque $M \in T$ on calcule la distance aux sept points de T juste au dessus. Si la distance est inférieure à d, on actualise.

Coût du calcul de la distance minimale.
- prétraitement: (1 seule fois)
tri du plan P par ordre d'abscisses croissantes et d'ordonnées croissantes O($n$log n).
- à chaque appel
* partage: linéaire car on a trié (un seul parcours de P_{abs})
* sélection des points de T: linéaire un seul parcours de P_{ord} en ne gardant que les points $\in [x_{lim} - d, x_{lim} + d]$
* calcul des distances dans T linéaire en le nombre de points de T (1 parcours + 7 calculs)

D'où $c_n = c_{\lfloor n/2 \rfloor} + c_{\lceil n/2 \rceil} + O(n)$
D'où c_n = O($n$log n)

5.5 Multiplication rapide des polynômes (algorithme de Karatsuba)

On représente le polynôme $P = \sum_0^n a_k X^k$
à coefficients dans $\mathbb{Z}$ par un tableau de taille $n + 1 (P.(k) = a_k)$

Pas de Array.length dans un algo récursif. Degré du polynôme $\neq$ taille du tableau.

```
1  let som p q =
2      let degp = Array.length p - 1 in
3          let r = Array.make (degp + 1) 0 in
4              for i to degp do
5                  r.(i) <- p.(i) + q.(i)
6              done; r;;
```

Le produit $(\sum_{i=0}^n a_i X^i) * (\sum_{j=0}^n b_j X^j))$
$= \sum_{i=0}^n \sum_{j=0}^n a_i b_j X^{i+j}$
nécessite (n + 1)^2 multiplications.

```
1  let prod p q =
2      let degp = Array.length p - 1 in
3          let r = Array.make (2 * degp + 1) 0 in
4              for i = 0 to degp do
5                  for j = 0 to degp do
6                      r.(i + j) <- r.(i + j) + p.(i) * q.(j)
7                  done; done; r;;
```

On cherche à réduire le nombre de multiplications en utilisant une stratégie DPR.
On pose m = $\lceil n/2 \rceil$ puis
P = $X^n P_1 + P_2$
Q = $X^n Q_1 + Q_2$
Le degré de chacun des polynômes
P_1, P_2, Q_1, Q_2 est $\leqslant \lfloor n/2 \rfloor$
Ainsi
PQ = $X^{2n} P_1 Q_1 + X^n (P_1 Q_2 + P_2 Q_1) + P_2 Q_2$
- Le coût de décomposition en polynôme de degré n / 2 est constant (accès aux coefficients de chaque polynôme).
- 4 appels récursifs sont nécessaires.
- Le coût de recomposition est linéaire (3 additions).
Le nombre de multiplications vérifie donc:
$c_n = 4c_{\lfloor n/2 \rfloor} + O(n)$
$p = 1$ et $q = 4$
$2^p < q$

$O(n^{log_2\ 4}) = O(n^2)$
On a rien gagné !

En revanche si l'on remarque que
$P_1Q_2 + P_2Q_1 = (P_1 + P_2)(Q_1 + Q_2) - P_1Q_1 - P_2Q_2$, on a alors:
$R_1 = P_1Q_1$
$R_2 = (P_1 + P_2)(Q_1 + Q_2)$
$R_3 = P_2Q_2$
PQ $= X^{2n}R_1 + X^n(R_2 - R_1 - R_3) + R_3$
On se ramène à 3 appels récursifs.
La relation de récurrence devient: $c_n = 3c_{\lfloor n/2 \rfloor} + O(n)$
$p = 1$ et $q = 3$ et $2^p < q$
complexité en $O(n^{log_2\ 3})$
avec $log_2\ 3 \approx 1.585$

Implémentation:
On suppose que les polynômes sont stockées dans des tableaux de taille 2^k (quitte à rajouter des 0) qui seront d'abord normalisés puis simplifiés.

```
let rec kara p q = match (Array.length p) with
| 1 -> [| p.(0) * q.(0); 0|]
| n -> let m = n / 2 in
    let p1 = Array.sub p m m and p2 = Array.sub p 0 n in (* error here ? n/m *)
    let q1 = Array.sub q m m and q2 = Array.sub q 0 m in (* on peut faire que des and *)
    let r1 = kara p1 q1 in
    let r2 = kara (somme p1 p2) (somme q1 q2) in
    let r3 = kara p2 q2 in
    let s = Array.make (2 * n) 0 in
        for i = 0 to n - 1 do
            s.(n + i) <- s.(n + i) + r1.(i);
            s.(n + i) <- s.(n + i) + r2.(i) - r1.(i) - r3.(i);
            s.(i) <- s.(i) + r3.(i)
            done; s;;
```

6 Programmation Dynamique

6.1 Exemple introductif coefficients binomiaux

On s'intéresse au calcul du coefficient:
$\binom{n}{p}$ Il est très tentant d'utiliser la formule de Pascal:
$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$

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

Cette fonction a l'avantage d'être claire et l'inconvénient d'être très peu efficace.

