Cours d'informatique

Monsieur Taïeb

Xavier Ploquin

10 mai 2006

SOMMAIRE

TABLE DES MATIÈRES

Première partie

Initiation à CAMLlight

CHAPITRE 1

Premiers pas en CAML light

CAMLlight est un langage compilé/interprété. Il faut taper le programme, le compiler, puis on peut l'interpréter. Il faut donc utiliser un interpréteur :

Code CAML :
```
>> 3 + 2 ;;
Réponse : 5, int
```

Le symbole « ; ;» déclenche la compilation.

int correspond au *type* entier. Un nombre de type entier est représenté en machine de façon exacte, écrit en base 2 sur 32 (ou 64) chiffres (ou bits). Un entier appartient donc à $[\![-2^{31}, 2^{31}-1]\!]$.

Code CAML :
```
>>2 000 000 000 + 1 000 000 000 ;;
int, -1 000 000 000
```

♡ Attention : En cas de dépassement le résultat est faux car il est calculé modulo 2^{32}.

Sous CAML tous les objets ont un type. Il existe d'autres types, notamment le type *float*, c'est le type «flottant» qui correspond à la notation scientifique des nombres : $1,243 \cdot 10^{82}$exposant, où 1,234 s'appelle la *mantisse* et 82 l'*exposant* Un nombre codé en flottant est un nombre réel approché. Tous les calculs effectués sont approchés.

Code CAML :
```
>> 1.5 + 1.5 ;;
Erreur, problème de type.
```

En effet «+» correspond à l'addition de deux entiers

Code CAML :
```
>> 1.5 +. 1.5 ;;
float, 3.000000001

>> 1.5 + 2 ;;
N'est pas possible ! Il faut donc rentrer :
>> 1.5 +. 2. ;;
float, 3.4999999999
```

1.1 Les variables

Code CAML :
```
>> let x = 12 ;;
x : int, 12

>> 3 * x + 1 ;;
int, 37

>> let y = x + 1 ;;
y : int, 13

>> let x = x +1 ;;
Il râle ! On ne modifie pas une variable !
```

1.2 Les fonctions

Code CAML :
```
>> let x = function x -> 3*x + 2 ;;
f: int->int

>> f(3) ;;
int,11
```

Variables de déclaration

Code CAML :
```
>> let f(x) = 3*x + 2 ;;
```

Un peu plus compliqué :

Code CAML :
```
>> let f(x,y) = x+y ;;
f: int*int -> int

>>f(3,4);;
int,7
```

«int*int» est le type «couple d'entiers».

Encore un peu plus compliqué :

Code CAML :
```
>> let f(g) = function x -> g(x+1)
f:(int->'a)-> (int->'a)
>>let g(x)=2*x ;;
>>let h = f(g);;
>>h(3);;
int, 8
```

Ici «'a» désigne un type quelconque mais les deux 'a désignent le même type, sinon caml aurait utilisé un «'b» etc.

1.3 Variables locales

Il est possible de déclarer des variables localement uniquement, ie : à l'intérieur d'un calcul.

Code CAML :
```
>> let a = ln 2 in a *. a ;;
float, 0.37
>>a;;
Erreur, connais pas
```

ou bien :

Code CAML :
```
>>let carré(x) = x *. x in carré(ln 2) ;;
float,0.37
>>carré(1.2);;
Erreur connais pas.
```

1.4 Curryfication

Ecrire autrement les fonctions à plusieurs variables.

Code CAML :
```
>> let somme x y = x + y ;;
somme : x -> function y -> (x + y)
somme int-> int-> int
>>somme(2,3);;
erreur, vous avez utilisé somme avec un couple d'entiers.
>>somme 2 3 ;;
int, 5
>>let f = somme 2;;
f::int -> int
>>f(3);;
int,5
```

L'intéret est de pouvoir ne placer qu'une partie des variables.

1.5 Exercices

Exercice 1.5.1: *Quelques calculs*

```
– >>7./.22.;;
– >>sin(1. /. 3.);;
– >>exp(sqrt(3.) +. 2.) ;;
```

Exercice 1.5.2: *De plus gros calculs...*

```
– >>let x = sqrt(3.) in exp(x)/. x;;
– >>let x = ln 7. in sin x *. x;;
– >>let x = cos(sqrt(2.))
    and y=sin(log(3.)) in
            (x-.y)/.(x+.y);;
```

Exercice 1.5.3: *Des fonctions*

```
– >>let f(x) = x+1 ;;
– >>let f(x) = exp(x*.(ln x));;
– >>let f x = 2. *. float_of_int(x)*.3.14;;
– >>let f x =
  if abs(x) <=1e(-7) then 1.
  else sin(x)/.x;;
```

Il ne faut pas faire de test d'égalités sur des float !

Exercice 1.5.4: *Radians & degré*

```
>>let g f x =
f( 2. *. 3.14159 * x /. 360.);;
```

Exercice 1.5.5: *D'autres fonctions*

```
– >>let f g = (g(0.)+.g(1.))/. 2. ;;
– >>let g(f,x) = f(x) + 2;;
  (('a->int)*('a))->int
```

Exercice 1.5.6: *Des fonctions plus compliquées...*

```
– >>let f x = sin (log x) in
       f(3.)/.f(cos(1.));;
– >>let f x =
  sqrt(2.) *. exp(x *. sqrt(2.)) in
  f(3.)/.exp(f(3.14159));;
```

Ou mieux :

```
>>let a = sqrt(2.) in
  let f x = a*. exp(a*.x) in
     f(3.)/.exp(f(3.14159));;
```

Exercice 1.5.7: *Toujours plus compliqué !*

```
>>let f x =
let a = sqrt(2.) in
let f y = a*.exp(a*.y) in
let b = sin x in
f(x)/.exp(f(b));;
```

Exercice 1.5.8: *Minimum de deux fonctions*

```
>>let inf (f,g) x =
let a = f(x) and b = g(x) in
if a<b then
a
else
b;;

type :  (('a->'b)*('a->'b))->('a->'b)
```

Exercice 1.5.9: *dérivation & développements limités*

```
>>let der f x =
(f(x+.0.000000001)-.f(x))/.(0.000000001);;

>>let DL2 f x0 x =
let a = der f x0 in
let b = der f a in
let c = x-x0 in
f(x0)+.(c)*.a+.(c)*.c/.2*.b;;
```

Exercice 1.5.10: *Mystère*

La fonction est de type (('a*'a)->'b)*'a*'a)->'b

CHAPITRE 2

Itération

L'itération consiste à effectuer plusieurs fois une même opération.

Itération inconditionnelle : On prend un ensemble fini E ordonné et pour $x \in E$ (pris dans l'ordre), on effectue $truc(x)$.

Itération conditionnelle :

- Tant qu'une certaine condition n'est pas vérifiée, faire truc.
- Faire truc jusqu'à ce qu'une certaine condition soit réalisée. (L'opération est effectuée, au moins une fois)

En CAML on dispose de l'itération inconditionelle («for» avec $E = [\![a, b]\!]$), et de l'itération inconditionnelle avec test au début («while »).

Code CAML : *Exemples*

```
>>let carré n =
     for i = 1 to n do
         print_int(i*i);
         print_newline();
         done;;
```

2.1 Les références

Si l'itération est destinée à faire évoluer des valeurs, on utilise des variables dont le contenu est destiné à bouger : des références. Une référence de type 'a est une feuille de papier sur laquelle est écrit un élément de type 'a.

Deux opérations sur les référence : écrire quelquechose dessus et en lire la valeur.

Code CAML : *Déclaration d'une référence*

```
>>let a = ref 3;;
>>!a;;
     int,3
>>a:=5;;
>>!a;;
     int,5
>>a:=!a+1;;
>>!a;;
     int,6
```

Code CAML :
```
>>let facto n =
     let a = ref 1 in
     for i = 2 to n do
          a:=!a+1;
          done;
     !a;;
```

Si n = 1, la boucle est effectuée pour tout $i \in \emptyset$ et n'est donc pas effectuée.

Code CAML : *Test de primalité*
```
>>let estpremier n =
     let onatrouvé = ref false and k = ref 2 in
     while (!onatrouvé=false)&&(!k*!k<=n) do
          if (n mod !k) = 0 then
               onatrouvé := true;
          incr(k);
          done;
     not(!onatrouvé);;
```

2.2 Structure de tableau

Un tableau est un groupe de références, une feuille sur laquelle on a n valeurs de type 'a. (tableau = nombre d'éléments, adresse mémoire de la première valeur, les autres suivent)

Code CAML : *Déclaration d'un tableau*
```
>>let t = make_vect 18 2.3;;
```

Où 18 est le nombre de cases et 2,3 la valeur d'utilisation mise dans toutes les cases : CAML cherche un espace mémoire disponible long de 18 flottants, initialise toutes les cases et renvoie l'adresse du premier élément.

Code CAML : *Affectation d'une valeur :*
```
>>t.(4)<-3.2;;
```

Code CAML : *Lecture d'une case :*
```
>>t.(4);;
```

♡ Attention : la numérotation commence à 0. La première est 0 et la dernière n-1.

Code CAML : *Nombres d'éléments d'un tableau*
```
>>vect_length t;;
```

Code CAML : *Autre manière de définir un tableau :*
```
>>let t = [|3;4;3;8|];;
```

Code CAML :
```
>>let carrés n =
     let t = make_vect n 0 in
     for i = 1 to n do
          t.(i-1)<-i*i;
          done;
     t;;
```

2.3 Exercices

Exercice 2.3.1: *Somme & produit*

```
>>let sum f a b =
let s = ref 0 in
for i = a to b do
s:=!s +f(i);
done;
!s;;
>>let prod f a b =
let s = ref 1 in
for i = a to b do
s:=!s *f(i);
done;
!s;;
```

Cette fonction est de type «int->int->int->int->int»

Exercice 2.3.2: *Suites récurrentes*

```
>>let iter f u0 n =
let suite = ref u0 in
for i = 1 to n do
suite := f(!suite);
done;
!suite;;
```

Cette fonction est du type «int->int->int->int->int» Preuve du programme : un invariant de boucle : après la ième itération, c contient U_n.

Exercice 2.3.3: *Test de primalité*

cf : Cours

Exercice 2.3.4: *Somme de chiffres*

```
>>let somchiffres n =
let somme = ref 0 in
let quotient = ref abs(n) in
while !quotient <> 0 do
somme :=!somme + (!quotient mod 10);
quotient:=!quotient/10;
done;
!somme;;

>>let somcubes() =
    let cube x = x*x*x in
    let i = ref 1 and cubmax = ref 0
    and result:=1 in
        while (cube !i <=10000) do
        let a = somchiffres(cube !i) in
        if (a >!cubmax) then
           (cubmax:=a result:=i);
        incr i;
done;
result;;
```

Cette fonction est du type «int->int».(Le abs permet de traiter aussi les nombres négatifs.)

Exercice 2.3.5: *Suite de syracuse*

```
>>let syracuse n =
     let k = ref 0 in
     let suite = ref n in
     while !suite <>1 do
          if (!suite mod 2)=0 then
               suite:=!suite/2
          else
               suite:=3*!suite+1;
          incr(k);
          done;
     !k;;
```

Exercice 2.3.6: *Map*

```
>>let map f t =
     let a = vect_length (t) in
     let u = make_vect a f(t.(0)) in
     for i = 1 to pred(a) do
          u.(i)<-f(t.(i));
          done;
     u;;
```

Exercice 2.3.7: *égalité de tableaux*

```
>>let test t u =
     let b = ref true in
     let n = vect_length t in
     let m = vect_length u in
     let k = ref 0 in
     if n<>m then
          b:=false
     else
          while (!b)&&(!k<n) do
               if t.(!k)=u.(!k) then
                    incr(k)
               else
                    b:=false;
               done;
     !b;;
```

La complexité en temps de l'algorithme est le temps d'exécution. Ici si n est la taille du tableau, la complexité vaut $\mathcal{O}(n)$ dans le pire des cas.

Exercice 2.3.8: *Minimum d'un tableau*

```
>>let mint t =
     let a = vect_length t in
     let k = ref t.(0) in
     for i = 1 to pred (a) do
          if t.(i)<(!k) then
               k:=t.(i);
          done;
     !k;;
```

Exercice 2.3.9: *Maximum d'un tableau et nombre d'occurrences en lecture unique*

```
>>let maks t =
     let k = ref 1 in
     let n = vect_length t in
     let m = ref t.(0) in
     for i = 1 to pred(n) do
          if t.(i)=(!m) then
               incr(k)
          else if t.(i)>(!m) then
               (m:=t.(i); k:=1);
          done;
     (!m,!k);;
```

Cette fonction est d'une complexité linéaire.(ie : du type $\mathcal{O}(1)$, appliqué n fois.)

CHAPITRE 3

Quelques algorithmes usuels sur les tableaux

3.1 recherche d'un élément dans un tableau

On dispose d'un tableau t et d'un élément e. On parcourt le tableau jusqu'à ce que l'on trouve e, ou bien que l'on arrive à la fin du tableau. On retourne à la fin un booléen résultat du test $e \in t$, et éventuellement l'indice, dans un couple.

Code CAML :

```
>>let recherche e t =
     let j = ref 0 in
     let resultat = ref false in
          while (!resultat=false)
          && (!j<vect_length t) do
            if t.(j)=e then
               result:=true;
               incr(j);
          done;
     !resultat;;
```

La complexité $\mathcal{O}(n)$ où n est la longueur du tableau dans le cas le pire (si e n'y est pas).

3.2 Recherche des éléments dans un tableau trié

On dispose d'un talbeau trié $(t.(0) \leq t.(1) \leq \cdots \leq t.(n-1))$ et d'un élément e.Si le tableau n'est pas vide, on considère l'élément médian du tableau, t.(m) où $m = \lfloor \frac{n-1}{2} \rfloor$.

- Si $e = t.(m)$ on a trouvé et on s'arrête.
- Si $e > t.(m)$ on se ramène à chercher dans le tableau de droite.
- Si $e < t.(m)$ On se ramène à chercher dans le tableau de gauche

En CAML On fait évoluer deux références «début» et «fin» correspondant aux indices de début et de fin du tableau extrait :

Code CAML :
```
>>let recherche e t =
     let début = ref 0
     and fin = ref (vect_length t -1) in
     let trouvé = ref false in
     while (!trouvé=false)
     &&(!début<=!fin) do
          let m=(!debut+!fin)/2 in
          if e=t.(m) then
               trouvé:=true
          else if e>t.(m) then
               début:=m+1
          else fin:=m-1
          done;
     !trouvé;;
```

Preuve de terminaison : Lors de l'exécution de la boucle, la valeur de fin-début déroit strictement et devient donc forcément négative à un moment ce qui garantit la terminaison de l'algorithme.
Soit $T(n)$ le temsp d'exécution del'algorithme sur un tableau de taille n. On a

$$T(n) = \mathcal{O}(1)+T(\frac{n}{2}) = \mathcal{O}(1)+\mathcal{O}(1)+T(\frac{n}{4})$$

$$= k\mathcal{O}(1) + T(\frac{n}{2^k}) = \mathcal{O}(\log_2(n)) = \mathcal{O}(\ln n)$$

3.3 Tri d'un tableau : Tri par sélection

On dispose en entrée d'un tebleau t, on veut réorodonner les éléments de t en un tableau t'.
Principe :

3	18	4	12	7	1	5
1	18	4	12	7	3	5
1	3	4	12	7	18	5
1	3	4	12	7	18	5
1	3	4	5	7	18	12
1	3	4	5	7	18	12
1	3	4	5	7	12	18

Code CAML :
```
>>let tri_par_selection t =
     let échange s a b =
          let u = s.(a) in
          (s.(a)<-s.(b);s.(b)<-u);
     in
     let le_plus_petit s a b =
          let resultat = ref a in
          for i=a+1 to b do
               if s.(i)<s.(!resultat) then
                    resultat:=i;
               done;
               !resultat
     in
     let n = vect_length t in
     let s = copy_vect t in
     for i = 0 to n-2 do
          let j = le_plus_petit s i (n-1) in
          échange s i j
          done;
     s;;
```

– Complexité de échange :

$$\mathcal{O}(1)$$

– Complexité de le_plus_petit :

$$\mathcal{O}(b-a)$$

– Complexité de tri :

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

3.4 Exercices

Exercice 3.4.1: *Test pour savoir si un tableau est trié*

```
>>let test t =
     let n =vect_length in
     let trié=ref true in
     let l = ref 0 in
     while (!trié )&&(!l<n-1) do
     if t.(!l)<t.(!l+1) then incr(l)
     else trié:=false
     done ;
     !trié;;
```

Si on a un tableau de longueur n où chaque case peu prendre une valeur entre 1 et n, la complexité moyenne est de $\sum_{k=0}^{n-2} \mathcal{O}(\frac{1}{k!})$

Soit t un tableau rempli de cases positives ou négative, déterminer a et b tels que $\sum_{t=a}^{b} t.(k)$ soit maximal. La complexité étant en $\mathcal{O}(n^2)$ ou mieux en $\mathcal{O}(n)$.

Exercice 3.4.2: *Somme maximale*

```
>>let somme t =
   let n = vect_length t in
   let j = ref 0 and  maks = ref 0 and megamaks = ref 0 in
   let a = ref 0 and b = ref 0 and c = ref 0 and d = ref 0 in
     while (t.(!j)<=0)&& (!j<n-1) do
       incr j;
     done;
     if (!j)=n-1 then
       (if t.(!j)<=0 then
             (a:=1;b:=0)
        else
             (a:=n-1;b:=n-1))
     else
       (megamaks:=t.(!j); maks:=t.(!j);
        a:=!j; c:=!j; d:=!j; b:=!j;
        for i = (!j)+1 to n-2 do
           if (!megamaks+t.(i))<=0 then
              (c:=i+1;d:=i+1;
               megamaks:=t.(!c))
           else
               d:=i;
           if t.(i)>0 then
              megamaks:=!megamaks +t.(i);
           if !megamaks>(!maks) then
              (a:=!c;b:=!d;maks:=!megamaks);
        done;
     if (t.(n-1)>0) && (!d=(n-2)) && ((!megamaks+t.(n-1))>(!maks)) then
           (a:=!c;b:=n-1);
     if t.(n-1)>(!maks) && !d<>n-2 then
           (a:=n-1;b:=n-1));
  (a,b);;
```

Exercice 3.4.3: *Ordre lexicographique sur des chaînes de caractères*

```
>>let compare s1 s2 =
    let n1=string_length s1
    and n2=string_length s2 in
    let j = ref 0 and continue=ref true in
     while !continue do
          if s1.[!j]<>s2.[!j] then
               continue:=false
          else
            (incr j;
            continue:=(!j<=n1)&&(!j<=n2));
          done;
     if !j=n1 then
          true
     else if !j=n2 then
          false
     else
          s1.[!j]<s2.[!j];
     ;;
```

(ie : tableaux de type char vect) On a la longueur de la chaine grâce à «string_length» et «s.[i]» donne le i_ème caractère.

CHAPITRE 4

Récursivité

4.1 Rappel

Soit $\mathcal{P}(n)$ une propriété portant sur l'entier $n \in \mathbb{N}$. On suppose que $\mathcal{P}(0), \mathcal{P}(1) \ldots, \mathcal{P}(n)$ sont vrais, et $\forall k > n, \mathcal{P}(0) \ldots \mathcal{P}(k-1)$ vrais implique $\mathcal{P}(k)$, alors $\mathcal{P}$ est vraie pour tout n.

Application en info : Soit P un programme agissant sur une donnée relative à un entier n. On suppose que si $n \leq r$ alors $P(n)$ termine et donne le bon résultat. Si $n > r$, $P(n)$ se ramène en temps fini à résoudre d'autres problèmes $P(k_1), P(k_2) \ldots, P(k_n)$ avec $k \leq n$ puis utilise les résultats pour résoudre le problème initial en temps fini, alors pout tout n le programme termine et donne le bon résultat.

Si $n > n_0$

- C'est le principe d'un algorithme récursif : à partir de la donnée n on crée un certain nombre de problèmes identiques portant sur des données plus petites
- On résout des problèmes plus petits en appelant la même procédure
- On exploite les résutlats pour el problème initial

Si $n < n_0$ on résout directement $P(n)$(Critère d'arrêt)

4.2 Récursivité en CAML

Pour construire une procédure récursive, on utilise la locution «rec».

Code CAML :

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

>>let rec pgcd a b =
     if  b = 0 then a
     else pgcd  b (a mod b);;
```

Code CAML : *Suite de Fibonacci*

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

Problème : si on lui demande fibo(9) il va calculer le terme fibo(8) deux fois, etc.
Complexité en temps d'exécution : Soit T(n) le temps d'exécution de fibo(n). T(n+2)=T(n+1)+T(n), donc T(n) suit la même relation de récurrence que fibo(n). on a donc $T(n) \sim fibo(n) \sim \left(\frac{1+\sqrt{5}}{2}\right)^n$
Ici un algorithme itératif calculant tous les termes et les stockant dans un tableau répondrait à la question en $\mathcal{O}(n)$.

Code CAML :

```
>>let rec util n = match n with
     0 -> (1,1)
    |_ -> let (a,b) = util(n-1) in (b,a+b)
          in let (a,b) = util n in a;;
```

«util(n)» retourne le couple fibo(n), fibo(n+1).

Code CAML : *Recherche récursive d'un élément dans un tableau trié*

```
>>let recherche e t =
   let rec rechbis e t a b =
        if b<a then false
        else let m = (a+b)/2 in
            if t.(m)=e then true
            else if t.(m)>e then
              rechbis e t a (m-1)
            else rechbis e t (m+1) b in
        rechbis e t 0 (vect_length t -1);;
```

4.3 Exercices

Exercice 4.3.1: *Suite récursive*

```
>>let rec suiterec n u0 f =
     if n = 0 then u0
     else f(suiterec (n-1) u0 f);;
```

Exercice 4.3.2: *Exponentiation rapide*

```
>>let rec exporapide a n =
     if n = 0 then 1
     else let q = n/2 and r = (n mod 2) in
          let u = exporapide (a*a) a in
              if r = 0 then u else u*a;;
```

Si T(n) désigne le temps d'exécution

$$T(n) = T(\tfrac{n}{2}) + \mathcal{O}(1) \Rightarrow T(n) = \mathcal{O}(\ln n)$$

Par contre la méthode naturelle est en $\mathcal{O}(n)$.

Pour les polynômes, sachant que le produit de deux polynômes de degrés d et d' s'exécute en un temps $\mathcal{O}(dd')$, la méthode naïve est en $\mathcal{O}(d^2)$ pour le calcul de p^2, $\mathcal{O}(2d^2)$ pour le calcul de $p^3 \ldots \mathcal{O}(n-1)d^2$ pour p^n on obtient donc un $\mathcal{O}(n^2d^2)$, alors qu'avec l'exponentiation rapide on obtient

$$T(n,d) = T(\frac{n}{2}, 2d) + \mathcal{O}(d^2) + \mathcal{O}(nd^2) = T(\frac{n}{4}) + \mathcal{O}(4d^2) + \mathcal{O}(2nd^2) = \cdots = T(1, nd) + \mathcal{O}(nd^2) + \mathcal{O}(n^2d^2)$$

donc en faisant la somme on obtient $\mathcal{O}(nd^2) + \mathcal{O}(n^2d^2)$ Moralité : cela ne sert à rien pour le polynômes.

Exercice 4.3.3: *Décomposition en facteurs premiers*

```
>>let decompostition n =
   let rec decomp n p =
        if p*p> n then
         (print_int n;print_newline())
        else if (n mod p)<> 0 then
         decomp n (p+1)
        else (print_int p;print_string "*";
              decomp(n/p) p)
   in decomp n 2;;
```

Exercice 4.3.4: *Anagrammes*

```
>>let anagram x =
    let rec ana a b =
      let n = string_length a in
        if n = 1 then
         (print_string (b^a);
         print_newline() )
        else (for i = 0 to n-1 do
ana (
(sub_string a 0 i)^
(sub_string a (i+1) (n-1-i)))
 (b^(sub_string a i 1)) ;
               done);
    in
    ana x "";;
```

Exercice 4.3.5: *Partitions d'un entier*

```
>>let partition n =
   let rec partbis n k =
        if (k=1) then 1
        else if n<k then partbis n n
        else if n=k then 1+partbis n k-1
        else
          partbis (n-k) k + partbis n (k-1)
   in partbis n n;;

>>let partprint n =
   let rec partbis n k a =
        if n = 0 then (print_string a;
          print_newline())
        else if (k=1) then
          partbis (n-1) k (a^"+1")
        else if n<k then partbis n n a
        else if n=k then
(print_string (a^"+"^(string_of_int k));
         print_newline();partbis n (k-1) a)
        else
(partbis (n-k) k (a^"+"^(string_of_int k));
           partbis n (k-1) a);
   in partbis n n "";;
```

Dans les partitions de n commençant par au moins k il y a celles qui commencent par k et celles qui commencent par quelque chose de plus petit, mais dans la dernière relation de récurrence il y a un problème si on obtient quelque chose de négatif ou nul. On peut remarquer que la complexité est égale au nombre de partitions du nombre ! Pour les afficher il faut en plus ajouter le préfixe dans la fonction «partbis»

CHAPITRE 5

Listes

5.1 Listes

Définition : Une liste est une *structure récursive*.
- Soit (arrêt) : la liste vide $\emptyset$
- soit un couple (a, l) où a est un élément du type $'t$ et l une liste.

Exemple : $(2,(3,(1,\emptyset)))$. C'est la liste $(2,3,1)$ c'est une liste du type «int_list».
La liste (a, l) est consituée de la tête a et de la queue l.

En Caml

Code CAML : *Définition directe*
```
>>let l=[2;3;1];;
   int list : [2;3;1]
```

Pour la tête de la liste la fonction «hd l;;» (mis pour «head»). Pour la queue de la liste «tl l;;» (mis pour «tail»). Pour obtenir le deuxième élément : «hd (tl l)».

L'avantage d'une liste par rapport à un tableau c'est que la liste est une structure dynamique. Pour rajouter un élément à une liste, on le rajoute en tête de liste :

Code CAML :
```
>>let l'=12::l ;;
```

On obtient laliste dont la tête est 12 etla queue est l

L'inconvénient d'une liste par rapport à un tableau c'est que l'on n'accède pas facilemet à un élément quelconque de la liste.

En Caml la gestion des listes utilise souvent la reconnaissance de motif :
- Soit l est du type liste vide [ ]
- Soit l est du type `i : :l`

Code CAML : *Fonction tête*
```
>>let tête l = match l with
     []    -> failwith("liste vide!")
    |t::q  -> t;;
```

La fonction queue se retrouverait en remplaçant t par q dans la dernière ligne.

Code CAML : *Pour obtenir la taille d'une liste*

```
>>let rec taille l = match l with
    [] -> 0
    |a::q ->1+taille q;;
```

Cette fonction s'appelle «list_length»

Code CAML : *Recherche d'un élément dans une liste*

```
>>let rec recherche e l = match l with
    [] -> false
    |t::q -> if e=t then true else
       recherche e q;;
```

Code CAML : *Concaténation*

```
>>let rec concat l1 l2 =
   match l1 with
   [] -> l2
   |t::q -> t::(concat q l2);;
```

Concaténer l_1 avec l_2 c'est scotcher l_1 devant l_2. le symbole «@» concatène deux listes.«l_1@l_2» Mais cette opération est un $\mathcal{O}(l_1)$.
Cette fonction est du type $alist- > alist- > alist$

Code CAML : *Tri par insertion*

```
>let rec insère e l = match l with
    [] -> [e]
    |t::q -> if e<=t then e::l
             else t::(insère e q);;

>let rec triparinsertion l = match l with
  [] -> []
  |e::q -> insère e (triparinsertion q);;
```

La procédure «insère» rajoute un élément dans une liste triée, à sa place
La compexité de l'insertion est au pire de $\mathcal{O}(n)$ et en moyenne de $\mathcal{O}(\frac{n}{2})$ (ce qui revient au même)
Pour le tri par insertion $T(n) = T(n-1) + \mathcal{O}(n)$ donc $T(n) = \mathcal{O}(1 + 2 + \cdots + n) = \mathcal{O}(n^2)$

5.2 Exercices

Exercice 5.2.1: *Fonction «map»*

```
>>let rec map f l = match l with
   [] -> []
   |t::q -> f(t)::(map f q);;
```

Cette fonction est du type $('a -> 'b) -> 'a\ list -> 'b\ list$

Exercice 5.2.2: *Somme des termes d'une liste*

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

Exercice 5.2.3: *Plus petit élément d'une liste*

```
>>let rec petit l = match l with
     []->failwith("liste vide !")
     |t::[] -> t
     |t::q -> let u = petit q in
              if t<=u then t else u;;
```

Exercice 5.2.4: *Test pour savoir si une liste est triée dans l'ordre croissant*

```
>>let rec ordrecroissant l = match l with
    [] -> true
    |a::[]-> true
    |a::b::q -> if a>b then false
                else ordrecroissant (b::q);;
```

Exercice 5.2.5: *Inversion d'une liste*

```
>>let inverse l =
    let l'=ref l
    and l''=ref [] in
    while !l'<>[] do
        l'':=(hd !l')::!l'';
        l':=tl(l');
        done;
        !l'';;

>>let rev l =
   let rec util l1 l2 =
      match l1 with
      [] -> l2
      |t::q -> util q (t::l2) in
   util l [];;
```

La deuxième fonction, récursive, est moins naturelle

Exercice 5.2.6: *Duplication d'une liste*

```
>>let rec duplique l = match l with
   [] -> []
   |t::q -> t::t::(duplique q);;
```

Exercice 5.2.7: *«Déduplication» d'une liste*

```
>>let rec dedup l = match l with
    []->[]
    |t::[]->l
    |a::b::q ->if a = b then
                   a::dedup q
               else
                   a::(dedup b::q);;
```

Exercice 5.2.8: *Têtes et queues de listes...*

```
>>let rec tq = match l with
   []->([],[])
   |t::q-> let (a,b)= tq q in
           match t with
           [] -> failwith("erreur")
           |c::[]->(c::a,b)
           |c::d->(c::a,d::b);;
```

Les tours de Hanoï

Exercice 5.2.9: *Les tours de Hanoï*

```
>>let rec hanoï n gauche milieu droite = match n with
    1-> (print_string (gauche^" "^"vers"^" "^droite);print_newline());
    |a->(hanoï (n-1) gauche droite milieu;
         print_string (gauche^" "^"vers"^" "^droite);print_newline();
         hanoï (n-1) milieu gauche droite);;

>hanoï 64 "riri" "fifi" "loulou";;
```

Le nombre de mouvements

$$T(n) = T(n-1) + 1 + T(n-1)$$

et

$$T(1) = 1 \Rightarrow 2^n - 1$$

CHAPITRE 6

Deux tris efficaces

6.1 Tri fusion

On considère[1] une liste l. On la coupe en deux : l_1 et l_2 d'environ même longueur. Récursivement, on trie l_1 et l_2. Ensuite on fusionne les résultats.

Exemple

$$[14, 5, 8, 18, 12, 24]$$

que l'on coupe en deux :

$$[14, 5, 8] \qquad [18, 12, 24]$$

on les trie :

$$[5, 8, 14] \qquad [12, 18, 24]$$

puis on les fusionne :

$$[5, 8, 12, 14, 13, 24]$$

Code CAML : *Tri fusion*

```
On commence par séparer les listes

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

Ensuite on fusionne les deux

>>let rec fusion l1 l2 = match (l1,l2) with
    ([],_)-> l2
    |(_,[])->l1
    |(a1::q1,a2::q2)-> if a1<a2 then
                            a1::(fusion q1 l2)
                       else
                            a2::(fusion l1 q2);;
```

[1] On peut aussi l'appeler mergesort

Code CAML : *Tri par fusion (suite)*

```
Il ne reste plus que le tri récursif

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

Complexité de l'algorithme On note $T(n)$ le temps nécessaire au tri ne n données[2]. Le temps nécessaire à la séparation est un $\mathcal{O}(n)$, la fusion quand à elle est aussi un $\mathcal{O}(n)$. Dès lors

$$T(n) = \mathcal{O}(n) + 2T(\frac{n}{2}) + \mathcal{O}(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

On s'intéresse au cas d'une puissance de 2 :

$$\begin{array}{lll} & T(2^n) & = & 2T(2^{n-1}) + \mathcal{O}(2^n) \\ ie & & \leq & 2T(2^{n-1}) + A2^n \\ ie & \dfrac{T(2^n)}{2^n} & \leq & \dfrac{T(2^{n-1})}{2^{n-1}} + A \\ \Rightarrow & \dfrac{T(2^n)}{2^n} - \dfrac{T(1)}{1} & \leq & An \\ ie : & T(2^n) & \leq & An2^n \end{array}$$

Pour n quelconque $2^{k-1} \leq n \leq 2^k \Leftrightarrow k-1 \leq \log_2(n) \leq k$. On a donc que

$$T(n) \leq T(2^k) \leq Ak2^k \leq 2A(\log_2(n)+1)n \in \mathcal{O}(n \ln n)$$

6.2 Tri rapide

C'est[3] le même principe, sauf en ce qui concerne la façon de séparer et fusionner les listes. On sépare les éléments de l en fonction d'un pivot.

$$\begin{array}{l} l_1 = \left\{x \in l \middle| x < pivot\right\} \\ l_2 = \left\{x \in l \middle| x \geq pivot\right\} \end{array}$$

La procédure de fusion est alors simplifiée.

Exemple

$$[14, 5, 12, 18, 8, 24]$$

On prend [4] le premier élément comme pivot :

$$[5, 12, 8] \qquad [12, 24]$$

et l'on trie et colle

$$[5, 8, 12, 15, 18, 24]$$

[2] Cela marche dès que l'on peut comparer les données
[3] On l'appelle aussi quicksort
[4] comme très souvent

Code CAML : *Tri rapide*

```
>>let rec separe l pivot = match l with
   []-> ([],[])
   |a::q -> let (l1,l2) = separe q pivot in
               if a < pivot then
                           (a::l1,l2)
               else
                           (l1,a::l2);;

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

Complexité La séparation est un $\mathcal{O}(n)$, la fusion est aussi un $\mathcal{O}(n)$.

$$T(n) = T(|l_1|) + T(|l_2|) + \mathcal{O}(n)$$

Dans le pire des cas (si la liste est triée...)

$$T(n) = T(n-1) + \mathcal{O}(n-1) \Rightarrow T(n) = \mathcal{O}(n^2)$$

C'est donc lent dans ce cas, mais en moyenne on va dire que

$$T(n) = T(\text{en moyenne } \frac{n}{2}) + T(\text{en moyenne } \frac{n}{2}) + \mathcal{O}(n) = \text{environ } \mathcal{O}(n \ln n)$$

Preuve rigoureuse : En fait, si on associe son rang à chaque élément, cela va dépendre du rang du pivot. On note K le rang du pivot, dès lors $|l_1| = K-1$ et $|l_2| = n-K$ Cependant la probabilité que le rang du pivot soit K est de $\frac{1}{n}$. Soit C_n l'espérance[5] de $T(n)$.

$$\begin{array}{lrcl}
 & C_n & = & \displaystyle\sum_{k=1}^{n} \mathbb{E}(T(n)|\text{rang du pivot } = K) \times \mathbb{P}(\text{rang du pivot } = K) \\
ie & & = & \displaystyle\frac{1}{n}\sum_{k=1}^{n}(C_{k-1} + C_{n-k} + \mathcal{O}(n)) \\
ie & C_n & = & \displaystyle\frac{2}{n}\sum_{k=0}^{n-1}(C_k + \mathcal{O}(n)) \\
\Rightarrow & nC_n & = & \displaystyle\left(2\sum_{k=0}^{n-1} C_k\right) + \mathcal{O}(n^2) \\
\Rightarrow & nC_n - (n-1)C_{n-1} & = & 2C_{n-1} + \mathcal{O}(n) \qquad \text{avec } \mathcal{O}(n^2) - \mathcal{O}(n-1)^2 = \mathcal{O}(n) \\
ie & C_n & = & \displaystyle\left(\frac{n+1}{n}\right) C_{n-1} + \mathcal{O}(1) \\
ie & \displaystyle\frac{C_n}{n+1} & = & \displaystyle\frac{C_{n-1}}{n} + \mathcal{O}\left(\frac{1}{n+1}\right) \\
ie & W_n - W_{n-1} & = & \displaystyle\mathcal{O}\left(\frac{1}{n+1}\right) \qquad \text{avec } W_n = \frac{C_n}{n} \\
ie & W_n & = & \mathcal{O}(\ln n) \\
ie & C_n & = & \mathcal{O}(n \ln n)
\end{array}$$

On utilise plutôt ce tri pour les listes, sauf bien sûr si l'on veut être certain d'avoir un tri en $n \ln n$.

[5] C'est à dire la complexité moyenne

6.3 Exercices

- Écrire fusion t a b c tel que :
  - Si t est trié entre les indices a et $b-1$
  - Si t est trié entre les indices b et $c-1$

  Alors en sortie t est trié entre les indices a et $c-1$. On fournit aussi en argument un tableau temporaire de même taille que t qui sert de mémoire d'appoint.
- Programmer trifusion récursif
- Trouver une variante itérative

Exercice 6.3.1: *Tri fusion d'un tableau*

```
>>let fusion t a b c temp =
   let i1 = ref a and i2 = ref b in
   for i = a to c-1 do
      if !i1=b then
                (temp.(i)<-t.(!i2);incr(i2))
      else if !i2=c then
                (temp(i)<-t.(!i1);incr(i2))
      else if t.(!i1)<=t.(!i2) then
                (temp.(i)<-t.(!i1);incr(i1))
      else
                (temp.(i)<-t.(!i2);incr(i2));
      done;
   for i = a to c-1 do
      t.(i)<-temp.(i);
   done;;

>>let trifusion t =
   let n = vect_legth t in
   let tt = copy_vect t in
   let temp = make_vect n t.(0) in
   let rec util tt temp a b =
      if (b-a) <= 1 then ()
      else let c = (a+b)/2 in
         (util tt temp a c;
          util tt temp c b;
          fusion tt a c b temp) in
   util tt temp 0 n;
   tt;;

>>
```

CHAPITRE 7

Diviser pour mieux règner

7.1 Diviser pour mieux règner

C'est un principe de conception de l'algorithme fonctionnant de la façon suivante : Soit P un problème à résoudre. On extrait à partir de P n problèmes $P_1, P_2 \ldots, P_n$ de tailles plus petite. On résout récursivement $P_1, P_2, \ldots, P_n$. On déduit des réponses la solution à P.

Exemples Les algorithmes de tri rapide et de tri fusion sont des algorithme qui divisent pour mieux règner : On coupe un liste en deux que l'on trie récursivement et l'on fusionne les résultats. On peut aussi citer la recherche dichotomique dans un tableau.

La stratégie peut être gagnante en terme de complexité (pour la recherche dans un tableau non trié on ne gagnerait rien ! Ce n'est donc pas toujours très intéressant.) La complexité de $T(n)$ vérifie souvent une récurrence de la forme : $T(n) = mT\left(\frac{n}{r}\right) + \mathcal{O}(f(n))$ où $m$est le nombre de sous problèmes, $\frac{n}{r}$ la taille des sous problèmes et $f(n)$ le temps nécessaire à la séparation et à la fusion des résultats.

Pour résoudre ces récurrences on travaille sur les $n = r^k$:

$$\begin{array}{rrcl} & T(r^k) & \leq & mT(r^{k-1}) + A \cdot f(r^k) \\ ie : & \frac{T(r^k)}{m^k} & \leq & \frac{T(r^{k-1})}{m^{k-1}} + A\frac{f(r^k)}{m^k} \\ ie : & \frac{T(r^k)}{m^k} & \leq & T(1) + A\sum_{j=1}^{k} \frac{f(r^j)}{m^j} \end{array}$$

Si $\sum_{j=1}^{+\infty} \frac{f(r^j)}{m^j}$ converge, on a $T(r^k) \leq Cm^k$. Or $n = r^k \Leftrightarrow k = \frac{\ln n}{\ln r}$ et donc $T(n) \leq Cm^{\frac{\ln n}{\ln r}} = Cn^{\left(\frac{\ln m}{\ln r}\right)}$ Par contre si la série ne converge pas on cherche un équivalent $g(k)$ de la série. et on a $T(r^k) \in \mathcal{O}(g(k)m^k)$ et

$$T(n) \in \mathcal{O}\left(g\left(\frac{\ln n}{\ln r} n^{\left(\frac{\ln m}{\ln r}\right)}\right)\right)$$

Exemples

- Dans le cas du tri fusion $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$ et $\frac{\ln m}{\ln r} = \frac{\ln 2}{\ln 2} = 1$ et la somme vaut donc $\sum_{j=1}^{k} \frac{f(2^j)}{2^j} = k$ Et $T(n) \in \mathcal{O}(\frac{\ln n}{\ln 2} n) = \mathcal{O}(n \ln n)$
- Produit de deux polynômes de degré (=taille) n :
  - On cherche à multiplier P et Q : On écrit $\left\{ \begin{array}{rcl} P & = & AX^{\frac{n}{2}} + B \\ Q & = & CX^{\frac{n}{2}} \end{array} \right.$ où A, B, C, D sont de taille $\frac{n}{2}$ on a alors
  $$PQ = ACX^n + (AD + BC)X^{\frac{n}{2}} + BD$$
  On calcule récursivement AC, AD, BC, BD, et on en déduit PQ.
  Complexité :
  $$T(n) = \underbrace{\mathcal{O}(n)}_{\text{séparation}} + 4T\left(\frac{n}{2}\right) + \mathcal{O}(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$
  Ainsi $r = 2$, $m = 4$, $f(n) = n$ on obtient : $\sum_{j=1}^{k} \frac{f(2^j)}{4^j} = \sum_{j=1}^{k} \frac{1}{2^j}$ converge.
  Donc $T(n) \in \mathcal{O}\left(n^{\left(\frac{\ln 4}{\ln 2}\right)}\right) = \mathcal{O}(n^2)$, bref cela ne sert à rien !
  - On fait la meêm séparation de P et Q en A, B, C, D. On calcule récursivement $(A+B)(C+D), AC, BD$, alors $AD + BC = (A + B)(C + D) - AC - BD$, la séparation reste en $\mathcal{O}(n)$, et la fusion des résultats est en $\mathcal{O}(n)$. $T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n)$ donc
  $$\sum \frac{f(r^j)}{3^j} = \sum \left(\frac{2}{3}\right)^j < +\infty$$
  et $T(n) \in \mathcal{O}\left(n^{\left(\frac{\ln 3}{\ln 2}\right)}\right) \sim \mathcal{O}(n^{1,6})$

7.2 exercices

On va implémenter cette technique de produit de polynômes. Un polynôme de degré n est codé sur un tableau de flottants de taille $n+1$. Si P est représenté par le tableau t on a $\sum_{k=0}^{n} t.(k)X^k$.

1. Coder la combinaison linéraire de deux polynômes.
2. Coder le produit normal de deux polynômes.
3. Coder le produit d'un polynôme par X^k.
4. Coder le produit récursif

Exercice 7.2.1: *Produit de deux polynômes*