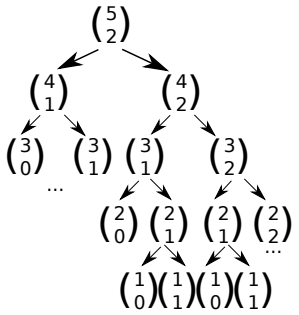

Figure 8:

Le calcul de $\binom{n}{p}$ se ramène à la résolution de deux sous problèmes $\binom{n-1}{p-1}$ et $\binom{n-1}{p}$ qui ne sont pas indépendant (les problèmes que l'on résoud pour l'un sont utiles pour l'autre). On commence par résoudre les plus petits sous problèmes et on combine leurs solutions pour résoudre les sous problèmes de plus en plus grands.

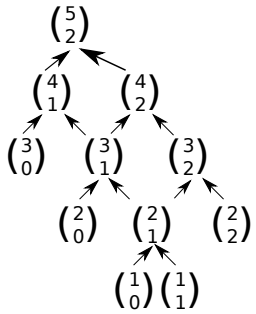

Figure 9:

On stocke les valeurs des coefficients binômiaux pour ne pas les recalculer plusieurs fois.

Rappel: on n'a pas un accès en temps constant au p-ième élement d'une liste

```
let binom n p =
    let c = Array.make_matrix (n + 1) (p + 1) 1 in (* just not to bully about indexes *)
        for i = 2 to n do
            for j = 1 to min (i - 1) p do (* for i index it already valued 1 *)
                c.(i).(j) <- c.(i - 1).(j) + c.(i - 1).(j - 1)
            done;
        done;
    c.(n).(p);; (* could use only two lines of data *)
```

On est en $O(np)$ en temps et en espace.
On a ajouté un coût spatial mais gagné en efficacité.
C'est l'intuition de la programmation dynamique.

Principe
Trouver la valeur d'une solution optimale en calculant la solution de sous problèmes.
Il faut d'abord trouver la relation entre les deux (si elle existe).
Différence avec DPR la solution d'un sous problème ne peut pas être réutilisée (DPR crée des problèmes indépendants)

6.2 La chaîne de montage (Programmation Dynamique)

6.2.1 Description du problème

Une entreprise a un atelier contenant deux chaînes de montage ayant chacune n stations de travail par lesquelles transitent les produits en construction.

On cherche à construire un produit le plus vite possible, éventuellement en passant d'une chaîne à l'autre entre deux postes.
Le transfert d'une chaîne à l'autre a un coût, rester sur la même chaîne n'en a pas.

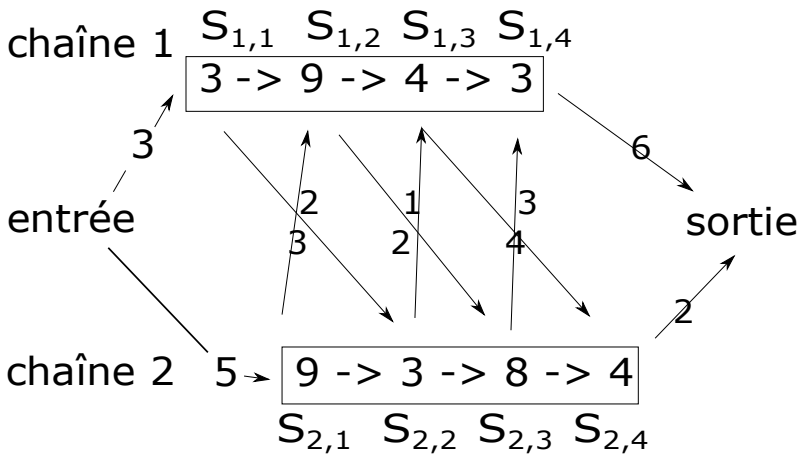

Figure 10:

Exemple

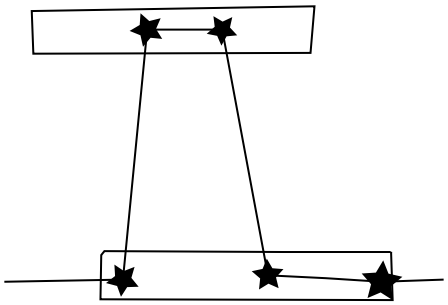

Figure 11:

Le problème consiste à traverser en un minimum de temps. la méthode naïve consistant à calculer tous les chemins possibles est en $O(2^n)$.