```
>>let somme p q a b =
   let n_p = vect_length p and n_q = vect_length q in
   let r = make_vect (max n_p n_p) 0. in
   for i = 0 to n_p-1 do r.(i)<-r.(i)+.a*.p.(i); done;
   for i = 0 to n_q-1 do r.(i)<-r.(i)+.b*.q.(i);done;
   r;;

>>let produit p q a b =
   let n_p = vect_length p and n_q = vect_length q in
   let r = make_vect n_p+n_q-1 0. in
   for i = 0 to n_p-1 do
      for j = 0 to n_q-1 do
         r.(i+j)<-p.(i)*.q.(j);
      done;
   done;
   r;;

>>let decalage p k =
   let n_p=vect_length p and
   let r = make_vect (n_p+k) 0. in
   for i = 0 to n_p-1 do
      r.(i+k)<-p.(i);
   done;
   r;;
>>let separe p m =
   let n_p = vect_length p in
   if m>=n_p then ([| |],p)
   else
   (let b = make_vect m 0. and b = make_vect (n_p-m) 0. in
   for i = 0 to m-1 do
      b.(i)<-p.(i);
   done;
   for i = m to n_p-1 do a.(i-m)<-p.(i);
   done);
   (a,b);;            Où p=a*X^m+b
```

Exercice 7.2.2: *Produit de deux polynômes (suite)*