6.2.2 Sous-structures optimales et relations de récurrences

On note $\forall (i,j) \in \{1,2\} * [\![1,n]\!]$
$m_{i,j}$ le coût minimal pour aboutir à la sortie de $S_{i,j}$
$C_{i,j}$ un chemin de coût minimal pour aboutir à la sortie du poste $S_{i,j}$
$C_{i,j}$ est représenté par un élement de $\{1,2\}^j$
Les temps de parcours sur la chaîne i sont notés $t_{i,j}$ avec $j \in [\![0,n+1]\!]$
$t_{i,0}$ temps d'accès à la chaîne
$t_{i,n+1}$ temps de sortie de la chaîne
pour $j \in [\![1,n]\!] t_{i,j}$ est le temps de traversée de la station $S_{i,j}$
$m_{11} = t_{10} + t_{11}$ $C_{11} = \{\{1\}\}$
$m_{21} = t_{20} + t_{21}$ $C_{21} = \{\{2\}\}$
Le temps de transfert du poste $S_{1,j}$ au poste $S_{2,j+1}$ (respectivement $S_{2,j}$ vers $S_{1,j+1}$) est noté $e_{1,j}$ (resp. $e_{2,j}$).
Soit un chemin optimal et soient $S_{i_1 1}, S_{i_2 2}, S_i ...$ la suite des postes traversés (avec $\forall k \in [\![1,n]\!] i_k \in \{1,2\}$)

La première constatation qui conduit à la mise en place d'une stratégie dynamique est que pour tout entier $j \in [\![1,n]\!]$ le chemin qui mène au poste $S_{i_j,j}$ est nécessairement optimal.
Dans le cas contraire, on pourrait remplacer cette partie du chemin par un trajet prenant un temps strictement inférieur et obtenir un chemin jusqu'à la sortie de poids strictement inférieur au chemin initial.
Par conséquent un chemin optimal est composé de sous chemins optimaux.

Soit $j \in [\![1,n]\!]$ un chemin optimal jusqu'à la sortie de $S_{1,j}$ est constitué soit d'un chemin optimal jusqu'à la sortie de $S_{1,j-1}$ soit d'un chemin optimal jusqu'à la sortie de $S_{2,j-1}$ et d'un transfert.

D'où:

$m_{1,j} = t_{1,j} + min(m_{1,j-1}, m_{2,j-1} + e_{2,j-1})$
Pour finir le temps minimal de traversée est
$m = min(m_{1,n} + t_{1,n+1}, m_{2,n} + t_{2,n+1})$
Les éléments de $C_{1,j}$ s'obtiennent de la manière suivante:
si $m_{1,j-1} < m_{2,j-1} + e_{2,j-1}$
$C_{i,j} = C_{1,j-1}, \{1\}$

si $m_{1,j-1} > m_{2,j-1} + e_{2,j-1}$
$C_{1,j} = C_{2,j-1}, \{1\}$

6.3 Calcul du temps optimal et d'un chemin optimal

```
1  (* t1 = [|t1,0; t1,1; t1,2; ...; t1,n+1|] n+2 cases
2  e1 = [|e1,1; e1,2; ...; e1,n-1|] n-1 cases*)
3  let plusCourtChemin t1 t2 e1 e2 =
4      let n = Array.length t1 - 2 in
5      let m1 = Array.make (n+1) 0 in
6      let m2 = Array.make (n+1) 0 in
7      let c1 = Array.make n [1] in
8      let c2 = Array.make n [2] in
9      m1.(1) <- t1.(0) + t1.(1);
10     m2.(1) <- t2.(0) + t2.(1);
11     for j = 1 to n-1 do
12         let a = m1.(j) + t1.(j+1) and
13         b = m2.(j) + e2.(j-1) + t1.(j+1) in
14         if a < b then
15             (m1.(j+1) <- a; c1.(j) <- 1::(c1.(j-1)))
16             (* on peut c1.(j-1)@[1] mais la complexite devient n^2 *)
17         else
18             (m1.(j+1) <- b; c1.(j) <- 1::c2.(j-1));
19         let c = m2.(j) + t2.(j+1) and
20             d = m1.(j) + e1.(j - 1) + t2.(j+1) in
21         if c < d then
22             (m2.(j+1) <- c; c2.(j) <- 2::c2.(j-1))
23         else
24             (m2.(j+1) <- d; c2.(j) <- 2::(c1.(j-1)))
25     done;
26     let e = m1.(n) + t1.(n+1) and f = m2.(n) + t2.(n+1) in
27     if e < f then (e, List.rev c1.(n-1))
28     else (f, List.rev c2.(n-1));;
```