```
>>let rec prod p q =
   let n_p = vect_length p and n_q = vect_length q in
   let m = (max n_p n_q)/2 in
   if (n_p<=5) && (n_q<=5) then
      produit p q
   else
      let (a,b)=separe p m and (c,d)=separe q m in
      let p_1= prod a c and p_2 = prod b d in
      let p_3= prod (somme a b 1. 1.) (somme c d 1. 1.) in
      let p_4 = somme p_3 (somme p_1 p_2 1. 1.) 1. -1. in
      somme (decale p_1 (m+m)) (somme (decale p_4 m) p_2 1. 1.) 1. 1.;;
```

Exercice : Soit $T(n)$ la complexité du meilleur algorithme permettant de multiplier deux polynômes de degré n. Soit $U(n)$ la complexité du meilleur algorithme permettant de calculer P^2 si P est de degré n. Montrer que

$$(T(n) = \mathcal{O}(U(n))) \wedge (U(n) = \mathcal{O}(T(n)))$$

Ce que l'on note parfois $T(n) = \Theta(U(n))$.

Démonstration. Il suffit de calculer e produit $P \times P$ pour obtenir P^2. On en déduit que $U(n) \leq$ Complexité de cet algorithme $= T(n)$. Par ailleurs $P \times Q = \left(\frac{P+Q}{2}\right)^2 - \left(\frac{P-Q}{2}\right)^2$ donc $T(n) \leq 2U(n) +$ Complexité de l'addition + complexité de la multiplication $\leq 2U(n) + \mathcal{O}(n)$ Or $n \in \mathcal{O}(U(n))$ car le résultat de $U(n)$ est de taille $2n$ donc $T(n) \in \mathcal{O}(U(n))$ □

Exercice : Soit $f(n)$ croissante et positive strictement. $f(2n) \in \mathcal{O}(f(n))$. Montrer que $\exists\gamma$: $f(n) \in \mathcal{O}(n^\gamma)$ et que $\forall r > 0, f(rn) \in \mathcal{O}(f(n))$.

CHAPITRE 8

Tris

8.1 Complexité minimale

On a déjà vu des tris de complexité $\mathcal{O}(n^2)$: tri par insertion, tri par sélection...Des tris de complexité $\mathcal{O}(n \ln n)$: tri fusion, tri rapide...Ici on va tenter de prouver que l'on ne peut pas faire mieux que $\mathcal{O}(n \ln n)$. Pour donner du sens à cette affirmation, détaillons .

Un tri est dit comparatif s'il ne fait que comparer des éléments entre eux sans tenir compte de leur valeur. : il pourrait fonctionner dans tout ensemble totalement ordonné (c'est le cas de tous ceux que l'on a vu jusqu'ici). Exemple de tri non comparatif : on a 26 individus à trier selon une de leurs caractéristique $c(n)$ où les $c(k)$ sont exactement $[\![1, n]\!]$. Ici il suffit de mettre chaque individu à sa place : ce tri n'est pas comparatif : on utilise la valeur de $c(k)$ pour savoir où placer un élément à la fin mais on ne les compare pas entre eux , si jamais on n'avait pas d'individu lié à la valeur $c(k)$ on pourrait se passer des données...

Un tri comparatif n'agit qu'en fonction de l'ordre initial dans lequel sont fournies les données. Il fera les mêmes test sur les données «8 4 12» que sur les données «6 3 18». Chaque test effectué conduit à un résultat vrai ou faux (V ou F). Considérons, à un ordre donné σ sur les données initiales, la suite des réponses aux différents tests effectués par l'algorithme. La réponse de σ va être égale à une suite $rep(\sigma) = r_1, r_2, \ldots, r_m$ où m dépend de σ (avec $r_i \in \{V, F\}$). ($\sigma \in \mathfrak{S}_n$). La complexité du tri sachant qu'il doit traiter σ est m. (le nombre de tests). L'algorithme de tri doit gérer différemment deux ordres différents. $\sigma \mapsto rep(\sigma)$ doit être injective. La complexité moyenne s'écrit $C = \dfrac{1}{n!} \displaystyle\sum_{\sigma \in \mathfrak{S}_n} m(\sigma)$. Il y a 2^k réponses possibles telles que $m = k$. Donc $\#\{\sigma \in \mathfrak{S}_n | m(\sigma) = k\} \leq 2^k$ et $\#\{\sigma \in \mathfrak{S}_n | m(\sigma) \leq k\} < 2^{k+1}$ et on a donc que $C \geq \dfrac{1}{n!} \displaystyle\sum_{\substack{\sigma \in \mathfrak{S}_n \\ m(\sigma) \geq k}} k \geq \dfrac{k}{n!}(n! - 2^k)$ soit k tel que $\dfrac{n!}{4} \leq 2^k \leq \dfrac{n!}{2}$ $(k = \lfloor \log_2 \dfrac{n!}{2} \rfloor)$

$C \geq \dfrac{k}{n!}(n! - \dfrac{n!}{2}) \geq \dfrac{k}{2} \geq \dfrac{1}{2}(\log_2 n! - 2)$ Or $n! \sim \sqrt{2\pi n}\left(\dfrac{n}{e}\right)^n \Rightarrow \log_2 n! \sim n \log_2 n$

Donc la complexité moyenne $C \geq$ quelques chose $\sim \dfrac{n \log_2 n}{2}$ et $n \ln n \in \mathcal{O}(C)$

8.2 Exercice

8.2.1 Les tri selon les pieds nickelés

1. Le tri ne fonctionnera que si le tiers du milieu est plus grand au sens large que le dernier tiers.

   Démonstration.

   $\boxed{\Leftarrow}$ Considérons que les éléments du tableau sot de trois types :
   - Type 1 : Devant aller dans le premier tableau.
   - Type 2 : Devant aller dans le deuxième tableau.
   - Type 3 : devant aller dans le dernier tableau.

   A l'issue de la première étape les éléments de type 3 présents dans les deuxpremiers tiers se retrouvent à droite (dans le tiers du milieu). Comme le tiers du milieu est plus grand que celui de droite, il contient tous les éléments de type 3 qui étaient dans le premier tiers. A l'étape 2, tous les éléments de type 3 sont dans le troisième tiers, ils doivent être mis à leur place dans le dernier tiers. Il ne reste dans les deux premiers tiers que des éléments de types 1 et 2. Ils seront donc triés à l'étape trois.

   $\boxed{\Rightarrow}$ Supposons le tiers du mileu plus petit que le tiers de droite. Supposons qu'initialement tous les éléments de type 3 sont vers la gauche du tableau (Par exemple le tableau est trié dans l'ordre décroissant). Après l'étape 1 il reste dans le tiers de gauche des éléments de type 3, puisqu'ils ne rentrent pas tous dans le tiers du milieu et qu'il y en avait au départ de quoi remplir le tiers du milieu et en mettre un de plus dans le tiers de gauche[1]. Ces éléments ne bougeront pas après l'étape deux et donc resteront dans les deux premiers tiers après l'étape 3 au lieu de se fixer dans le tiers de droite.

   □

Exercice 8.2.1: *Le tri des pieds nickelés*

2.
```
>>let tri_des_pieds_nickelés t =
   let rec tripar3 t i j =
     match (i-j) with
     1-> if t.(i)>t.(j) then
           (let u = t.(i) in
           t.(i)<-t.(j);
           t.(j)<-u)
    |0->()
    |_->let m = (j-i+1)/3 in (tripar3 t i (j-m);
                              tripar3 t (i+m) j;
                              tripar3 t i (j-m));
    in tripar3 t 0 (vect_length t -1);;
```

3. On a comme complexité :

$$T(n) = 3T\left(\frac{3}{2}n\right) + \mathcal{O}(1)$$

et donc $T(n) \in \mathcal{O}\left(n^{\left(\frac{\ln 3}{\ln \frac{3}{2}}\right)}\right) \simeq \mathcal{O}(n^{2,7})$

[1] qui n'est pas vide...

8.2.2 Tri par paquets

Exercice 8.2.2: *Tri par paquets*

1.
```
>>let triparpaquets t =
   let rec insere e l = match l with
      [] -> [e]
     |a::q -> if e <= a then e::l
              else a::(insere e q)
   in
   let n = vect_length t in
   let nn = float_of_int n in
   let paq = make_vect n [] in
   for i = 0 to pred(n) do
      let m = int_of_float (nn*.t.(i)) in
      paq.(m)<- insere t.(i) paq.(m)
      done;
   let l = ref [] in
      for i = pred(n) downto 0 do
         l:=paq.(i)@(!l);          On peut utiliser vect_of_list !l
         done;
   !l;;
```

2. La complexité est

$$T(n) = \underbrace{\mathcal{O}(n)}_{\text{Parcours de tous les éléments}} + \underbrace{\mathcal{O}(n)}_{\text{Concaténation}} + \sum_{k=0}^{n-1} \mathcal{O}\left((\text{longueur } paq.(k))^2\right)$$

$$\begin{aligned} \mathbb{E}(T(n)) &= \mathcal{O}(n) + \mathcal{O}\left(\sum_{k=0}^{n-1} \underbrace{\mathbb{E}((long.paq.(k))^2)}_{\text{Ne dépend pas de k}}\right) \\ ie: &= \mathcal{O}(n) + \mathcal{O}(n\mathbb{E}((long.paq.(0))^2)) \end{aligned}$$

Or $\mathbb{E}((long.paq.(0))^2) = \sum_{k=0}^{n} \mathbb{P}(\text{longueur} = k) \times k^2$, mais

$$\mathbb{P}(\text{longueur } = k) = \binom{n}{k}\left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k} \leq \binom{n}{k}\frac{1}{n^k} = \frac{1}{k!}\frac{n(n-1)\dots(n-k+1)}{n\times n\times \dots n} \leq \frac{1}{k!}$$

et $\mathbb{E}((long.paq.(0))^2) \leq \sum_{k=0}^{n} \frac{k^2}{k!} \leq \sum_{k=0}^{+\infty} \frac{k^2}{k!} = C$ et au final $\mathbb{E}(T(n)) \leq \mathcal{O}(n) + \mathcal{O}(Cn) = \mathcal{O}(n)$

CHAPITRE 9

Programmation dynamique

C'est une technique qui consiste éventuellement à rajouter des arguments au problème puis à résoudre successivement des problèmes de plus en plus gros, jusqu'à arriver au problème final. Il s'agit d'une alternive aux solutions récursives, parfois intéressante lorsqu'il s'agit de problèmes d'optimisation.

9.1 La location de skis

Un loueur de skis possède r clients de tailles respectives $t_1, \ldots, t_r$. Il dispose aussi de p paires de skis de tailles $s_1, \ldots, s_p$ $(p \geq r)$. Il cherche l'injection $\{\text{clients}\} \to \{\text{skis}\}$ c'est à dire de $\sigma : \mathbb{N}_r \to \mathbb{N}_p$ telle que l'inconfort $\sum_{k=1}^{r} |t\_s_{\sigma(k)}|$ soit minimal.

1. On suppose que les t_i et les s_i sont triés. Montrer que l'injection σ optimale est croissante.
2. Envisager un algorithme récursif pour résoudre le problème. Évaluer sa complexité.
3. Résoudre le problème par programmation dynamique. On considèrera les sous-problèmes : «affecter aux clients $[\![1, r']\!]$ des skis dans $[\![1, p']\!]$ lorsque $r' \leq r$ et $p' \leq p$. Évaluer la complexité.

Solutions

1.
2. Appelons $Opt(r', p')$ la fonction qui calcule l'injection optimale de $[\![1, r']\!] \to [\![1, p']\!]$ et qui renvoie le couple (σ, sc) où σ est l'injection et sc l'inconfort minimal : $\sum_{k=1}^{r'} |t_k - s_{\sigma(k)}|$ sachant que $\forall k, \sigma(k) \leq p'$.
   - Si $r' = 1$, on cherche le ski s_i $(i \leq p')$ le plus proche de la personne 1.
   - Si $r' > 1$, pour k compris entre r' et p', on essaye de donner à r' le ski n^o k. On calcule donc $(\sigma_k, sc_k) = Opt(r'-1, k-1)$. Puis on cherche un k tel que $sc_k + |t_{r'} - s_k|$ soit minimal. On renvoie alors $(\tilde{\sigma} = \sigma_k \text{ et } r' \to k,\ sc_k + |t_{r'} - s_k|)$
   - Complexité : $T(r, p) = \sum_{k=r}^{p} T(r-1, k-1) + \mathcal{O}(r, p)$ Or $\binom{p}{r} = \sum_{k=r}^{p} \binom{k-1}{r-1}$. On en déduit que $T(r, p) \geq C\binom{p}{r}$
3. On construit progressivement un tableau C à double entrée (r', p') où $1 \leq r' \leq r$ et$1 \leq p' \leq p$. Dans chaque case il y aura (σ, sc) correspondant à $Opt(r', p')$. On calcule d'abord tous les

$C(1,k)$ pour $k \in \mathbb{N}_p$ (en $\mathcal{O}(p)$), puis on remplit itérativement les lignes de $r'=2$ jusqu'à r avec $\forall p' \geq r', C(r',p') = (\sigma, sc)$ correspondant à $\min_{r'\leq k\leq p}(C(r'-1,k-1)+|t_{r'}-s_k|)$ ce qui se fait en $\mathcal{O}(rp^2)$. À la fin la case qui nous intéresse n'est autre que $C(r,p)$.

9.2 Le problème du gymnase

Vous êtes le gestionnaire d'un gymnase ouvert de 8h à 20h. Il y a dansla ville n clubs sportifs qui désirent occuper le gymnase dans le créneau $[a_k, b_k[$. Trouver une ensemble de créneaux disjoints qui maximise l'occupation totale, ie : trouver $I \subset \mathbb{N}_n$ tel que $\forall (i,j) \in I, [a_i,b_i[\cap[a_j,b_j[= \emptyset$ et $\sum_I (b_i - a_i)$ soit maximale.

Question associée : on suppose que $\bigcup_i [a_i,b_i[= [8h, 20h[$. Montrer que dans ce cas la solution optimale vérifie $\sum_i (b_i - a_i) > 6h$

On suppose que les b_i sont triés dans l'ordre croissant. On construit un tableau $Opt(1\dots n)$ où $Opt(k)$ représente l'occupation optimale du créneau $[8h, b_k[$ par les événements $([a_i,b_i[)_{i\in\mathbb{N}_k}$ (une liste σ des indices des créneaux, un entier ou un flottant s qui est le score d'occupation).

$Opt(1) = ([\ 1\ ], b_1 - a_1)$. Supposons $Opt(1)\dots, Opt(k-1)$ calculés. Le recouvrement $Opt(k)$ si l'événement k n'y est pas, est $Opt(k-1)$. Si l'événement $[a_k,b_k[$ est dans le recouvrement optimal, le reste du recouvrement est un recouvrement optimal de $[0,a_k[$. C'est $Opt(r)$ où r est le plus grand entier tel que $b_r \leq a_k$. On compare donc $Opt(k-1)$ avec $(b_k - a_k) + Opt(r)$ et on prend le plus grand des deux : s.

- Si $s = Opt(k-1)$, $\quad Opt(k) = (\sigma_{k-1}, s_{k-1})$.
- Si $s = (b_k - a_k) + Opt(r)$, $\quad Opt(k) = (k :: \sigma_r, s)$

La recherche de s s'effectue en $\mathcal{O}(\ln n)$ car c'est une recherche dichotomique dans le tableau des b_k La complexité générale est donc $n\mathcal{O}(\ln n)$

Montrons que de tout recouvrement complet on peut extraire un sous-recouvrement disjoint de longueur supérieure à la moitié de l'intervalle de disponibilité grâce à un algorithme récursif. Pour chaque créneau $[a_k,b_k]$, l'enlever de la liste des créneaux disponibles si jamais il est inclus dans un autre. (ce qui se fait en $\mathcal{O}(n^2)$) Les créneaux qui restent ne sont inclus dans aucun autre. On les ordonne $\mathcal{O}(n\ln n)$, puis on enlève les c_k tels que $c_k \subset (c_{k-1} \cup c_{k+1})$, *ie* : $a_{k+1} \leq b_{k-1}$. On se retrouve avec un ensemble de créneaux qui recouvrent encore tout. Les sous recouvrements $c_2, c_4, \dots$ et $c_1, c_3, \dots$ sont disjoints et l'un des deux est supérieur à 6 heures. Il n'y a pas toujours mieux.

CHAPITRE 10

Logique propositionnelle

10.1 Logique propositionnelle

Une proposition est une phrase à laquelle on peut associer un statut ou faux (V ou F).

Exemple :
- $2 + 2 = 5$ est une proposition.
- $x + y = 3$ est une proposition
- «Tous les éléphants roses ont 5 pattes» (c'est vrai !)

On peut combiner des proposition pour en faire d'autres. On dispose pour cela de 4 opérateurs principaux :

La négation $\neg p$ est vrai si et seulement si p est faux

La conjonction $a \wedge b$ est vrai dès que a et b sont simultanéments vrais.

La disjonction $a \vee b$ est vrai dès que l'une des deux propositions est vraie.

L'implication $a \Rightarrow b$ est vrai si quand a est vrai alors b est vrai.

A partir de ces opérateurs on va créer une classe de formules logiques. C'est un ensemble défini récursivement par :
- Soit la formule est un *littéral* id est une variable booléenne, du genre $P_1, P_2, \ldots$.
- Soit la formule s'écrit $\neg f$ ou $f_1 \wedge f_2$, ou $f_1 \vee f_2$ ou $f_1 \Longrightarrow f_2$ où f, f_1, f_2 sont des formules.

Exemple :

$$((p_1 \wedge p_2) \vee (p_3 \wedge p_1)) \Longrightarrow (p_1 \Longrightarrow p_3) \qquad (f)$$

Une formule peut être évaluée lorsque l'on donne des valeurs booléennes (V,F) aux variables propositionnelles. Par exemple pour f : si $p_1 = V, p_2 = F$ et $p_3 = V$ alors $f = V$.

Une formule est *satisfiable*, s'il existe une instanciation des variables propositionnelles qui évalue f en V. Si elle n'est pas satisfiable on dit que f est une *contradiction*. Si f est toujours satisfaite indépendamment des valeurs de p_k, c'est une *tautologie*. On a que f est satisfiable si et seulement si $\neg f$ n'est pas une tautologie. C'est un problème difficile en informatique.

Exemples :
- $p_1 \wedge (\neg p_1)$ est une contradiction.
- $p_1 \wedge (p_2 \vee (\neg p_1))$ est satisfiable (avec p_1 et p_2 vrais.)
- $p_1 \Rightarrow p_1$ est une tautologie.

Une formule f induit une fonction booléenne : si f contient n variables propositionnelles $p_1, p_2, \ldots, p_n$, la formule induit une fonction de $\{V, F\}^n \longrightarrow \{V, F\}$. On dit que deux formules f_1 et f_2 sont équivalentes si elles induisent la même fonction booléenne. On note $f_1 \equiv f_2$.

Proposition 10.1 []

Si
$$f(p_1, p_2 \ldots, p_n) \equiv g(p_1, p_2, \ldots, p_n)$$
et que $f_1, \ldots, f_n$ sont des formules alors
$$f(f_1, \ldots, f_n) \equiv g(f_1, \ldots, f_n)$$

Exemple : $\neg(p_1 \vee p_2) \equiv \neg p_1 \wedge \neg p_2$ et donc $\neg((p_1 \Rightarrow p_2) \vee (p_3 \Rightarrow p_1)) \equiv (\neg(p_1 \Rightarrow p_2) \wedge \neg(p_3 \Rightarrow p_1))$

Règles de calcul propositionnel :

$$\begin{array}{rcl}
\neg(p_1 \wedge p_2) & \equiv & \neg p_1 \vee \neg p_2 \\
\neg(p_1 \vee p_2) & \equiv & \neg p_1 \wedge \neg p_2 \\
\neg(\neg p_1) & \equiv & p_1 \\
\neg(p_1 \Rightarrow p_2) & \equiv & p_1 \wedge \neg p_2 \\
p_1 \wedge (p_2 \wedge p_3) & \equiv & (p_1 \wedge p_2) \wedge p_3 \\
p_1 \vee (p_2 \vee p_3) & \equiv & (p_1 \vee p_2) \vee p_3 \\
p_1 \vee (p_2 \wedge p_3) & \equiv & (p_1 \vee p_2) \wedge (p_1 \vee p_3) \\
p_1 \wedge (p_2 \vee p_3) & \equiv & (p_1 \wedge p_2) \vee (p_1 \wedge p_3)
\end{array}$$

Pour étudier la satisfaisabilité ou léquivalence de deux formules on peut dresser une *table de vérité* : pour chaque valeur des variables on calcule f.

Exemple :
$$f = (p_1 \Rightarrow p_2) \vee (p_2 \Rightarrow p_3) \vee (p_3 \Rightarrow p_1)$$

On dresse la table de vérité :

P_1	P_2	P_3	f
V	V	V	V
V	V	F	V
V	F	V	V
V	F	F	V
F	V	V	V
F	V	F	V
F	F	V	V
F	F	F	V

f est une tautologie.

Démontrons la même proposition par calcul formel :

$$\begin{array}{rcl}
\neg f & \equiv & \neg(p_1 \Rightarrow p_3) \wedge \neg(p_2 \Rightarrow p_3) \wedge \neg(p_3 \Rightarrow p_1) \\
ie & \equiv & (p_1 \wedge \neg p_2) \wedge (p_2 \wedge \neg p_3) \wedge (p_3 \wedge \neg p_1) \\
ie : & \equiv & (p_1 \wedge \neg p_1) \wedge (p_2 \wedge \neg p_2) \wedge (p_3 \wedge \neg p_3)
\end{array}$$

10.2 Exercices

Soit n impair supérieur ou égal à 3. Quelles sont les valeurs de $p_1, p_2, \ldots, p_n$ qui satisfont :

$$f = (p_1 \Rightarrow p_2) \wedge (p_3 \Rightarrow p_4) \wedge \cdots \wedge (p_n \Rightarrow \neg p_1) \wedge (\neg p_2 \Rightarrow \neg p_3) \wedge \cdots \wedge (\neg p_{n-1} \Rightarrow \neg p_n)$$

$$f \equiv V \Leftrightarrow \begin{cases} \forall k \text{ impair } \neq 1, n \quad p_k \equiv V \Rightarrow p_{k-1} \text{ et} p_{k+1} \text{ sont vrais} \\ p_1 \text{ vrai} \Rightarrow p_2 \text{ vrai et } p_n \text{ fausse} \\ \text{ et } p_n \text{ vraie} \Rightarrow p_1 \equiv F \text{ et } p_{n-1} \equiv V \end{cases}$$

Ce résultat n'est pas très satisfaisant mais c'est déjà pas mal...

10.3 Le mystère du pharaon

1. on peut dire que

$$(A_1 \wedge A_2 \wedge \neg A_3) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge A_2 \wedge \neg A_3)$$

2. La traduction de l'égyptien donne : $A_1 \equiv B \Rightarrow N$, $A_2 \equiv B \wedge N$ et $A_3 \equiv N$
3. $A_1 \wedge A_2 \wedge \neg A_3 \equiv (B \Rightarrow N) \wedge B \wedge N \wedge \neg N \equiv F$ essayons l'autre :

$$\begin{array}{rrl} & A_1 \wedge \neg A_2 \wedge A_3 & \equiv (B \Rightarrow N) \wedge (\neg B \vee \neg N) \wedge N \\ ie & & \equiv (B \Rightarrow N) \wedge (N \vee \neg N) \vee (N \wedge \neg B) \\ ie & & \equiv (B \Rightarrow N) \wedge (N \wedge \neg B) \\ ie & & \equiv N \wedge \neg B \end{array}$$

   et enfin :

$$\neg A_1 \wedge A_2 \wedge A_3 \equiv B \wedge \neg N \wedge B \wedge N \wedge N \equiv F$$

   donc il y a bien une seule solution !

4. Donc : $A_1 \equiv \neg(P \Rightarrow G)$ et $A_2 \equiv \neg M \wedge G$ et enfin $B_3 \equiv (\neg G \wedge \neg G) \vee (\neg G \wedge \neg M) \vee (\neg G \wedge \neg P)$ On a donc $A_3 \Rightarrow B_3$, même si A_3 est inconnue ! On considère le système obtenu en remplaçant A_3 par B_3 dans le texte, mais on ne peut en conclure que le fait que le premier système soit satisfait implique le deuxième soit satisfait, mais $A_1 \equiv P \wedge \neg G$ donc $A_1 \wedge A_2 \equiv P \wedge \neg G \wedge \neg M$ qui n'est pas satisfiable. Donc $A_1 \wedge A_2 \wedge \neg A_3$ et $A_1 \wedge A_2 \wedge \neg B_3$ ne sont jamais satisfaites, mais le fait que le premier système soit satisfait implique que A_3 soit satisfaite et doncque B_3 le soit aussi et donc qu le scond système le soit aussi. Or $A_1 \equiv P \wedge \neg G$ $A_2 \equiv \neg M \wedge G$ et $B_3 \equiv \neg G$, donc

$$\begin{array}{rrl} & A_1 \wedge \neg A_2 \wedge B_3 & \equiv P \wedge \neg G \wedge (M \vee \neg G) \wedge \neg G \\ ie : & & \equiv \neg G \wedge P \end{array}$$

   et

$$A_1 \wedge A_2 \wedge A_3 \equiv (G \wedge \neg P) \wedge \neg G \wedge \neg M \wedge G \equiv F$$

   Comme on a deux solutions qui satisfont l'énoncé, ce n'est pas vrai , il faut modifier l'énoncé : les «...»ne sont pas l'un des trois pavés : on ne peut décider lequel est vrai.

10.4 Formes normales d'une formule

À équivalence près, on peut transformer une formule en une conjonction de disjonctions ou en une disjonction de conjonctions.

Exemple :

$$(P_1 \vee P_3 \vee \neg P_4) \wedge (\neg P_1 \vee P_2 \vee P_3) \wedge (P_1 \vee \neg P_4)$$

qui est une forme normale conjonctive et

$$(P_1 \wedge \neg P_3 \wedge P_4) \vee (P_2 \wedge \neg P_3 \wedge P_4) \vee (P_1 \wedge \neg P_2)$$

qui est une forme normale disjonctive.

Preuve de l'existence d'une forme normale disjonctive : Soit F une forme portant sur les variables $P_1, \ldots, P_n$. Soit

$$\mathcal{A} = \left\{(x_1, \ldots, x_n) \in \{V, F\}^n \middle| F(x_1, \ldots, x_n) \text{ vraie}\right\} = F^{-1}(V)$$

À chaque $(x_1, \ldots, x_n) \in \mathcal{A}$ on a une clause conjonctive :

$$C(x_1, \ldots, x_n) = \left(\bigwedge_{x_i = V} P_i\right) \wedge \left(\bigwedge_{x_i = F} \neg P_i\right)$$

qui est satisfaite si et seulement si $\forall i, P_i = x_i$.

On a alors $F = \bigwedge_{(x_1,\ldots,x_n)\in\mathcal{A}} C(x_1, \ldots, x_n)$.

Exemple :

$$F = (P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_3)$$

Table de vérité :

P_1	P_2	P_3	F
V	V	V	V
V	V	F	F
V	F	V	F
V	F	F	F
F	V	V	V
F	V	F	F
F	F	V	V
F	F	F	V

on a donc

$$F \equiv \begin{array}{ll} & (P_1 \wedge P_2 \wedge P_3) \\ \vee & (\neg P_1 \wedge P_2 \wedge P_3) \\ \vee & (\neg P_1 \wedge \neg P_2 \wedge P_3) \\ \vee & (\neg P_1 \wedge \neg P_2 \wedge \neg P_3) \end{array}$$

Deuxième preuve : Par *induction structurelle* (récurrence sur la structure)

Rappelons que la définition d'un formule est récursive :

- Soit un littéral
- Soit non une formule
- Soit une conjonction de formules
- Soit une disjonction de formules

– Soit une implication de formules.

initialisation Tout littéral est lui même une forme normale disjonctive.

Propagation Soit F_1 et F_2 tels que $F_1 \equiv C_1 \vee C_2 \vee \ldots \vee C_p$ et $F_2 \equiv D_1 \vee D_2 \ldots \vee D_m$ où les C_i, D_i sont des conjonctions. Alors :

$$F_1 \vee F_2 \equiv C_1 \vee C_2 \vee \ldots \vee C_p \vee D_1 \vee \ldots \vee D_m \quad \square$$

$$F_1 \wedge F_2 \equiv (C_1 \vee C_2 \ldots \vee C_p) \wedge (D_1 \vee \ldots \vee D_m) \equiv \bigvee_{(i,j)} (\underbrace{C_i \wedge D_j}_{\text{Conjonction}}) \quad \square$$

et

$$\neg F_1 \equiv \bigwedge_{i=1}^{p} (\neg C_i)$$

et $\neg C_i$ est une disjonction de littéraux. Distributivité : $\bigwedge_{i=1}^{p} (\neg C_i)$ est une disjonction de conjonctions. Et

$$F_1 \Rightarrow F_2 \equiv F_1 \vee \neg F_2$$

et l'on se rapporte donc aux cas précédents.

Par induction structurelle toute formule est équivalente à une disjonction de conjonctions de littéraux.

Exemple :

$$\begin{aligned} P_1 \Rightarrow (P_2 \vee P_3) \equiv (\neg P_1) \vee (P_2 \wedge P_3) &\equiv \neg P_1 \vee P_2 \vee P_3 \\ &\equiv (\neg P_1 \wedge P_2) \wedge ((\neg P_3 \wedge P_1) \vee (\neg P_3 \wedge P_3)) \\ &\equiv (\neg P_1 \wedge P_2) \wedge ((\neg P_3 \wedge P_1) \vee F) \\ (\neg(P_1 \vee \neg P_2)) \wedge (\neg P_3 \wedge (P_1 \vee P_3)) &\equiv (\neg P_1 \wedge P_2) \wedge (\neg P_3 \wedge P_1) \\ &\equiv \neg P_1 \wedge P_2 \wedge \neg P_3 \wedge P_1 \\ &\equiv F \end{aligned}$$

En moyenne on peut représenter une formule de n variables par une conjonction de 2^n termes...Ce n'est donc aps toujours très intéressant en moyenne !

Proposition 10.2 []

Toute formule admet une forme normale conjonctive :

$$F \equiv (\ldots \vee \ldots \vee \ldots) \wedge (\ldots \vee \ldots \vee \ldots) \wedge \ldots$$

Démonstration.

$$F \equiv \neg(\neg F)$$

On prend une forme normale disjonctive de $\neg F$:

$$C_1 \vee C_2 \vee \ldots \vee C_n$$

où tous les C_i sont des conjonctions de littéraux.

$$F \equiv \neg(C_1 \vee \ldots \vee C_n) \equiv (\neg C_1 \wedge \ldots \wedge \neg C_n)$$

où les $\neg C_i$ sont des disjonctions de littéraux. $\square$

Exemple

$$P_1 \Rightarrow P_2 \equiv \neg(P_1 \wedge \neg P_2) \equiv \underbrace{(\neg P_1 \vee P_2)}_{\text{forme normale conjonctive}} \equiv \underbrace{(\neg P_1) \vee (P_2)}_{\text{forme normale disjonctive}}$$

Circuits logiques élémentaires : [1] Les transistors sont des circuits logiques élémentaires : $\vee, \wedge$ et $\neg$. Il y a aussi dans le circuit des sources V ou F. Exemple[2] de circuit réalisant

$$S = (E_1 \wedge E_2) \vee (E_1 \wedge \neg E_3)$$

Exemple : Additionneur 2-bit $S_3S_2S_1 = E_2E_1 + E_2'E_1'$ en base 2.

$$S_1 \equiv (E_1 \wedge \neg E_1') \vee (\neg E_1 \wedge E_1')$$

et

$$S_2 \equiv complique\_de(E_1, E_2, E_1', E_2')$$

pareil pour E_3.

Pour simplifier on construit un nouveau circuit logique. [3] Où + réalise l'addition 1-bit chiffre par chiffre. $C = A + B$ et $D =$ retenue de $(A + B)$.

[1] Bon là il faut imaginer une schéma de puce électronique avec plein de pattes schématisant les entrées et d'autant de pattes pour les sorties...Avec des transistors élémentaires qui réalisent une formule logique élémentaire entre deux entrées. Reste le problème d'optimisation du coût...

[2] Il faut toujours imaginer un circuit...

[3] un schéma...

10.5 Exercice : Problème logique et automate

1. La porte s'ouvrira si les positions des leviers satisfont :

$$R = (E_1 \wedge E_2 \wedge E_3) \vee (\neg E_1 \wedge \neg E_2 \wedge \neg E_3)$$

2. On a :

$$E_1 \equiv \neg(L_2 \wedge \neg L_1 \wedge \neg L_3) \vee (L_1 \wedge L_2 \wedge L_3)$$

$$E_2 \equiv (L_3 \vee (\neg L_1 \wedge \neg L_2)) \Rightarrow (L_3 \wedge (L_1 \vee L_2))$$

$$E_3 \equiv (L_1 \Rightarrow L_3) \wedge (L_2 \Rightarrow L_1)$$

3. On obtient :

$$E_1 \equiv (L_1 \vee \neg L_2 \vee L_3) \wedge (\neg L_1 \vee \neg L_2 \vee \neg L_3)$$

$$\begin{array}{rcl} E_2 & \equiv & \neg(L_3 \vee (\neg L_1 \wedge \neg L_2)) \vee (L_3 \wedge L_1 \wedge (L_1 \wedge L_2)) \\ ie & \equiv & (\neg L_3 \wedge (L_1 \vee L_2)) \vee (L_3 \wedge (L_1 \vee L_2)) \\ ie & \equiv & L_1 \vee L_2 \end{array}$$

$$E_3 \equiv (\neg L_1 \vee L_3) \wedge (\neg L_2 \vee L_1)$$

et donc :

$$\begin{array}{rcl} E_1 \wedge E_2 \wedge E_3 & \equiv & (L_1 \vee \neg L_2 \vee L_3) \wedge (\neg L_1 \vee \neg L_2 \wedge \neg L_3) \wedge (L_1 \vee L_2) \wedge (\neg L_1 \vee L_3) \wedge (\neg L_2 \wedge L_1) \\ ie & \equiv & L_1 \wedge (\neg L_1 \vee L_3) \wedge (L_1 \vee \neg L_2 \vee L_3) \wedge (\neg L_1 \vee \neg L_2 \vee \neg L_3) \\ ie : & \equiv & L_1 \wedge L_3 \wedge \neg L_2 \end{array}$$

10.6 Concours centrale supélec 1999

10.6.1 Théorie

1. Elle est satisfiable si, et seulement si, elle contient au moins une clause satisfiable. Toutes les clauses duales ne contenant pas $p \wedge \neg p$ sont satisfiables.
2. Soient A, B, C des formules propositionnelles.
   – Considérons la valuation v telle que $v(V) = \bar{1}$ et $v(F) = \bar{0}$

   $$v(A \oplus B) = v(A) + v(B) \qquad v(A \wedge B) = v(A) \times v(B)$$

   Comme l'addition est associative et commutative dans $\mathbb{F}_2$, et que le produit est distributif sur l'addition, on a :

   $$v(A \oplus B) = v(A) + v(B) = v(B) + v(A) = v(B \oplus A)$$

   et par conséquent $A \oplus B \equiv B \oplus A$. De même on montre $v(A \oplus (B \oplus C)) = v((A \oplus B) \oplus C)$ et donc $A \oplus (B \oplus C) = (A \oplus B) \oplus C$
   – Montrons par induction structurelle que toute formule est équivalente à une formule n'utilisant que 1, $\wedge$ et $\oplus$:

   Initialisation $0 \equiv 1 \oplus 1$

   Propagation Si $A \equiv A'$ avec A' s'écrivant avec 1,$\wedge$ et $\oplus$ et $B \equiv B'$(idem). On a

   $$\neg A \equiv \neg A' \equiv A' \oplus 1$$

   et

   $$A \vee B \equiv \neg(\neg A' \wedge \neg B')$$

   et on a mieux : $A \oplus B \oplus (A \wedge B)$ et enfin $A \Rightarrow B \equiv \neg A \vee B$.
3. Montrons par induction structurelle sur le système $(\oplus, \wedge, 1)$
   – Pour les littéraux :

   $$\begin{aligned} p_i &\equiv p_i \\ 1 &\equiv 1 \\ 0 &\equiv 1 \oplus 1 \end{aligned}$$

   – Soient A et B tels que $A = \bigoplus_{i=1}^{n} C_i$ et $B = \bigoplus_{j=1}^{m} D_j$. On voit immédiatement que $A \oplus B$ est de la bonne forme, et

   $$A \wedge B = \bigoplus_{(i,j) \in \mathbb{N}_n \times \mathbb{N}_m} C_i \wedge D_j$$

   ce qui clot la preuve par induction structurelle.
   Montrons par récurrence surle nombre de clauses de A' que si A' posède des clauses qui se répètent que $A' \equiv B'$ où les clauses de B' ne se répètent pas :
   Si $A' \equiv C_1$ alors c'est bon. Si $A' \equiv C_1 \oplus \cdots \oplus C_n$ et que $C_i \equiv C_j$ alors $C_j \oplus C_k = 0$, ces clauses se compensent, on peut les supprimer et on obtient une formule équivalente à A' mais ne contenant pas ces deux mêmes clauses. De même si C_i contient plusieurs fois le même littéral on peut supprimer les doublons pour obtenir une clause équivalente.
4. Il suffit de remarquer que par unicité une formule n'a pour forme normale réduite 1 que si son unique forme normale logique est 1…La complexité d'un tel algorithme permettant de convertir sous cette forme permet de décider si une formule est une tautologie ou non, sa complexité est donc supérieure à la complexité d'un algorithme permettant de décider de la satisfiabilité d'une formule[4].

[4] Que l'on peut noter S.A.T

5. Une *clause de Horn* peut ne pas être réduite. Considérons l'algorithme suivant de simplification des clauses de horn :
   - Si C est du type $[p_i]$ elle est déjà simple.
   - Si $C = [\neg p_1, \neg P_2, \ldots, \neg p_n]$, on supprime les occurences multiples de $\neg p$ en les remplaçant par une seule et on obtient une clause simple équivalente.
   - Si $C = [\neg p_1, \neg p_2, \ldots, \neg p_n, p_{n+1}]$ on supprime d'abord les doublons des clauses négatives,et si $\neg p_{n+1}$ fait partie de la clause alors $C \equiv 1$ et donc on enlève C de la liste des clauses.

   (a) **Cas n° 1** Chaque formule élémentaire contient un littéral négatif et donc sera satisfaite par $\forall i, p_i \equiv F$

   Cas n° 2 Toutes contiennent un littéral positif donc elles seront satisfaites par $\forall i, p_i \equiv V$

   (b) On a $A = \bigwedge_{i=1}^{n} E_i \wedge [p] \equiv \bigwedge_{i=1}^{n} (E_i \wedge p)$ Mais si $E_j = \neg p \vee D_{i,j}$, $E_j \wedge p \equiv (\neg p \wedge p) \vee (D_{i,j} \wedge p) \equiv D_{i,j} \wedge p$ et $A \equiv \bigwedge_{i=1}^{n} (D_{i,j} \wedge p) \equiv D_{i,1} \wedge \ldots \wedge [p]$ Mais si on a $E_j = [\neg p]$ alors $D_{i,j} = F$ alors $A \equiv F$,donc A n'es pas satisfiable.

6. Si le premier cas n'est aps réalisé alors $\exists i : E_i = [p]$ si en plus le deuxième cas n'est pas vérifié alors $\nexists j : E_j = [\neg p]$ donc $A \equiv E_i \wedge \left(\bigwedge_{j \neq i} D_{i,j} \right)$ où les $D_{i,j}$ n'ont plus la variable p. Ainsi A satisfiable équivaut à ce que $\bigwedge_{j \neq i} D_{i,j}$ satisfiable.(En fait le deuxième cas ne sertà rien) On obtient ainsi un algorithme récursif qui permet en un temps polynomial en le nombre de clauses de A de se ramener à une autre formule ayant une clause de moins ! Le temps d'exécution vérifie donc

$$T(n) \leq P(n) + T(n-1)$$

et donc

$$T(n) \leq T(0) + \sum_{k=1}^{n} P(k)$$

Deuxième partie

TP de Caml

CHAPITRE 11

TP1 : Premiers pas en Caml Light

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante. Les corrigés seront mis en ligne peu après. Tous ces documents se trouveront à l'adresse suivante :

`http ://www.nicollet.net/colles/`

11.1 Introduction

Ceux qui sont déjà familiers avec le système Caml peuvent choisir de sauter cette partie et de passer directement à la suivante.

11.1.1 Utilisation du terminal

Le terminal de `Caml Light` permet d'exécuter de manière interactive des programmes écrits dans ce langage. On peut y écrire directement les programmes, ou bien les copier-coller à partir d'un éditeur de texte. Le terminal attend de rencontrer le mot-clé `;;` (deux point-virgule consécutifs) avant de commencer à traiter le programme. Lorsque celui-ci est rencontré, il évalue le résultat du programme et l'affiche au format suivant :

`-` `:` *type* `=` *valeur*

Question 1

À l'aide du terminal `Caml Light`, déterminer le type et la valeur du résultat de chacun de ces programmes :

```
1;;
13 + 5;;
(7+9)/2;;
3 > 4;;
"Bonjour";;
1.2;;
```

Plus généralement, toute expression mathématique faisant intervenir des entiers et les opérations simples `+` `-` `*` ou `/` est un programme `Caml Light` correct dont le résultat est égal à la valeur de cette expression.

11.1.2 Types des expressions

Le terminal, en plus de calculer le résultat d'une expression, calcule également son type : `int` pour un entier, `float` pour un réel, `bool` pour une valeur booléenne, `string` pour du texte, et bien d'autres encore. L'intérêt de cela est de pouvoir détecter une partie des erreurs qui peuvent se glisser dans le programme : le terminal interdit l'exécution d'une opération sur des objets qui ne sont pas du bon type, et affiche une erreur expliquant quelle partie du programme comporte l'erreur.

Question 2

Parmi les programmes suivants, lesquels comportent une erreur de type, et lesquels fonctionnent correctement ? Pouvez-vous expliquer pourquoi ?

```
(1 +  2) = 3;;
 1 + (2  = 3);;
"Texte" + "autre texte";;
exp(1.234) < 1.5;;
2 * 3.141592;;
```

Afin d'éviter les erreurs, `Caml Light` interdit de mélanger des valeurs `int` et `float`. Les opérations sur des nombres réels se font avec `+. -. *.` ou `/.` (on ajoute un point après le symbole). Il faut également faire attention aux constantes : `2 : int`, mais `2. : float`.

Ainsi, il est possible de corriger le dernier programme de la question 2 en écrivant : `2. *. 3.141592 ; ;`

11.1.3 Noms de variables

Au moyen de l'instruction `let`, il est possible de donner un nom à une valeur. Par exemple, `let pi = 3.141592 ; ;` donne le nom `pi` à la valeur 3.141592. Toute utilisation du nom sera ensuite remplacée par la valeur qui y est associée.

Un nom continue à designer la valeur à laquelle il est attaché jusqu'à ce qu'une nouvelle instruction `let` lui indique de désigner une autre valeur. Cette modification n'a aucun effet sur les utilisations précédentes du nom.

Lorsqu'une telle instruction est exécutée par le terminal `Caml Light`, elle renvoie une réponse de la forme : *nom* `:` *type* `=` *valeur*

Question 3

Essayer de prévoir le résultat de la suite d'instructions ci-dessous sans l'exécuter, puis vérifier les prédictions à l'aide du terminal.

```
let y = 10;;
let y = x + x;;
let x = 20;;
x + y;;
```

On note P le polynôme à coefficients entiers $3X^2 + 2X - 1$. Donner les noms a, b, c et x aux bonnes valeurs pour que le programme ci-dessous calcule la valeur de $P(0)$. Même question pour $P(1)$.

```
(a * x + b) * x + c;;
```

11.2 Outils fréquemment utilisés

11.2.1 Fonctions

En plus des valeurs de base vues précedemment, `Caml Light` permet de manipuler des fonctions :

```
function x -> x * x;;
```

Cet exemple définit une fonction qui associe à chaque entier son carré. Son type est donc `int -> int`. Il est bien sûr possible de donner un nom aux fonctions. Pour cela, on procède exactement de la même manière qu'avec une autre valeur :

```
let f = function x -> x * x;;
f(2);;
```

On peut procéder de manière plus courte en écrivant :

```
let f(x) = x * x;;
```

Question 4

En utilisant la fonction f ci-dessus, construire la fonction g qui à l'entier x associe la valeur en x du polynôme de la question 3). Vérifier la valeur de g en -1, 0 et 1.

11.2.2 Conditions

Lorsque l'expression à évaluer doit dépendre de la valeur d'une autre expression, on utilise la structure conditionnelle `if .. then .. else ...` Par exemple, la fonction pour calculer une valeur absolue peut être définie comme :

```
let abs(x) = if x < 0 then -x else x;;
```

Question 5

Écrire une fonction `pair` qui à un entier pair[a] associe le texte `"pair"` et qui à un entier impair associe le texte `"impair"`.
L'utiliser pour écrire la fonction définie par :

$$f(x) = \begin{cases} x/2 & \text{si } x \text{ est pair} \\ 3x+1 & \text{si } x \text{ est impair} \end{cases}$$

[a] On peut, pour connaître la parité d'un entier, se servir de l'opérateur modulo : `a mod b` est égal à 0 si et seulement si a est divisible par b.

11.2.3 Définitions locales

Pour éviter d'utiliser trop d'identifiants, on peut n'utiliser un nom de variable que localement :

```
let g(x) = let f(x) = x * x in f(x) + 1;;
           ^^^^^^^^^^^^^^^^^^
```

Ce code associe le nom `f` à la fonction x^2, à l'intérieur de la fonction `g`.

Question 6

Déterminer toutes les racines du polynôme g défini par :

```
let g(x) =
let x = x - 2 in
let f(x) = x * x - 1 in
let f(x) = f(x+3) in
let f = f(x) in
f;;
```

11.2.4 Couples et n-uplets

On peut construire des couples ou des n-uplets de valeurs grâce à la syntaxe suivante :

```
0,1,2,3,4,5,6,7;;
```

N'importe quelle valeur (entier, fonction, p-uplet) peut être élément d'un n-uplet. `Caml Light` permet également d'utiliser les expressions idiomatiques suivantes :

```
let couple = (1,2);;
let (x,y) = couple;;

let triplet = (x,y,3);;
let f(x,y,z) = x + y + z in f(1,2,3) = f(triplet);;
```

La première expression permet de nommer les n éléments d'un n-uplet, et la deuxième d'utiliser des functions à plusieurs variables pour lesquelles il est possible de passer les arguments un à un ou tous en même temps.

Question 7

Écrire une fonction qui évalue le produit scalaire de deux vecteurs écrits comme des couples de réels. L'utiliser pour écrire une fonction qui évalue la norme euclidienne d'un vecteur.

11.3 Sujets avancés

11.3.1 Propriétés fonctionnelles

Une fonction est un objet comme les autres. Elle peut par exemple être un argument d'une autre fonction.

```
let g(f) = f(1);;
g(function x -> x+1);;
```

Une conséquence de cela est la Curryfication (d'après Haskell Curry) ; il est possible d'écrire une fonction qui à une valeur associe une fonction.

```
let somme(x) = (function y -> x + y) in
let ajoute_un = somme(1) in
somme(1)(2) = ajoute_un(2);;
```

Ici, `ajoute_un` est une fonction, qui ajoute 1 à son argument entier. Il est possible d'écrire de manière abrégée le code ci-dessus grâce à la notation curryfiée :

```
let somme_curry x y = x + y;;
```

Question 8

Construire une fonction h telle que pour toutes fonctions f et g, $h(f, g)$ soit la fonction H définie par :

$$H(x) = 1 + f(1 + g(1 + x)))$$

Évaluer cette fonction avec $f(x) = x + 1$ et $g(x) = x - 2$ pour $x \in \{1, 3, 5, 16\}$.

11.3.2 Polymorphisme

Le polymorphisme permet de réutiliser une fonction dans de nombreuses situations différentes en la laissant s'adapter à tous les types. En effet, chaque fonction attend des arguments dont le type est fixé, et dans ces conditions il devrait être nécessaire de réécrire une fonction aussi simple que l'identité une fois pour chaque type auquel on voudrait l'appliquer : `int`, `float`, `int -> int`, `(int -> int) -> (int -> int)` :

```
let id(x) = x;;
id(1);;
id(1.);;
id(function x -> x+1);;
id(id);;
```

`Caml Light` contourne cela en autorisant une fonction à avoir une infinité de types. Ainsi, lorsqu'on appelle la fonction sur un entier, elle devient de type `int -> int`. Lorsqu'on l'appelle sur un nombre réel, elle devient de type `float -> float`. On utilise pour représenter cela une variable de type : `'a` est un type, n'importe lequel, et pour n'importe quelle valeur de `'a` la fonction `id` est de type `'a -> 'a`. Si nécessaire, on utilise les variables `'b`, `'c` et ainsi de suite. Par exemple :

```
# let premier(x,y,z) = x;;
premier : 'a * 'b * 'c -> 'a = <fun>

# premier(1,2,3);;
- : int = 1

# premier(true,"bonjour",(function x -> x + 1));;
- : bool = true
```

Question 9

Quel serait le type de l'opérateur composition C défini par : $C(f, g)$ est la fonction qui à x associe $g(f(x))$?
Implémenter cet opérateur en `Caml Light`, vérifier qu'il est du type prévu. L'essayer avec les fonctions de transformation `float_of_int` et `int_of_float`.

11.3.3 Effets de bord

Jusqu'à présent, le résultat d'une fonction était déterminé uniquement par ses arguments, et décrit entièrement par sa valeur. Il existe des cas où ce résultat dépend d'autres paramètres : des fonctions qui lisent un texte entré par l'utilisateur, ou qui génèrent des nombres aléatoires. Il existe également des fonctions dont l'effet ne se limite pas à retourner une valeur, mais qui ont également des effets externes (dits *de bord*) : les fonctions qui affichent ou dessinent, mais

également les fonctions de lecture ou de nombre aléatoire, dont l'appel modifie indirectement le résultat des appels ultérieurs.

On utilise pour simplifier l'écriture un paramètre vide, appelé `unit`, représenté par `()`, et des point-virgule simples qui permettent de séparer des instructions sans que le terminal les interprète séparément. L'exemple suivant crée deux nombres aléatoires entre 1 et 6 et les affiche :

```
let alea( ) =
let valeur = random__int(6) + 1 in
print_string "La valeur du dé est: ";
print_int valeur;
print_newline( );;

alea( );;
alea( );;
```

Les fonctions de lecture et écriture sont symétriques : `read_int read_line read_float` et `print_int print_string print_newline print_float`. Les fonctions `random__int random__float` renvoient une valeur aléatoire comprise entre 0 inclus et leur argument exclus.

Question 10

Écrire une fonction qui tire au hasard deux nombres entre 0 et 9, et renvoie leur somme. Écrire une fonction qui calcule [a] la probabilité d'avoir obtenu cette somme et l'afficher sous la forme suivante :

```
Somme: 12
Probabilité: 6%
```

[a] Si $x \leq 9$, il existe x lancers possibles pour lesquels la somme est égale à x, et $18 - x$ lancers possibles si $x \geq 9$

11.4 Pour finir

Question 11

On va écrire une petite calculatrice capable d'interpréter des commandes[a] de la forme suivante :

```
+
10
35
```

D'abord, écrire une fonction `operation(op)` qui prend en argument une fonction, lit deux entiers x et y, et affiche le résultat de `op(x,y)`.
Écrire également une fonction `erreur(ligne)` qui indique que le texte `ligne` n'est pas une opération connue.
Les utiliser pour écrire la fonction `calculatrice( )`, qui lit une ligne de texte. Si celle-ci vaut `"+"`, alors elle lit deux entiers, les ajoute et affiche le résultat. Si celle-ci vaut `"-"`, alors elle lit deux entiers, soustrait le second au premier, et affiche le résultat. Dans les autres cas, elle affiche `"opération inconnue"` suivi de la ligne lue.
Tester la calculatrice, puis l'étendre à d'autres opérations.

[a] La calculatrice va lire les commandes qui lui sont envoyées à travers le terminal

11.5 Bonnes adresses

Il est possible de télécharger `Caml Light` à l'adresse suivante :

`http ://caml.inria.fr/download.fr.html`

Le manuel de référence en anglais, qui n'est pas vraiment adressé aux débutants en informatique, se trouve quant à lui sur :

`http ://caml.inria.fr/pub/docs/manual-caml-light/`

11.6 Correction

Question 1 : Voilà les différents résultats que l'on obtiendra en tapant les commandes que l'on proposait :

Exercice 11.6.1:

```
1;;
  - : int = 1

13 + 5;;
  - : int = 18

(7+9)/2;;
  - : int = 8

3 > 4;;
  - : bool = false

"Bonjour";;
  - : string = "Bonjour"

1.2;;
  - : float = 1.2
```

Question 2 : Voici les résultats obtenus en tapant les différents codes proposés :

$(1 + 2) = 3$;;
(* Ce code est correct et renvoie 'true' *)
$1 + (2 = 3)$;;
(* Incorrect : il additionne l'entier 1 et la valeur 'false' *)
"Texte" + "autre texte" ;;
(* Incorrect : l'addition '+' ne peut s'appliquer que sur des entiers *)
$\exp(1.234) < 1.5$;;
(* Correct : on applique exp() à un réel, et on compare son résultat réel * à un autre réel *)
2 * 3.141592 ;;
(* Incorrect : '2' et '*' sont des entiers, on ne peut pas les mélanger avec un réel comme '3.141592' *)

CHAPITRE 12

TP2 : Références, boucles, tableaux

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante. Les corrigés seront mis en ligne peu après. Tous ces documents se trouveront à l'adresse suivante :

`http ://www.nicollet.net/colles/`

12.1 Références

Une référence est une valeur qui se comporte comme une flèche : elle pointe en permanence vers une autre valeur, et il est possible de la réorienter pour la faire pointer vers une autre valeur du même type.

Le code suivant crée une référence (pointant vers 2), change sa valeur (la réoriente vers 3), et renvoie sa valeur (l'objet vers lequel elle pointe) :

```
# let x = ref 2 ; ;
x : int ref = ref 2

# x := 3 ; ;
- : unit = ()

#!x ; ;
- : int = 2
```

Une référence vers des valeurs de type `'a` aura pour type `'a ref`. Cela permet d'empêcher les erreurs où une valeur d'un type incorrect serait attribuée à une référence.

Question 1

Que font les programmes suivants ?

```
lex f(x) =
  incr x ; !x
in f(ref 17) ; ;

let a = ref 3 in
let b = a in
a := 4 ;
!b ; ;
```

Question 2

Écrire les fonctions :
- `get(x)` qui prend en argument une référence x et renvoie son contenu.
- `set(x)(v)` qui prend en argument une référence x et la fait pointer vers v.
- `clone(x)` qui renvoie une référence pointant vers le même objet que x.

Quel est leur type ?

12.2 Boucles

Il est souvent nécessaire d'effectuer plusieurs fois de suite une même opération (à effet de bord) dans un algorithme. Pour cela, on utilise des boucles : il en existe deux sortes en `Caml Light`, chacune étant destinée à un certain ensemble de situations.

12.2.1 Boucle `for`

La boucle `for` permet d'exécuter une série d'opérations un nombre prédéterminé de fois. Par exemple, pour afficher les entiers de *min* à *max* (inclus), on écrit :

```
for entier = min to max do
  print_int entier;
  print_newline ( )
done
```

Une telle boucle s'exécute une fois pour chaque valeur entière entre le minimum et le maximum. Le type d'une boucle est `unit` : il est possible de l'utiliser à gauche d'un point-virgule.

Question 3

Afficher la table de multiplication (un carré 10×10).

Il est possible d'utiliser une référence pour mémoriser une valeur calculée pendant l'exécution d'une boucle (puisque la boucle peut modifier la référence), et utiliser ensuite la valeur de la référence.

La boucle `for` s'avère très utile lorsqu'on cherche à calculer le n-ème terme d'une suite définie par récurrence, ou pour des opérations comme des sommes ou des produits d'un nombre prédéterminé de termes.

Question 4

Écrire les fonctions qui calculent le n-ème terme des suites d'entiers suivantes :

$$a_{n+1} = f(n, a_n) \quad a_0 = x$$

$$u_{n+1} = (n+1) \cdot u_n \quad u_0 = 1$$

$$w_{i+2} = w_{i+1} + w_i \quad w_0 = w_1 = 1$$

Où f est une fonction et x une valeur, qui seront des arguments de la fonction `a(f,x)(n)` calculant le n-ème terme de a_n.
On remarquera que certaines suites peuvent s'exprimer en fonction d'autres.

12.2.2 Boucle `while`

La boucle `while` permet d'exécuter une série d'opérations jusqu'à ce qu'une propriété devienne vraie. Par exemple, on peut écrire un programme qui divise un entier par deux jusqu'à ce qu'il soit nul :

```
let entier = ref (read_int ( )) in
while !entier <> 0 do
  print_int !entier ; print_newline( ) ;
  entier := !entier / 2
done
```

La boucle `while` est utile lorsqu'on souhaite compter des choses. Pour cela, on utilise en général une référence entière dont la valeur initiale est zéro, et on y ajoute un (avec `incr`) à chaque fois que la boucle est exécutée.

Question 5

Soit la suite de Syracuse définie par son terme initial $u_0 > 0$ et la relation de récurrence :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Écrire une fonction qui, étant donnée la valeur de u_0, calcule la plus petite valeur de n pour laquelle $u_n = 1$.

Plus généralement, on utilise une boucle `while` dès qu'il devient difficile d'exprimer le nombre de répétitions de la boucle avant de la commencer. En particulier, c'est le cas lorsque le nombre de répétitions est une conséquence directe du travail de la boucle.

Même si le nombre maximal d'opérations est connu à l'avance, on peut vouloir (pour des raisons de vitesse) arrêter l'exécution dès que le résultat est obtenu.

Question 6

Déterminer si un entier $n \geq 2$ est premier[a].
Vérifier qu'il y a 16342 nombres premiers inférieurs à 180000.

[a] Indication : n est premier ssi aucun entier $1 < m \leq \sqrt{n}$ ne divise n

---

12.3 Tableaux

Les références ne permettent de conserver qu'une quantité fixe de données en mémoire. Lorsqu'on souhaite pouvoir manipuler des quantités plus grandes de données, on utilise des tableaux. Le code suivant crée un tableau contenant les entiers premiers inférieurs à 20 :

```
[| 2 ; 3 ; 5 ; 7 ; 11 ; 13 ; 17 |] ; ;
```

Il est également possible de créer un tableau de taille arbitraire avec la fonction `make_vect`. Ci-dessous, un tableau de taille 10 dont tous les éléments sont le mot "Bonjour".

```
let bonjour = make_vect 10 "Bonjour" ; ;
```

On peut enfin accéder, en lecture et en écriture, aux éléments d'un tableau. Dans un tableau de taille `n`, les élements sont numérotés de `0` inclus à `n` exclus. Le code ci-dessous lit donc le troisième élement, puis modifie le septième et affiche le tableau modifié.

```
bonjour.(2) ; ; bonjour.(6) <- "Au revoir" ; bonjour ; ;
```

Les tableaux ont chacun leur type. Un tableau d'objets `'a` sera un `'a vect`, et on ne peut y écrire des objets d'un type autre que `'a`. On peut connaître la longueur d'un tableau en lui appliquant la fonction `vect_length`.

Question 7

Écrire une fonction qui prend en argument deux tableaux a et b de réels et calcule leur produit scalaire, à savoir (avec n la taille du plus petit) $\sum_{i=0}^{n-1} a_i b_i$.

Lorsque le résultat recherché est un tableau, une boucle `for` est en général un très bon choix si la longueur du tableau est connue à l'avance.

Question 8

Écrire une fonction qui prend en argument une taille n et un réel X, et qui construit un tableau dont le i-ème élément est X^{i-1}.

12.4 Pour finir

12.4.1 Polynômes

On souhaite travailler sur des polynômes, qu'on choisit de représenter comme des tableaux de coefficients réels. Ainsi :

`[| 1. ; -2. ; 0. ; 3. ; 0. |]` représente $3X^3 - 2X + 1$

Plus généralement, si $n =$ `vect_length P` $- 1$,

$$P(X) = \sum_{i=0}^{n} \texttt{P.(i)} X^i$$

Question 9

Écrire une fonction `deg` calculant le degré d'un polynôme (l'indice de son plus grand coefficient non nul). Pour simplifier, on convient que le degré d'un polynôme nul est 0.

Écrire une fonction `eval` qui calcule la valeur d'un polynôme pour une valeur de X donnée. Utiliser les questions 7 et 8.

Construire le terme $h_n(X)$ de la suite définie par : $h_k(X) = a_{n-k} + X h_{k-1}$ avec $h_0(X) = a_n$. Cette suite vérifie : $h_n(X) = \sum_{i=0}^{n} a_i X^i$. Utiliser cette propriété pour en déduire une fonction `eval2` plus rapide que `eval`.

12.4.2 Cours de la bourse

On veut jouer en bourse et de gagner de l'argent. On s'intéresse pour cela à une action en particulier. On connaît son prix sur une période donnée, au jour le jour. On souhaite en acheter une unité un jour a (en dépensant le prix qu'elle a atteint), puis la revendre un jour $v \geq a$ (en gagnant le prix de l'action le jour v). Le profit est, bien entendu, la différence entre le prix auquel on l'a vendu et le prix auquel on l'a acheté.

Question 10

Écrire une fonction capable de calculer le jour d'achat et le jour de vente qui permettent le profit maximal.

(Indication 1) En supposant qu'on veuille vendre à une date v donnée, trouver la date a à laquelle il faut acheter pour maximiser le profit.

(Indication 2) En connaissant le jour d'achat a optimal pour un jour de vente v, comment peut-on calculer le jour d'achat optimal pour le jour $v + 1$?

On supposera que les prix sont donnés comme un tableau d'entiers de taille supérieure ou égale à 1, dont le i-ème élément est le prix de l'action pour le jour i.

Créer un tableau de taille 20.000 contenant des entiers aléatoirement choisis et entre 0 et 150.000, et calculer les jours d'achat et de vente, ainsi que le profit obtenu.

12.5 Correction

Question 1 : let f(x) =
(* Ajouter 1 à la référence x *) incr x ;
(* Renvoyer la valeur de la référence *) !x in
(* Appeler f sur une référence à 17, ce qui renvoie 18 *) f(ref 17) ;;
(* Créer une référence vers 3 et lui donner le nom a *) let a = ref 3 in
(* Lui donner également le nom b *) let b = a in
(* Modifier la référence à travers le nom a *) a := 4 ;
(* Renvoyer la valeur de la référence à travers le nom b, ce qui renvoie 4 *) !b ;;

Question 2 : Voilà quelques fonctions dont l'utilité saura vous ravir...