begin/end autour de bloc d'instructions camel ? semaine pro: DS info caml (written the 29/05/19)

6.4 Multiplication matricielle

6.4.1 Problématique

Soient $p, q, r \in \mathbb{N}^3$
Le produit de deux éléments $A \in M_{p,q}(\mathbb{K})$ et $B \in M_{q,r}(\mathbb{K})$
se fait par la formule
$\forall (i,j) \in [\![1,p]\!] * [\![1,r]\!], (AB)_{i,j} = \sum_{k=1}^{q} A_{ik} B_{kj}$
Il nécessite O(pqr) multiplications.
On considère maintenant un produit compatible de n matrices $A_0, A_1, ..., A_{n-1}$
dont les dimensions sont notées (p_i, p_{i+1}) $i \in [\![0, n-1]\!]$.
Le produit matriciel étant associatif il y a plusieurs façons de calculer
$A_0 A_1 ... A_{n-1}$
Le temps de calcul va dépendre de l'ordre des multiplications.
Par exemple: si A_0, A_2 sont des matrices lignes (1, n) et A_1, A_3 sont des matrices colonnes (n, 1), le calcul de $A_0 A_1 A_2 A_3$ prend:

- $2n + 1$ multiplications avec le parenthèsage $(A_0A_1)(A_2A_3)$
- $2n^2 + n$ multiplications avec le parenthèsage $(A_0(A_1A_2))A_3$

On cherche le parenthèsage optimal c'est à dire qui minimise le nombre de multiplications.

Remarque: le nombre de parenthèsage total pour n matrices
$c_1 = 1$
$c_n = \sum_{i=1}^{n-1} c_i * c_{n-i}$ $\frac{4^{n-1}}{\sqrt{\pi} n^{\frac{3}{2}}}$

6.4.2 Analyse du problème

Pour calculer le produit $A_0...A_{n-1}$ de manière optimale, il faut pour $k \in [\![0, n-2]\!]$ calculer les produits $(A_0...A_k)$ et $(A_{k+1}...A_{n-1})$ où les calculs de $(A_0...A_k)$ et $(A_{k+1}...A_{n-1})$ sont faits de manière optimale.

Supposons connu $\forall k \in [\![0, n-2]\!]$ le coût minimal $m_{0,k}$ pour calculer $A_0...A_k$ et $m_{k+1,n-1}$ calculer $A_{k+1}A_{n-1}$ alors le coût minimal $A_0...A_{n-1}$ avec la décomposition précédente est donnée par
$m_{0,n-1} = min_{k \in [\![0,n-2]\!]}(m_{0,k} + m_{k+1,n-1} + p_0p_{k+1}p_n)$

$(A_0...A_k)(A_{k+1}...A_{n-1})$ coût: $m_{0,k} + m_{k+1,n-1} + p_0p_{k+1}p_n$

Plus généralement, le coût minimal $m_{i,j}$ pour calculer le produit $A_i...A_j$ vaut 0 si $i = j$ et lorsque $j > i$
$m_{i,j} = min_{k \in [\![i,j-1]\!]}(m_{i,k} + m_{k+1,j} + p_ip_{k+1}p_{j+1})$

Cette formule de récurrence permet de calculer de proche en proche le coût minimal $m_{0,n-1}$.
Pour trouver le parenthèsage optimal il suffit de conserve k_{ij} lorsque m_{ij} est calculé.

6.5 Généralités sur la programmation dynamique

Les problèmes pouvant se résoudre efficacement par une méthode programmation dynamique présentent les caractéristiques suivantes:
Sous-structure optimale
Une solution optimale est constituée de solutions optimales à des sous-problèmes.
Espace des sous-problèmes
La recherche d'une solution optimale est ramenée à la résolution progressive d'un certain nombre de sous-problèmes.
- pour la chaîne de montage: calcul du chemin optimal vers n'importe quel poste
- pour le produit matriciel, coût optimal de n'importe quel A_iA_j

Remarque on ne peut pas se contenter, du coût optimal A_0A_k
Espace des sous-problèmes pour la chaîne de montage est en $O(n)$ et $O(n^2)$ pour le produit.