```
let get(x) = !x;;
```

(* 'a ref -> 'a *)

```
let set(x)(v) = x := v;;
```

(* 'a ref -> 'a -> unit *)

```
let clone(x) = ref !x;;
```

(* 'a ref -> 'a ref *)

Question 3 : Affichons un table de multiplications :

Exercice 12.5.3: *Table...*

```
# let table () =
      for i = 1 to 10 do
         for j = 1 to 10 do
            print_int (i*j);
            if i*j<10 then print_string " ";
            if i*j<100 then print_string " ";
            print_string " ";
         done;
         print_newline();
      done;;
```

Question 4 : Et voilà quelques suites :

Exercice 12.5.4: *Des suites*

```
#let rec suite f u0 = function
     |0->u0
     |n->f(n,(suite f u0 (n-1)));;
#let factorielle =
     suite (function (x,y)->x*y) 1;;

#let fibonnaci k=
    let rec f = function
      |0|1->(1,1)
      |n-> let (a,b) = f(n-1) in
           (a+b,a) in
    let (a,b) = f(k) in a;;
```

Question 5 : Voilà la très célèbre suite de syracuse[1]

Exercice 12.5.5:
```
# let syracuse x =
      let u = ref  0 in
      let suite = ref x in
      while !suite <>1 do
          incr(u);
          if !suite mod 2 = 0 then suite:= (!suite)/2
          else suite:= 3*(!suite) +1;
          done;
      !u;;
```

Question 6 : Quelques fonctions sur les nombres premiers...

Exercice 12.5.6:
```
# let estpremier n=
       if n<2 then false
       else (let u = ref true and k = ref 2 in
            while !u && (!k)*(!k) <= n do
                u:=(n mod !k <> 0);
                incr(k);
            done;
            !u);;

# let combien_de_premiers a b =
       let compte = ref 0 in
       for i = a to b do
           if estpremier i then incr(compte);
           done;
       !compte;;
```

Question 7 : On va faire une fonction produit scalaire de deux tableaux

Exercice 12.5.7: *Produit scalaire*
```
# let scalaire P Q =
      let n = vect_length P
      and m = vect_length Q in
      let p = min n m in
      let result = ref 0. in
      for i = 0 to pred(p) do
         result := !result +. P.(i)*.Q.(i);
         done;
      !result;;
```

Question 8 : Et maintenant des puissances !

[1] De l'université du même nom, je suppose

Exercice 12.5.8: *Puissances !*

```
# let power x n =
     let t = make_vect n 1. in
     for i = 1 to n-1 do
        t.(i)<-t.(i-1)*.x;
        done;
     t;;
```

Question 9 : Et ces fonctions vont nous servir à faire un peu de calcul sur les polynômes.

Exercice 12.5.9: *Et des polynômes !*

```
# let deg p =
      let n =ref (vect_length p -1) in
      while p.(!n) = 0. && !n>0 do decr(n); done;
      !n;;

# let eval p x =
      let n = deg p +1 in
      let t = power x (n+1) in
      scalaire p t;;

# let horner (p) (x) =

  let n = deg p in
  let h = ref p.(n) in

  (* Pour k > 0, h(k) = p(n-k) + X h(k-1) *)
  for k = 1 to deg p do
    h := p.(n-k) +. x *. (!h)
  done;

  !h;;
```

Question 10 : Et pour finir on va se faire un peu d'argent de poche en jouant les madame Soleil !

Exercice 12.5.10:

```
# let onvagagner t =
let n = vect_length t and achat = ref 0 and vente = ref 0 and
achatbis = ref 0 and ventebis = ref 0 and magot = ref 0 and
pactole = ref 0 in
for i = 1  to n-1 do
          if t.(i)<t.(!achatbis) then
               (if !magot>(!pactole) then (achat:=!achatbis;
                    vente:=!ventebis;pactole:=!magot);
                magot:=0;achatbis:=i;ventebis:=i)
          else if t.(i)>t.(!ventebis) then
               (ventebis:=i;
                magot:=(t.(!ventebis)-t.(!achatbis)));
        done;
        if !pactole>=(!magot) then (!achat,!vente)
        else (!achatbis,!ventebis);;
```

CHAPITRE 13

TP3 : Listes, récursion, types somme

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante. Les corrigés seront mis en ligne peu après. Tous ces documents se trouveront à l'adresse suivante :

`http ://www.nicollet.net/colles/`

13.1 Récursion

On dit qu'une fonction est *récursive* lorsqu'elle apparaît dans sa propre définition. Il s'agit de la traduction dans le langage `Caml Light` des suites définies par récurrence. Par exemple :

$$u_n = u_{n-1} \cdot n \qquad u_0 = 1$$

Se traduit en `Caml Light` par la fonction récursive :

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

Le symbole `rec` indique que la fonction est récursive : le nom `u` est disponible à l'intérieur de la définition (et pas uniquement après la fin de celle-ci : essayez de retirer le `rec`, pour voir).

Lorsque `u(n)` est exécutée, elle va appeler `u(n-1)`, qui va appeler `u(n-2)` et ainsi de suite jusqu'à 0. Il va donc y avoir $n+1$ appels à `u` pour calculer `u(n)`.

Question 1

Écrire une fonction récursive simple qui calcule le terme n de la suite de Fibonnaci :

$$u_n = u_{n-1} + u_{n-2} \qquad u_0 = u_1 = 1$$

On note A_n le nombre d'appels à `u` qui sont effectués lorsqu'on calcule `u(n)`. Que peut-on dire sur la suite A_n ? Comparer avec le nombre d'opérations effectuées si on avait utilisé une boucle `for`.

Question 2

On remarque que :

$$x^0 = 1 \qquad x^{2k} = (x^2)^k \qquad x^{2k+1} = x(x^2)^k$$

Si l'on dispose d'une fonction qui est capable de calculer efficacement x^k pour tout x et pour tout $k < n$, comment calculer efficacement x^n ?
Écrire la fonction récursive utilisant cette methode (appelée *exponentiation rapide*) pour calculer x^k pour x et k donnés.

Un problème peut être résolu récursivement s'il vérifie deux conditions :

- Il exite une notion de *taille* : par exemple, le problème de calculer u_6 est plus grand que celui de calculer u_5. Plus généralement, il faut arriver à définir une notion de taille à partir des arguments : taille d'un intervalle de recherche, somme en valeur absolue des arguments, taille de l'argument ou autres propriétés du problème.
- Il est possible d'exprimer un problème à partir de problèmes strictement plus petits (et, pour les plus petits problèmes, la solution est triviale).

Ces deux propriétés assurent qu'une fonction récursive pourrait résoudre un problème en le réduisant à des problèmes de plus en plus petits, jusqu'à atteindre un problème si petit qu'il devient trivial à résoudre.

Question 3

On s'intéresse pour la troisième fois à la suite de Syracuse, définie par son premier terme et par la récurrence :

$$s_{n+1} = \begin{cases} s_n/2 & \text{si } s_n \text{ est pair} \\ 3s_n + 1 & \text{sinon} \end{cases}$$

Cette fois, on s'intéresse aux antécédents de rang n d'une valeur donnée[a].
Écrire une fonction `ante(y)(n)` qui affiche tous les antécédents de `y` de rang inferieur ou égal à `n`, séparés par des espaces. Pour simplifier, l'ordre ne sera pas important, et un même nombre pourra apparaître plusieurs fois.

[a] On considèrera que x est un antécédent de rang n de y si et seulement si, lorsque l'on fixe le premier terme $s_0 = x$, $s_n = y$.

13.2 Listes

Une valeur `v` est une liste dans les deux cas suivants :
- si elle est la liste vide, notée `v = []`
- ou bien si elle contient une valeur `h` (*head*, la tête de `v`) et une liste `t` (*tail*, la queue de `v`). On note alors `v = h::t`

Les listes sont donc une structure récursive, puisque toutes les listes (sauf la liste vide) sont définies à partir d'une liste plus petite (leur queue). Par exemple, une liste des entiers de 1 à 3 est `1::(2::(3::[]))`, qu'il est possible pour des raisons de lisibilité, d'écrire de manière abrégée `[1; 2; 3]`

Contrairement aux tableaux, les listes ne peuvent pas être modifiées. La manipulation des listes ne comporte donc que deux possibilités : parcourir la liste pour en extraire des informations, ou créer une nouvelle liste.

13.2.1 Créer une liste

Créer une liste est très simple, et se fait grâce à une fonction récursive. La queue est la valeur de la fonction sur un problème plus petit, et la tête est le résultat d'un calcul supplémentaire. Dans les problèmes triviaux, la fonction renvoie souvent la liste vide. Par exemple, la liste des carrés des entiers inférieurs à n est donnée par :

```
let rec carres (n) =
  if n = 0 then []
  else (n*n)::(carres (n-1));;
```

Question 4

Écrire une fonction qui calcule la liste de diviseurs premiers d'un entier n, dans l'ordre croissant et apparaissant autant de fois dans la liste que dans n. Le résultat attendu est donc, par exemple :

```
# diviseurs(18);;
- : int list = [ 2; 3; 3 ]
```

On pourra, pour s'aider, écrire une fonction qui calcule le plus petit diviseur premier d'un nombre.

13.2.2 Parcourir une liste

Pour parcourir une liste, on utilise également une fonction récursive. Celle-ci utilise le puissant système de motifs de `Caml Light` (le *pattern matching*) pour détecter si l'on a atteint la fin de la liste (un problème trivial) ou si l'on peut réduire le problème à un problème plus petit (la queue de la liste).

Par exemple, la fonction ci-dessous donne la longueur d'une liste :

```
let rec length = function
  | [] -> 0
  | h::t -> length t + 1;;
```

`Caml Light` parcourt dans l'ordre les motifs (qui commencent après chaque `|`). Si un motif correspond à l'objet examiné, alors le code qui est juste à droite du `->` qui suit est utilisé pour calculer la valeur. Les noms qui ont été donnés aux sous-parties de l'objet dans le motif sont disponibles dans ce code (comme `t` dans l'exemple ci-dessus).

Question 5

Écrire un programme qui calcule la somme des éléments d'une liste d'entiers.
Écrire un programme qui calcule le produit des éléments d'une liste de réels.

13.2.3 Transformation de listes

On ne peut pas transformer les listes, puisqu'on ne peut pas les modifier ! En revanche, on peut créer une liste à partir des données lues dans une autre. Pour cela, il suffit en général d'écrire une fonction qui lit des données dans une liste, et qui renvoie une liste.

Question 6

La fonction `list_map` prend en argument une liste `[x1 ; x2 ...xn]` et une fonction `f` et renvoie la liste `[f(x1) ; f(x2) ...f(xn)]`.
Réecrivez vous-même cette fonction.

Question 7

Écrire une fonction `insert` qui, étant donnés une liste triée `l` et un élément `e`, renvoie la liste triée contenant `e` ainsi que les éléments de `l` (on parle d'insérer un élément dans une liste triée).
En déduire (et écrire) une fonction qui applique le tri par insertion pour trier une liste.

13.3 Autres structures récursives

La définition récursive de la liste est : `[]` ou `h: : t`.

Il est possible en `Caml Light` de définir des types suivant ce modèle, en utilisant le mot-clé `type` et une syntaxe proche de celle du pattern matching. Par exemple, on peut définir une liste d'entiers simple suivant le modèle des vraies listes :

```
type autreListe =
  | ListeVide | TeteQueue of int * autreListe ; ;

let rec longueur = function
  | ListeVide -> 0 | TeteQueue (h,t) -> longueur t + 1 ; ;

longueur (TeteQueue(10,TeteQueue(3,TeteQueue(7,ListeVide)))) ; ;
```

Question 8

On utilise le type suivant :

```
type expression =
  | Var | Const of float
  | Mul of expression * expression
  | Add of expression * expression | Sub of expression * expression ; ;
```

Ce type permet de représenter des expressions mathématiques. Par exemple, on représente $(1 + x) * 2$ par `Mul (Add (Const 1.,Var),Const 2.)`.

- Écrire une fonction qui affiche l'expression sous une forme lisible. Attention au parenthésage !
- Écrire une fonction récursive qui renvoie l'expression dans laquelle on a remplacé toutes les occurences de la variable une valeur donnée. Par exemple, remplacer la variable par 1 dans $(x + 1) * x$ donne $(1 + 1) * 1$
- Écrire une fonction qui récursive lit une expression sans variables et qui calcule son résultat. Par exemple, $(1 + 1) * 2$ donne 1.
- Écrire une fonction qui calcule la dérivée d'une expression par rapport à une variable donnée.

13.4 Sujets divers

13.4.1 Reformulations

Parfois, il est utile de reformuler le problème, de manière à ce qu'il comporte une notion de taille. Par exemple, chercher l'élément minimal d'un tableau est un problème qui ne fait pas intervenir de taille. En revanche, chercher l'élément minimal entre les indices 0 et n d'un tableau fait intervenir explicitement la taille n du problème.

Lorsqu'on reformule un problème, on écrit d'abord une fonction qui résout le problème modifié. Puis on écrit une fonction qui traduit le problème original en le problème modifié, utilise la fonction précédente, et renvoie le résultat. Par exemple :

```
let min_tableau (tabl) = (* Probleme initial *)
  let rec min_rec_tableau (n) = (* Probleme modifie *)
    if n = 0 then tabl.(0)
    else min (tabl.(n)) (min_rec_tableau (n-1))
  in min_rec_tableau (vect_length tabl - 1) ; ;
```

Question 9

Écrire une fonction `second(liste)` qui calcule récursivement le deuxième plus grand élément d'une liste (celui qui serait en deuxième position si la liste était triée). La complexité doit être linéaire en la taille de la liste (on ne peut donc pas la trier).

13.4.2 Récursion terminale

On parle de récursion terminale (*tail recursion*) lorsque la dernière chose que renvoie une fonction récursive est la fonction elle-même. Par exemple :

```
let facto (n) =
```

```
let rec tail (n) (m) =
  if n = 0 then m
  else
    tail (n-1) (n*m)
in tail (n) (1) ; ;
```

Chaque appel consomme un peu de mémoire de ce qu'on appelle la *pile*, que le programme utilise pour se souvenir des opérations qui restent à faire dans chaque appel. La factorielle normale ne peut fonctionner pour n grand, car elle doit retenir l'opération `n*` pour tout `n`. Au contraire, la récursion terminale ne coûte pas de mémoire (il n'y a rien à retenir), et fonctionne donc pour n'importe quelle valeur de n.

Une telle fonction transmet la valeur finale en argument à chaque appel, et renvoie cet argument lorsque la récursion est terminée. C'est le principe de l'argument `m` ci-dessus.

Question 10

Écrire une fonction en récursion terminale `rev` qui inverse une liste (au sens que `[1 ; 3 ; 2 ; 4]` devient `[4 ; 2 ; 3 ; 1]`).

En déduire une fonction en récursion terminale qui prend en argument une liste `l` et un élément `e`, et renvoie une liste qui contient tous les éléments de la liste initiale sauf `e` (et dans le même ordre).

13.4.3 Pattern matching

Le *pattern matching* est très puissant : il permet de reconnaître à peu près n'importe quelle forme d'objets, en appliquant dessus des contraintes arbitraires et en donnant des noms aux différentes parties et sous-parties de la forme reconnue. Voici un exemple un peu plus avancé pour compter le nombre d'élements consécutifs égaux dans une liste :

```
let rec doublons = function
  | []| [_ ] -> 0
  | a: : (b: : _ as t) when a = b -> doublons t + 1
  | _ : : t -> doublons t ; ;
```

- La deuxième ligne dit que si la liste est vide, ou contient un seul élément, alors il n'y a pas de doublons. Le `_` signifie qu'on ne donne pas à l'élément en question un nom.
- La troisième dit que si la liste contient au moins deux éléments, on appelle `a` le premier et `b` le second, et on donne le nom `t` à la liste `b: : _` avec `as`. Si la contrainte `a = b` introduite par `when` est vérifiée, alors on calcule le nombre de doublons de `t` et on y ajoute 1.
- La quatrième ligne gère les cas restants, où le premier et deuxième élément sont différents, et calcule le nombre de doublons de la queue.

Question 11

(Très difficile, seulement si vous avez le temps)

On travaille sur le même type que l'exercice 8. Notre objectif est d'intégrer une expression. Malheureusement, même sur des expressions aussi simples (juste des opérations de base) l'intégration est très difficile. Il faut donc commencer par pré-traiter l'expression pour la mettre sous une forme que nous sommes capables d'intégrer.

Étape 1 Utiliser le pattern matching pour écrire une fonction qui remplace dans une expression toutes les soustractions $a-b$ par des additions $a+(-1*b)$.

Étape 2 Utiliser le pattern matching pour écrire une fonction qui lit une expression et développe tous les produits de sommes en sommes de produits. Par exemple :
$(1+X)\cdot(1+2\cdot(1+X))$
est transformée en
$1\cdot 1+1\cdot 2\cdot 1+1\cdot 2\cdot X+X\cdot 1+X\cdot 2\cdot 1+X\cdot 2\cdot X$

Étape 3 On va maintenant utiliser l'associativité de la multiplication pour transformer un produit de constantes et de variables en un couple (c,d), tel que l'expression soit égale à cX^d. Par exemple :
$X\cdot 2.\cdot 1.$
devient
$(2.,1)$

Étape 4 On a obtenu une expression qui est une somme de termes polynômiaux. Parcourir cette expression et construire une liste de couples (c,d) correspondant aux termes de la somme. Ainsi,
$1\cdot 1+1\cdot 2\cdot 1+1\cdot 2\cdot X+X\cdot 1+X\cdot 2\cdot 1+X\cdot 2\cdot X$
correspond à (pas forcément dans cet ordre)
`[1.,0 ; 2.,0 ; 2.,1 ; 1.,1 ; 2.,1 ; 2.,2]`

Étape 5 Construire une fonction qui à un couple associe sa primitive (sous la forme d'une expression mathématique) en remarquant que la primitive de (c,d) est $cX^{d+1}/(d+1)$. On pensera d'ailleurs à l'exponentiation rapide pour construire X^{d+1}. Ainsi,
`2.,2`
devient :
`Mul(2./.3.,Mul(Var,Mul(Var,Var)))`

Étape 6 Écrire une fonction qui utilise `list_map` pour créer une liste des primitives, et une fonction qui prend cette liste et construit la somme de ses éléments. Par exemple,
`[1.,0 ; 2.,0 ; 2.,1 ; 1.,1 ; 2.,1 ; 2.,2]`
devient :
$1X/1+2X/1+2X\cdot X/2+1X\cdot X/2+2X\cdot X/2+2X\cdot X\cdot X/3$

Le problème est résolu : on a réussi à trouver une primitive de l'expression initiale. S'en convaincre en intégrant puis dérivant un polynôme et en calculant sa valeur en assez de points.

13.5 Correction

Question 1 : On va calculer la suite de Fibonnaci de manière très peu maligne puis on va compter combien d'années il nous faudra attendre un résultat...

Exercice 13.5.1: *Fibonnaci récursif*

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

Hum...Alors la complexité : on peut s'apercevoir que, en notant u_n la valeur de la suite de fibonnaci et $A(n)$ le nombre d'appels récursifs qui sont effectués par la machine lorsqu'on lui demande de calculer u_n. On remarque que

$$\begin{array}{rrcl} & A(n) & = & 1 + A(n-1) + A(n-2) \\ ie & \dfrac{A(n)+1}{2} & = & \dfrac{A(n-1)+1}{2} + \dfrac{A(n-2)+1}{2} \end{array}$$

et comme $\dfrac{A(0)+1}{2} = \dfrac{A(1)+1}{2} = 1$ on a $u_n = \dfrac{A(n)+1}{2} \iff A(n) = 2u_n - 1$. On a donc une complexité exponentielle alors qu'une version avec une boucle *for* permettrait de le faire en tempslinéaire, et mieux encore, une version récursive plus maligne permet de le faire en tout juste n opérations :

Code CAML : *Une version plus astucieuse...*