Un algorithme utilisant la récurrence est descendant (on part du gros problème aux petits) tandis qu'un algorithme de programmation dynamique bottom-up/ascendant part des petits problèmes pour remonter au gros

Reconstruction des solutions optimales
Pour construire une solution optimale un algorithme de programmation dynamique explore les coûts des différents sous-problèmes et construit sa solution en faisant un choix qui minimise le coût global.
Complexité
Les temps d'éxécution d'un algorithme de programmation dynamyque dépend du produit de 2 facteurs: le nombre de sous-problèmes à résoudre et le nombre de choix pour chaque sous-problèmes.
- chaîne de montage: il y a $O(n)$ problèmes à résoudre et deux choix à chaque fois: le temps d'éxécution est en $O(n)$
- pour le produit matriciel: on a $O(n^2)$ problèmes à résoudre, et au plus $n - 1$ choix, d'où une complexité en $O(n^3)$

7 Structures de données

7.1 Structure de données abstraites

Def: On appelle type de donnée abstrait (TDA) un modèle formel de données structurées.
On appelle structure de données abstraite un TDA muni d'opérations.

Ex: Listes, vecteurs, piles, files, arbres, dictionnaire, tas...

Types d'opérations:
- construction / initialisation
initialiser une instance de type de donnée, ex: Array.make - décision: retourne un booléen, ex: is_empty sur les piles et files - accès: retourne une information, ex: Array.length
- modification: transforme la structure ou le contenu de l'instance On dit qu'une modification est persistante si elle retourne une nouvelle copie de l'instance (ex: :: pour les listes)

> Dans un programme impératif (contrairement à un programme récursif) a :: l doit être remplacé par a ::! l

impérative si elle agit directement sur la structure (ex: $<-$)

Def: une structure est dite statique si elle est de taille fixe, dynamique sinon.

Listes: persistantes et dynamiques Tableaux: impératifs et statiques

7.2 Piles et files

Les piles (stack) et les files (queue) sont des structures de données fondamentales en informatique. Elles se distinguent par les conditions d'ajout et d'accès aux éléments.
- les piles sont fondées sur le principe LIFO (Last In First Out)
- Les files sont fondées sur le principe FIFO (First In First Out)

7.2.1 Piles

Soit t un type. On définit une pile comme séquence finie (éventuellement vide) d'éléments de type t.
Le premier élément d'une pile est appelé sommet

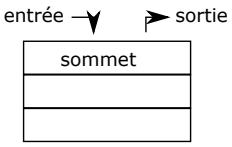

Figure 12:

Une pile est une structure dynamique où les insetions et suppressions se font au sommet de la pile.
Opérations: création d'une pile vide, ajout d'un élément au sommet (empiler), extraction de l'élement au sommet (dépiler), regarder l'élément au sommet sans l'enlever, demander à la pile si elle est vide. Module stack

```
1 open Stack;;
2 let pile = create();;
3 for i = 1 to 5 do push i pile done;;
4 top pile;;
5 pop pile;;
6 is_empty pile;;
```

7.2.2 Files

Soit t un type. On définit une file sur t comme une séquence finie (éventuellement vide) d'éléments de type t.
Le premier élément d'une file non vide est appelé tête de file.
L'entrée de la file est la place où insérer un nouvel élément, il se trouve après le dernier élément de la file.
Opéations les insertions se font en queue de file et les suppressions en tête de file

Module Queue

```
1 open Queue;;
2 let file = create();;
3 for i = 1 to 5 do add i file done;;
4 take file;;
```

```
5 is_empty file;;
6 for i = 1 to 4 do print_int (take file) done;;
```

DS: on ne peut pas accéder à un élément d'une liste

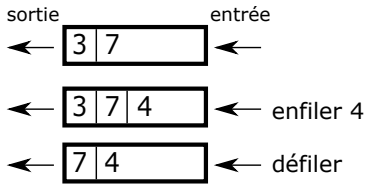

Figure 13:

fst (1, 2);; renvoie 1
List.map (fun x -> x * x) [1;2; 3];; renvoie les carrés
<> comme opérateur différent
String.length str
();;
Array.sub v deb long (borne supérieure exclue)
La complexité de la fonction Array.sub est linéaire en la taille du vecteur résultat, donc O(n).
Array.of_list