```
# let rec fibo k=
    let rec harry = function
     |0|1-> (1,1)
     |n-> let (a,b) = harry (n-1) in
            (a+b,a)
     in
    fst(harry k);;
```

Question 2 : Maintenant voici l'algorithme implémenté en toutes les langages(enfin peut être pas en *Caml*...) : l'exponentiation rapide, ou comment calculer une puissance de x en environ $\log_2 x$.

Exercice 13.5.2:

```
# let rec exprap x k =
       let a = (x mod 2) in
       if k = 0 then 1
       else if k = 1 then x
       else let u = exprap x (k/2) in
            u*u*(exprap x a);;
```

Question 3 : Cette fois-ci on va chercher les antécédents d'un terme de la suite de Syracuse.

Exercice 13.5.3:

```
# let rec anté n x =
    if n > 0 then
       (  print_int (2*x);
          print_newline();
          anté (n-1) (2*x);
          if (x mod 2 = 0) && ((x-1) mod 3 = 0 )then
            (  let u = (x-1)/3 in
               if ((u mod 2) = 1) then
                  (  print_int u;
                     print_newline();
                     anté (n-1) u  )
             )
        )
;;
```

Question 4 : Où l'on apprend à faire une liste des diviseurs d'un entier.

Exercice 13.5.4:

```
#let petitdiv n =
    let bool =ref true and j = ref 1 in
    while !bool do
       incr(j);
       bool:=(n mod !j <> 0);
    done;
    !j;;

#let rec diviseurs = function
    |1->[]
    |n-> let u = petitdiv n in
            u::(diviseurs (n/u));;
```

Question 5 : Quelques opérations sur les listes.

Exercice 13.5.5:

```
# let rec prodliste = function
    |[]->1
    |t::q -> t*(prodliste q);;

#let rec sommeliste = function
    |[]-> 0
    |t::q -> t+sommeliste q;;
```

Question 6 : Une fonction bien utile :

Exercice 13.5.6:

```
# let rec listmap f  = function
    |[]->[]
    |t::q-> f(t)::(listmap f q);;
```

Question 7 : Un tri pour les listes.

Exercice 13.5.7: *Tri par insertion*

```
# let rec insert e = function
    |[]-> [e]
    |t::q -> if e<=t then e::t::q
             else t::(insert e q);;
# let rec tri = function
    |[]->[]
    |t::q-> insert t (tri q);;
```

Question 8 : Maintenant les types définis par nos soins !

Exercice 13.5.8:

```
# let rec affichage = function
    |Var -> print_char ‘x‘
    |Const (a) -> print_float a
    |Mul(b,c) ->   (print_string"(";
                    affichage b;
                    print_string "*";
                    affichage c;
                    print_string")")
    |Add (d,e) ->  (print_string "(";
                    affichage d;
                    print_string "+";
                    affichage e;
                    print_string ")")
    |Sub (f,g) ->  (print_string "(";
                    affichage f;
                    print_string "-";
                    affichage g;
                    print_string")");;

# let rec rempl valeur = function
    |Var-> Const valeur
    |Const c -> Const c
    |Mul (b,c) -> Mul (rempl valeur b, rempl valeur c)
    |Add (d,e) -> Add (rempl valeur d, rempl valeur e)
    |Sub (f,g) -> Sub (rempl valeur f, rempl valeur g);;

# let rec calcul = function
    |Var -> failwith "T’y es fou"
    |Const a -> a
    |Mul (b,c) -> calcul b *. calcul c
    |Add (d,e) -> calcul d +. calcul e
    |Sub (f,g) -> calcul f -. calcul g;;

# let rec dériv = function
   |Var -> Const 1.
   |Const a -> Const 0.
   |Mul (b,c) -> Add(Mul(dériv b, c),Mul(b,dériv c))
   |Add (d,e) -> Add(dériv d,dériv e)
   |Sub (f,g) -> Sub(dériv f, dériv g);;
```

Question 9 : On va donner les deux plus grands éléments d'une liste :

Exercice 13.5.9:
```
# let podium l =
   let rec duo premier deuxième = function
         |[]->(premier, deuxième)
         |t::q-> duo (max premier t) (max (min premier t) deuxième) q
    in
   match l with
      |[]|[_]-> failwith "Ta liste est trop courte..."
      |t::r::q -> let (a,b) = duo (max t r) (min t r) q in
                  b;;
```

Question 10 : Inversons une liste maintenant :

Exercice 13.5.10: *Inversion d'une liste*
```
# let rev l =
    let rec reverse autreliste = function
        |[] -> autreliste
        |t::q -> reverse (t::autreliste) q
    in

    reverse [] l;;
```

Question 11 : Enfin nous allons intégrer les plynômes de la question 8 !

Code CAML :
```
# let rec plusdesoustraction = function
    |Var ->Var
    |Const a -> Const a
    |Mul (b,c) -> Mul (plusdesoustraction b, plusdesoustraction c)
    |Add (d,e) -> Add (plusdesoustraction d, plusdesoustraction e)
    |Sub (f,g) -> Add (plusdesoustraction f,
                       Mul(Const (-1.), plusdesoustraction g));;

#let rec développement = function
    |Var->Var
    |Const a -> Const a
    |Mul(Add(b,c),d) -> Add(Mul(b,d),Mul(c,d))
    |Mul(e,Add(f,g)) -> Add(Mul(e,f),Mul(e,g))
    |Mul(h,i) -> Mul(h,i)
    |Add(j,k)->Add(j,k)
    |Sub(l,m) -> Sub(l,m);;

# let rec puissances = function
    |Mul(a,b)-> let (a',a'') = puissances a
                and (b',b'') = puissances b in
                ((a'*.b'),a''+b'')
    |Const c -> (c,0)
    |Var -> (1.,1);;
```

Exercice 13.5.11: *Intégration !*

```
#let rec transfo = function
    |Add(a,b) -> (transfo a)@(transfo b)
    |n -> [puissances n];;

#let int (a,b) =
    (a/.(float_of_int(b+1)),b+1);;

#let primit (a,b) =
    let (c,d) = int (a,b) in
    let rec monome = function
        |0-> Const 1.
        |1 -> Var
        |n-> Mul(Var,monome (n-1))
    in
    let x = monome d in
    Mul(Const c, x);;

#let integr l =
    let s = list__map primit l in
    let rec somme = function
        |[]-> Const 0.
        |t::[] -> t
        |r::q -> Add(r,somme q)
    in
    somme s;;

# let intégration x =
    integr(transfo(développement(plusdesoustraction(x))));;
```

CHAPITRE 14

TP4 : Ensembles, chaînes, types produit

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante. Les corrigés seront mis en ligne peu après. Tous ces documents se trouveront à l'adresse suivante :

```
http ://www.nicollet.net/colles/
```

14.1 Ensembles

De nombreux énoncés de concours font intervenir la notion d'ensemble. Le plus souvent, il s'agit de travailler sur les sous-parties de $\{0 \dots N\}$. On supposera dans la suite que N est une constante dont la valeur est péremptoirement fixée à $N = 1000$

Il existe deux manières canoniques de représenter des ensembles : par des listes triées, et par des tableaux de booléens, chacune ayant des avantages et des inconvénients.

14.1.1 Listes triées

On considère qu'un ensemble est représenté par la liste croissante des entiers qu'il contient. Par exemple, $\{6, 3, 5, 1\}$ correspond à la liste `[ 1 ; 3 ; 5 ; 6 ]`.

Question 1

Écrire avec des listes triées les propriétés usuelles qu'on peut vouloir connaître sur des ensembles (appartenance d'un élément, inclusion) ainsi que les opérations usuelles (union, intersection). Quelles sont leurs complexités en fonction de la taille de l'ensemble représenté ?

14.1.2 Tableaux

On considère qu'un ensemble est représenté par un tableau de booléens. L'élément i du tableau est `true` si et seulement si i appartient à l'ensemble. Par exemple, si $N = 4$, l'ensemble $\{3, 1\}$ est représenté par le tableau `[| false ; true ; false ; true ; false |]`.

Question 2

Écrire les propriétés et opérations usuelles des ensembles avec des tableaux. Quelles sont leurs complexités en fonction de N ?

14.1.3 Comparatif

Les tableaux sont les plus rapides pour ce qui est de déterminer l'appartenance d'un élément. Cependant, ils sont plus lents que les listes sur toutes les autres opérations, et consomment plus de mémoire. Au final, il est préférable de n'utiliser la représentation par tableaux que si les tests d'appartenance sont beaucoup plus fréquents que n'importe quelle autre opération.

14.2 Chaînes

Une chaîne de caractères est un bout de texte manipulable par `Caml Light`, par exemple `"Bonjour !"`. En pratique, une chaîne `s` se comporte comme un tableau de caractères de longueur `string_length s`, à cela près qu'au lieu de parenthèses, il faut utiliser des crochets :

```
# let texte = "Bonjour !" in
texte.[2] ; ;
- : char = ‘n‘
```

Question 3

Écrire une fonction qui retourne une chaîne.

Par exemple, `"abracadabra"` devient `"arbadacarba"`.

On peut aussi, par exemple, créer la chaîne `"xxxxx"` avec la fonction `make_string 5 ‘x‘`. Attention aux guillemets des caractères, ce sont des guillements simples inversés. Enfin, on peut concaténer deux chaînes avec l'opérateur ^ comme dans l'exemple suivant :

```
# let n = 14 in "Bonjour, il est "^(string_of_int n)^" heures" ; ;
- : string = "Bonjour, il est 14 heures"
```

Question 4

Écrire une fonction qui prend en argument deux chaînes (un texte et un motif) et renvoie la liste des indices auxquels le motif apparaît dans la chaîne.

Par exemple, `find("ana")("ananas")` renvoie `[ 0, 2 ]`.

Il est également possible de se servir du pattern matching pour reconnaître tout un intervalle de valeurs de caractères (minuscules, majuscules, chiffres), comme par exemple :

```
match ‘c‘ with
| ‘a‘ .. ‘z‘ -> print_endline "Minuscule !"
| ‘A‘ .. ‘Z‘ -> print_endline "Majuscule !"
| ‘0‘ .. ‘9‘ -> print_endline "Chiffre !"
| _ -> ( ) ; ;
```

Question 5

La fonction suivante transforme une majuscule en une minuscule :

```
let tolower c = char_of_int (int_of_char c + 32) ; ;
```

Modifier ou réutiliser la fonction `find` de la question précédente pour qu'elle ne trouve que les mots entiers. On supposera les mots délimités par des espaces et de la ponctuation (`- ! ? . ; : ,`) qui sera typographiquement correcte (les phrases commencent par une majuscule et se finissent par un point). De plus, le mot à rechercher sera un vrai mot : il ne contiendra que des lettres. On pourra, éventuellement, modifier les chaînes reçues en entrée avant de faire la recherche.

Par exemple, `find2("ananas")("What ? Ananas ? No, bananas !")` renvoie la liste `[ 6 ]` (il trouve "Ananas" mais pas "bananas").

14.3 Types produit

14.3.1 Introduction

En plus des types somme que nous avons vus la semaine dernière, on peut définir en `Caml Light` de nouveaux types produits. Le principe est de créer un objet comportant plusieurs sous-objets (appelés *labels*), chacun ayant une valeur donnée. Un exemple canonique est :

```
type person =
  {
    name : string ;
    age : int ;
  } ; ;
```

Il est alors possible de créer un objet de type `person` en donnant à chacun des labels une valeur.

```
let douglas =
  { name = "Douglas Adams" ; age = 42 } ; ;
```

Les labels d'un objet peuvent être lus de la même manière que les éléments d'une chaîne ou d'un tableau. Par exemple, `douglas.age` renvoie `42`. `Caml Light` comprend lorsqu'on utilise cette expression que `douglas` est de type `person` (par conséquent, le nom d'un champ ne peut pas être utilisé par plusieurs types).

Question 6

Construire un type `complex` représentant un nombre complexe par ses parties réelles et imaginaires. Écrire des fonctions qui ajoutent, multiplient et affichent des nombres complexes.

Considérons l'exemple suivant :

```
type ordinal = { incr : unit -> unit ; print : unit -> unit } ; ;

let ordinal ( ) =
  let valeur = ref 0 in
  { incr = fun ( ) -> incr valeur ; print = fun ( ) -> print_int !valeur } ; ;
let un = ordinal( ) in un.incr ( ) ; un.print( ) ; ;
```

Dans cet exemple, l'utilisateur dispose d'un objet dont il peut augmenter la valeur de un, et qu'il peut afficher. Le fonctionnement de ces deux actions est défini grâce à la référence `valeur`, qui est utilisée par les deux fonctions en question mais n'est pas accessible à l'utilisateur. Il est alors possible de définir le fonctionnement d'une autre manière (qui peut être plus rapide, ou moins gourmande en mémoire) sans que l'utilisateur ait à changer son code (puisque de son point de vue, l'objet n'aura pas changé).

Question 7

On donne le type suivant :

```
type stack = {
  push : int -> unit;
  pop : unit -> unit;
  size : unit -> unit;
}
```

On souhaite que la fonction `push` empile son argument, que la fonction `pop` dépile le sommet de la pile (ou renvoie 0 si la pile est vide), et que la fonction `size` renvoie le nombre d'éléments empilés et non encore dépilés.
Écrire une fonction qui renvoie un objet de type `stack` qui fonctionne correctement, et le tester.

14.3.2 Labels mutables

On peut rendre certains labels mutables : il devient alors possible de modifier leur valeur après la création de l'objet (comme les cases d'un tableau ou d'une chaîne) :

```
type my_ref = { mutable content : int };;
let x = { content = 10 };;
x.content <- x.content + 1;;
```

Question 8

Créer un type pour représenter un compte en banque, donné par son solde en cents et le nom de son propriétaire.
Créer un tableau de tels objets, et écrire :
- Une fonction qui affiche pour chaque compte du tableau son propriétaire et son solde.
- Une fonction qui ajoute ou retire de l'argent à un compte i.
- Une fonction qui effectue un transfert d'un compte i à un compte j.

14.3.3 Types paramétrés

On peut créer une pile d'entiers (comme dans la question 7), mais il faudrait le réécrire à l'identique pour obtenir une pile de réels, de complexes, de listes, de tableaux, de fonctions et ainsi de suite, ce qui serait une perte de temps. Cela est résolu grâce à des types paramétrés, dont on a déjà vu quelques exemples comme `bool ref`, `float vect` ou `int list`. On peut paramétrer un type par un autre type de la manière suivante :

```
type 'a my_ref = { mutable c : 'a };;
```

Dans ce cas, l'objet `{ c = false }` sera de type `bool my_ref`, et l'objet `{ c = 1 }` sera, lui, de type `int my_ref`, permettant de réutiliser le type `my_ref` (et toutes les fonctions s'appliquant sur lui sans faire d'hypothèse sur le type de son contenu) pour n'importe quel type `'a`.

On peut donner plusieurs arguments de type de la manière suivante :

```
type ('a,'b) my_pair = { fst : 'a; snd : 'b };;
```

Enfin, les types somme peuvent eux aussi être paramétrés :

```
type 'a my_list =
  | Vide
  | TeteQueue of 'a * 'a my_list
```

14.4 Pour finir

Question 9

Un quine[a] est un programme dont le seul effet est d'afficher son propre code source. Il a été prouvé que n'importe quel langage de programmation Turing-complet pouvait être utilisé pour écrire un quine, c'est donc en particulier le cas de `Caml Light`.

Écrire un quine en `Caml Light`. Ne vous inquiétez pas, c'est toujours un peu douloureux la première fois, n'hésitez donc pas à demander des indications.

[a] nommé d'après le mathématicien Willard Van Orman Quine

14.5 Correction

Question 1 : Les fonctions habituelles des ensembles sur les listes :

Exercice 14.5.1:

```
# let rec lst_appartient (x) = function
| [] -> false
| h::t -> (h = x) || (h < x) && (lst_appartient x t);;

#let rec lst_union (a) (b) =
match (a,b) with
| ([],t) -> t
| (t,[]) -> t
| (ha::ta),(hb::tb) ->
if ha > hb then hb::(lst_union a tb)
else ha::(lst_union ta b);;

# let rec lst_inter (a) (b) =
match (a,b) with
| ([],_) | (_,[]) -> []
| (ha::ta),(hb::tb) ->
if ha = hb then ha::(lst_inter ta tb)
else if ha < hb then lst_inter ta b
else lst_inter a tb;;

#let lst_inclus (a) (b) =
(lst_inter a b) = a;;
```

Question 2 : Et cette fois ci sur les tableaux !

Exercice 14.5.2:

```
#let vec_appartient (x) (e) = e.(x);;

# let vec_union t s =
   let n = vect_length t and m = vect_length s in
   let u = make_vect (max m n ) false in
   for i = 0 to pred(min n m) do
       u.(i)<-(t.(i))||s.(i);
       done;
   if m>n then
       (for i = n to m-1 do
       u.(i)<-s.(i);
       done)
   else if n>m then
       (for i = m to pred(m) do
        u.(i)<-t.(i);
        done);
   u;;
```

Code CAML :

```
#let vec_inter (a) (b) =
let c = copy_vect (a) in
for i = 0 to min (vect_length a) (vect_length b) - 1 do
  c.(i) <- c.(i) && b.(i)
done;
c;;

#let vec_inclus (a) (b) =
   if vect_length a >vect_length b then false
   else
        (let inclus = ref true in
for i = 0 to vect_length a - 1 do
if (vec_appartient i a) && (not vec_appartient i b) then inclus := false
done;
!inclus);;
```

Question 3 : Retournons une chaine de caractères pour voir...

Exercice 14.5.3: *Tourne toi ...*

```
#let benoit x =
    let n = string_length x in
    for i = 0  to (n-1)/2 do
      let u = x.[n-1-i] in
      x.[n-1-i]<-x.[i];
      x.[i]<-u;
    done;
    x;;
```

Question 4 : Cette fois-ci on va rechercher des motifs dans un texte !

Exercice 14.5.4: *Recherche de motifs*

```
# let find motif chaine =
    let n = string_length motif
    and m = string_length chaine
    and u  = ref [] in
    for i = m-n downto 0 do
       if sub_string chaine i n = motif then u:=i::(!u);
    done;
    !u;;
```

Question 5 : Et maintenant on recherche dans mots uniquement !

Exercice 14.5.5: *Rechercher de mots*

```
# let tolower x = char_of_int (int_of_char x + 32);;

#let transform (str) =
for i = 0 to string_length str - 1 do
str.[i] <- match str.[i] with
| ‘a‘ .. ‘z‘ as n -> n
| ‘A‘ .. ‘Z‘ as n -> tolower n
| n -> ‘§‘
done;
"§"^str^"§";;

# let find2 (sub) (str) =
find (transform sub) (transform str);;
```

Question 6 : On va de nouveau jouer avec les types non prédéfinis de *Caml*. Et cette fois-ci ce sera plus dur...

Exercice 14.5.6:

```
#type complex =

 Re : float;
 Im : float;
;;

#let affichage x =
    print_float x.Re;
    print_string "+";
    print_string "ix";
    print_float x.Im;;

#let sommec a b =
    Re = a.Re+.b.Re ; Im = a.Im+.b.Im;;

#let prodc a b =
   Re = a.Re*.b.Re -. a.Im*.b.Im ; Im = a.Re*.b.Im +. a.Im*.b.Re ;;
```

Question 7 : Et maintenant un objet un peu plus compliqué !

Exercice 14.5.7:

```
# type stack =

push : int -> unit;
pop : unit -> int;
size : unit -> int;
;;

#let make_stack ( ) =
let contents = ref []
and size = ref 0 in

push =
(fun x -> contents := x::(!contents); incr size);

pop =
(fun ( ) -> match !contents with
| [] -> 0
| h::t -> contents := t; decr size; h);

size =
(fun ( ) -> !size )
;;
```

Question 8 : Faisons un peu de banque maintenant[1]

Code CAML :

```
#type compte =
 mutable solde : float ;
  nom : string
;;

#let suisse =
   make_vect 100  solde = 1000000. ; nom = "crésus" ;;
   suisse.(0)<- solde = 1000000000. ; nom = "picsou";;

# let relevé t x =
   print_string ("Propriétaire : Mr "^(t.(x).nom));
   print_newline();
   print_string "solde : ";
   print_float t.(x).solde;
   print_string "euros ";;
```

[1] Et pourtant l'auteur n'est pas à l'X...

Exercice 14.5.8:

```
#let versement somme compte banque =
   banque.(compte).solde <-banque.(compte).solde +. somme;
   print_string ("Mr "^(banque.(compte).nom)^" vient de ");
   if somme >=0. then print_string ("recevoir "^(
   string_of_float somme)^" euros")
   else print_string ("subir un prélèvement de "^(
   string_of_float (-1.*.somme))^" euros");;

#let transfert banque compte1 compte2 montant =
   banque.(compte1).solde <- banque.(compte1).solde -. montant;
   banque.(compte2).solde <- banque.(compte2).solde +. montant;
    print_string ("Mr "^(banque.(compte1).nom)^" vient de verser "^(
    string_of_float montant)^"veuros à Mr "^(banque.(compte2).nom));;
```

Question 9 : Pour finir on va tenter de réaliser une quine...Voilà la correction !

```
(* C'est difficile d'imaginer comment écrire un quine, la première fois.
   Mais une fois qu'on a compris, cela devient très facile.

   Le problème est évident quand on y réfléchit quelques instants:

print_endline "print_endline \"print_endline \\\"print_endline ...

   Il faut donc contourner cela en séparant d'un côté le programme lui-même,
   et d'un autre côté les données à afficher. On parle souvent de

quine "quine"

   Où le programme quine appliqué à une chaîne "x" affiche:

quine "x" ----> x "x"

  Il suffit alors d'écrire le code source de quine à la place du x,
  et on aura obtenu ce qu'on cherche. Première version:

Code:     (function x -> print_endline (x^" ""^x^"";;"))
Données: "(function x -> print_endline (x^" ""^x^"";;"))"

  Petit problème: les guillements sont mal interprétés! Pour afficher des
  guillements dans un langage de programmation, on utilise \" (le symbole
  \ sert à afficher le caractère qui le suit sans essayer de l'interpréter).
  On aboutit alors à une deuxième version:

Code:     (function x -> print_endline (x^" \""^x^"\";;"))
Données: "(function x -> print_endline (x^\" \"\"^x^\"\";;\"))"

  Affichage:

Code:     (function x -> print_endline (x^" ""^x^"";;"))
Données: "(function x -> print_endline (x^" ""^x^"";;"))"

  Nous approchons du but, mais il reste deux problèmes: d'abord,
  le code généré est incorrect (puisque le \" est devenu un ");
```

```
  Pour cela, on écrit: \\\" (\\ puis \", qui devient \ puis ", soit \").

Code:     (function x -> print_endline (x^" \""^x^"\";;"))
Données: "(function x -> print_endline (x^\" \\\"\"^x^\"\\\";;\"))"

  Le deuxième problème vient du fait que le processus supprime
  automatiquement les \ des données. Pour reconstruire les données
  dans leur état initial, il faut donc reconstruire les \. Caml Light
  propose une fonction, string_for_read, qui fait exactement ça.

  On aboutit alors à la version finale:

Code:     (function x -> print_endline (x^" \""^(string_for_read x)^"\";;"))
Données: "(function x -> print_endline (x^\" \\\"\"^(string_for_read x)^\"\\\";;\"))"

  On peut vérifier que ça fonctionne: *)

(function x -> print_endline (x^" \""^(string_for_read x)^"\";;"))
 "(function x -> print_endline (x^\" \\\"\"^(string_for_read x)^\"\\\";;\"))";;

(* Maintenant, on peut commencer à améliorer ça. D'abord, bien sûr, on retire
   les espaces inutiles et on remplace "function" par son abbréviation,
   "fun" *)

(fun x->print_endline(x^"\""^(string_for_read x)^"\";;"))
"(fun x->print_endline(x^\"\\\"\"^(string_for_read x)^\"\\\";;\"))";;

(* C'est déjà mieux, mais on peut faire encore plus court en utilisant
   la fonction printf__printf. Celle-ci prend un premier argument qui est
   une chaîne de format. Par exemple:

printf__printf "%s -- %s" "foo" "bar";; affiche: foo -- bar

   Cela aura pour effet de raccourcir les concaténations et
   parenthèses nécessaires pour print_endline: *)

(fun x->printf__printf"%s\"%s\";;\n"x(string_for_read x))
"(fun x->printf__printf\"%s\\\"%s\\\";;\\n\"x(string_for_read x))";;

(* On a gagné un caractère, c'est maigre. Malheureusement, il n'y a
   pas grand-chose de plus qu'on puisse faire. Pour la curiosité, voici
   un quine qui utilise let au lieu de fun : *)

let x="in printf__printf\"let x=\\\"%s\\\"%s;;\\n\"(string_for_read x)x"in
printf__printf"let x=\"%s\"%s;;\n"(string_for_read x)x;;

(* Enfin, pour en savoir plus sur les quines (et une explication plus
   claire que la mienne), n'hésitez pas à aller embêter David Madore:

   http://www.madore.org/~david/computers/quine.html

   *)
```

CHAPITRE 15

TP5 : graphics, dessin, géométrie plane.

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante. Les corrigés seront mis en ligne peu après. Tous ces documents se trouveront à l'adresse suivante :

`http ://www.nicollet.net/colles/`

Les fonctions nécessaires pour dessiner se trouvent dans une librairie externe qui n'est pas chargée initialement : il faut donc demander au terminal `Caml Light` de charger cette librairie. Le code ci-dessous effectue le chargement et ouvre une fenêtre graphique de 640 par 480 pixels (attention au `#` supplémentaire à la première ligne) :

```
# #open "graphics";;
# open_graph "640x480";;
- : unit = ()
```

15.1 Couleurs

La fonction la plus simple est la fonction `plot x y`, qui modifie la couleur du pixel de coordonnées (x, y). Le pixel $(0, 0)$ se trouve en bas à gauche de l'écran. Pour indiquer quelle couleur utiliser, on utilise la fonction `set_color c`, qui dit au système d'utiliser la couleur c pour tous les dessins jusqu'à ce qu'on lui demande à nouveau de la changer.

Pour effacer tout ce qui se trouve à l'écran, on utilise la fonction `clear_graph ( )`.

On définit une couleur par un mélange de rouge, vert et bleu grâce à la fonction `rgb r v b`, où (r, v, b) sont des entiers entre 0 et 255 indiquant la quantité de chacune des couleurs. Certaines ont déjà un nom.

couleur	`Caml Light`	R	V	B	couleur	`Caml Light`	R	V	B
noir	`black`	0	0	0	blanc	`white`	255	255	255
rouge	`red`	255	0	0	vert	`green`	0	255	0
bleu	`blue`	0	0	255	jaune	`yellow`	255	255	0
magenta	`magenta`	255	0	255	cyan	`cyan`	0	255	255
orange	non défini	255	128	0	gris	non défini	128	128	128

Par exemple, pour dessiner en orange :

```
set_color (rgb 255 128 0);;
```

Question 1

Avec les fonction décrites ci-dessus, dessiner un carré de taille 256 par 256 en dégradé de couleurs, avec dans chaque coin (de gauche à droite et de haut en bas) : du cyan, du blanc, du bleu, du magenta.

Indication : utiliser deux boucles `for` et se servir des indices de boucle pour définir les couleurs.

15.2 Segments

15.2.1 Dessin

`Caml Light` utilise pour les segments un crayon, qui se déplace sur la fenêtre. Pour déplacer le crayon levé, on utilise la fonction `moveto x y`, et pour déplacer le crayon posé (donc, en dessinant un trait) on utilise la fonction `lineto x y`. Par exemple, la fonction ci-dessous dessine un trait entre (x_1, y_1) et (x_2, y_2) à l'échelle 10 : 1.

```
let draw (x1,y1) (x2,y2) =
  moveto (x1*10) (y1*10) ;
  lineto (x2*10) (y2*10) ; ;
```

Question 2

Écrire une fonction `make_points w h n` qui crée un tableau de `n` points (x, y) pris au hasard[a] avec $0 \leq x < w$ et $0 \leq y < h$.

Écrire[b] une fonction `complet t` qui prend en argument un tableau de points et dessine pour chaque couple de points le segment reliant ces deux point, à l'échelle 10 : 1.

[a] penser à `random__int`
[b] et tester

Question 3

La fonction `do_list f l` de `Caml Light` applique la fonction `f` aux éléments de la liste `l`, dans l'ordre. En déduire une fonction `dessine t l` qui pour un tableau de points `t` et une liste de couples d'indices `l` dessine le segment entre les points `t.(i)` et `t.(j)` pour tout couple `(i,j)` de `l`.

Modifier la fonction `complet` pour créer la fonction `segments` qui renvoie la liste des couples de points à dessiner, dans un ordre quelconque. Par exemple, `segments [| (10,3) ; (4,2) ; (2 ;13) |]` donne `[ (0,1) ; (0,2) ; (1,2) ]`.

Créer un tableau `t` et tester `dessine t (segments t)`. L'objectif de cet exercice est d'obtenir une liste de segments pour pouvoir travailler dessus (`segments`) et dessiner les segments restants une fois le travail fini (`dessine`).

Important : un segment est un couple constitué des indices des deux extrémites. Pour pouvoir travailler sur les segments, il faut disposer à la fois de la liste des segments et du tableau des points.

15.2.2 Enveloppe convexe

On ne souhaite garder que les segments qui font partie de l'enveloppe convexe, à savoir les côtés du plus petit polygone qui contient tous les points du tableau.

Si A, B et C sont des points du plan, on définit la formule suivante :

$$\mathcal{C}(A,B,C) = sign\big((C_x - A_x)(B_y - A_y) + (C_y - A_y)(A_x - B_x)\big)$$

où $sign(x)$ vaut -1, 0 et 1 suivant que x est négatif, nul ou positif (respectivement). Cette formule renvoie -1 si C est à gauche de la droite AB, 1 s'il est à droite, et 0 s'il est dessus.[1]

Question 4

Programmer la fonction $\mathcal{C}$, et écrire une fonction `sameside t i j` qui détermine si tous les points de `t` sont du même côté du segment `t.(i) t.(j)`.

Sachant que si un segment fait partie de l'enveloppe convexe, alors tous les points du tableau sont du même côté de ce segment, en déduire une fonction `hull t l` qui prend en entrée une liste de segments et un tableau et renvoie la liste des segments qui, parmi ceux-ci, font partie de l'enveloppe convexe.

Essayer `dessine t (hull t (segments t))` pour afficher l'intégralité de l'enveloppe convexe.

15.2.3 Triangulation

Trianguler un ensemble de points signifie tracer un à un des segments entre les points, jusqu'à ce qu'il ne reste plus aucun segment à tracer qui n'intersecte pas au moins un autre segment. Le résultat de cette opération est un pavage de l'enveloppe convexe par des triangles dont les sommets sont les points de l'ensemble.

Ici, on va utiliser un algorithme récursif A peu efficace mais simple à écrire. Il prend en argument un ensemble de segments quelconque, et renvoie un ensemble de segments ne s'intersectant pas, de la manière suivante :

- $A(\emptyset) = \emptyset$.
- $A(\{a\} \cup S) = \{a\} \cup A(\rho_a(S))$ où l'on note $\rho_a(S)$ l'ensemble des segments de S n'intersectant pas a.

Question 5

Écrire une fonction `inter a b c d` qui détermine si les segments `a b` et `c d` s'intersectent (s'ils partagent un sommet, ce n'est pas une intersection). Se servir de la fonction $\mathcal{C}$ vue précédemment.

En déduire une fonction `retire t (i,j) l` qui calcule ρ (autrement dit, retire de `l` les segments qui intersectent le segment `t.(i) t.(j)`).

Implémenter par une fonction récursive `sup_inter t l` l'algorithme A, qui renvoie une liste de segments entre les points de `t` ne s'intersectant pas.

Trianguler un ensemble `t` de points par :
`dessine t (sup_inter t (segments t))`

---

[1] La notion de gauche et droite dépend de l'orientation, trigonométrique ou anti-trigonométrique, que l'on a choisie pour le plan. Le choix n'est pas vraiment important ici.

Cette approche simpliste donne un résultat peu esthétique, car les triangles sont trop aplatis. Il existe des méthodes complexes pour construire une belle triangulation (par exemple, une triangulation de Delaunay).

Nous allons cependant ici utiliser une méthode simple : plus un segment est proche de la tête de la liste, plus il a de chances d'apparaître dans la triangulation. Si l'on place les segments les plus courts d'abord, les triangles ne seront en général pas aplatis.

On rappelle que la longueur d'un segment AB dans le plan est égale à

$$\sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}$$

De plus, `Caml Light` dispose d'une fonction de tri `sort__sort f l`. La fonction `f` décrit l'ordre à employer, autrement dit `f a b = true` $\Longleftrightarrow$ `a <= b`. La fonction de tri renvoie la liste `l` triée suivant l'ordre décrit par `f`.

Question 6

Écrire une fonction `long t i j` qui renvoie le carré de la longueur du segment `t.(i) t.(j)` [a]

Écrire une fonction `sort t l` qui trie par taille croissante la liste `l` de segments de `t`. Ne pas réinventer la roue et utiliser la fonction `sort__sort` de `Caml Light`.

Tracer : `dessine t (sup_inter t (sort t (segments t)))`

[a] question auxiliaire : pourquoi le carré ?

15.3 Interactivité

La fonction `read_key( )` renvoie le caractère correspondant à la touche qui est pressée. Lorsque cette fonction est appellée, l'exécution est interrompue jusqu'à la pression d'une touche.

La fonction `mouse_pos( )` renvoie immédiatement la position de la souris dans la fenêtre graphique.

Pour déterminer si une touche est pressée sans attendre qu'elle le soit, on peut utiliser l'expression suivante, qui est vraie si et seulement si une touche est pressée :

```
(wait_next_event [ Poll ; Key_pressed ]).keypressed
```

Question 7

Créer un tableau de 640 réels appellé `terrain` et avec pour valeur initiale 50. Écrire une fonction `vert i` qui trace un trait noir vertical d'abscisse `i` et de hauteur `terrain.(i)`, et un trait blanc de même abscisse au-dessus, de hauteur `480.-.terrain.(i)`.

Écrire une fonction `even i` qui calcule la différence δ entre `terrain.(i)` et `terrain.(i-1)`, puis qui augmente le plus petit des deux de $|\delta/2|$ et diminue le plus grand des deux de $|\delta/2|$ (ignorer les cas où $|\delta| \leq 1$).

Écrire une fonction `tick()` qui appelle les fonction `even` et `vert` sur toutes les cases du tableau (sauf la case 0).

Écrire une boucle infinie qui appelle `tick()`, et qui vérifie si une touche est pressée.
- Si la touche `'q'` est pressée, la boucle s'interrompt avec `failwith "Fin"`.
- Si la touche espace est pressée, et si le curseur est de coordonnées (x, y), on modifie `terrain.(x)` pour qu'il soit égal à y.
- On ne fait rien si c'est une autre touche.

15.4 En apprendre plus

Il est conseillé d'étudier la documentation de la librairie `graphics` si vous avez l'intention de vous en servir : elle dispose de plus de fonctions qu'il n'est possible d'en étudier en deux heures.

`http ://caml.inria.fr/pub/docs/manual-caml-light/node16.html`

15.5 Correction

Question 1 : On va d'abord donner une fonction qui permet de répondre à la question puis on va la compléter en un cube...

Exercice 15.5.1:

```
#for i = 0 to 255 do
for j = 0 to 255 do
set_color (rgb i j 255);
plot i j
done
done;

#for i = 0 to 255 do
for j = 0 to 180 do
let j' = j * 1414/1000 in
set_color (rgb i 255 (255-j'));
plot (i+j) (j+255)
done
done;
for i = 0 to 180 do
for j = 0 to 255 do
let i' = i * 1414/1000 in
set_color (rgb 255 j (255-i'));
plot (i+255) (i+j)
done
done;;
```

Question 2 : On va se donner un programme qui permet de créer des tableaux de points aléatoires et pour vérifier que cela fonctionne on va le tester sur un programme qui trace des segments en les agrandissant 10 fois.

Exercice 15.5.2:

```
# let make_points w h n =
   let t = make_vect n (0,0) in
   for i = 0 to n-1 do
      t.(i)<-(random__int(w),random__int(h));
   done;
   t;;

#let draw (x1,y1) (x2,y2) k =
   moveto (x1*k) (y1*k);
   lineto (x2*k) (y2*k);;

let complet t =
   let n = vect_length t in
   for i = 0 to (n-2) do
      for j = succ(i) to n-1 do
         draw (t.(i)) (t.(j)) 10;
      done;
   done;;
```

Question 3 : Il faut créer une fonction qui permet de tracer des points donnés par une liste de couples de points et le tableau correspondant.

Exercice 15.5.3: *Tracé de segments*

```
# let dessine t l =
    do_list (function (a,b) -> draw t.(a) t.(b) 1) l;;

# let segments t =
   let n = vect_length t in
   let l = ref [] in
   for i = 0 to n-2 do
      for j = 1 to n-1 do
         l:=((i,j))::(!l);
      done;
   done;
   !l;;
```

Question 4 : On veut donner une enveloppe convexe à un ensemble de segments.

Exercice 15.5.4: *Enveloppe convexe*

```
# let C ((a,b),(c,d),(e,f)) =
   let x = (e-a)*(d-b)+(f-b)*(a-c) in
      if x = 0 then 0
      else if x> 0 then 1
      else -1;;

# let sameside t i j =
let n = vect_length t in
let c = ref 0 in
let e = ref 0 in
let all = ref true in
while !all && !e < n do
let s = C(t.(i),t.(j),t.(!e)) in
if !c * s < 0 then all := false
else (incr e; c := !c + s)
done;
!all;;

# let rec hull (t) = function
| [] -> []
| (i,j)::q -> if sameside t i j then (i,j)::(hull t q) else hull t q;;
```

Question 5 : On veut trianguler entièrement le plan :

Exercice 15.5.5:

```
# let inter a b c d =
(C(a,b,c)) * (C(a,b,d)) < 0 && (C(c,d,a)) * (C(c,d,b)) < 0;;

# let rec retire (t) (i,j) = function
| [] -> []
| (ai,aj)::q ->
if inter t.(i) t.(j) t.(ai) t.(aj) then retire t (i,j) q
else (ai,aj)::(retire t (i,j) q);;

# let rec sup_inter (t) = function
| [] -> []
| a::s -> a::( sup_inter t (retire t a s));;

# let triangule (t) =
dessine t (sup_inter t (segments t));;
```

Question 6 : Ce serait bien d'avoir de jolis triangles. . . !

Exercice 15.5.6: *De beaux triangles*

```
# let long (ax,ay) (bx,by) =
(ax - bx) * (ax - bx) + (ay - by) * (ay - by);;

# let ordre t (i1,j1) (i2,j2) =
long t.(i1) t.(j1) <= long t.(i2) t.(j2);;

#let sort t l = sort__sort (ordre t) l;;
```

Question 7 : Cette fois-ci on va utiliser des fonctions interactives de *Caml* !

Code CAML :

```
# let terrain = make_vect 640 50.;;

#let vert (i) =
set_color black;
moveto i 0;
lineto i (int_of_float terrain.(i));
set_color white;
lineto i 480;;

#let even (i) =
let delta = terrain.(i) -. terrain.(i-1) in
if delta *. delta > 1. then begin
terrain.(i) <- terrain.(i) -. delta*.0.5;
terrain.(i-1) <- terrain.(i-1) +. delta*.0.5;
end;;
```

Exercice 15.5.7: *Une goutte d'eau ?*

```
#let tick ( ) =
vert 0;
for i = 1 to vect_length terrain - 1 do
vert i;
even i;
done;;

# while true do
tick ( );
if (wait_next_event [Poll; Key_pressed]).keypressed then
match read_key ( ) with
| 'q' -> failwith "Fin"
| ' ' -> let (i,j) = mouse_pos ( ) in if i >= 0 && i < 640 then
             terrain.(i) <- float_of_int j;
| _ -> ( )
done;;
```

Code CAML : *Et une version plus compliquée...*

```
#clear_graph ( );;

let w = 640;;(*
let a = 0.5;;
let b = 0.5;;
let c = 0.97;;
let m = 20;; *)


let a = 0.3;;
let b = 1.0;;
let c = 0.97;;
let m = 20;;

let terrain = make_vect w 300.;;
let deriv = make_vect w 0.;;

let vert (i) =
set_color black;
moveto i 0;
lineto i (int_of_float terrain.(i));
set_color white;
lineto i 480;;
```

Code CAML :

```
# let diffc (i) =
let s = ref 0. in
let t = ref 0. in
for k = 1 to m do
s := !s +. (terrain.((w+i-k)mod w) +.
            terrain.((w+i+k)mod w)) /.
             (float_of_int k);
t := !t +. 2. /. (float_of_int k);
done;
deriv.(i) <- c *. deriv.(i) -. a *. (terrain.(i) -. (!s)/.(!t));;

let normc (i) =
terrain.(i) <- terrain.(i) +. deriv.(i) *. b;;

let tick ( ) =
for i = 0 to vect_length terrain - 1 do
vert i;
diffc i;
done;
for i = 0 to vect_length terrain - 1 do
normc i;
done;;

while true do
tick ( );
if (wait_next_event [Poll; Key_pressed]).keypressed then
match read_key ( ) with
| ‘q‘ -> failwith "Fin"
| ‘ ‘ -> let (i,j) = mouse_pos ( ) in if i >= 0 && i < 640 then
               terrain.(i) <- float_of_int j;
| _ -> ( )
done;;
```

CHAPITRE 16

TP6 : diviser pour règner, programmation dynamique

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante. Les corrigés seront mis en ligne peu après. Tous ces documents se trouveront à l'adresse suivante :

`http ://www.nicollet.net/colles/`

16.1 Diviser pour régner

Un algorithme *diviser pour régner* se décompose en trois parties : division, traitement récursif, et fusion. Contrairement à un algorithme récursif usuel, on réduit ici le problème à non pas une, mais plusieurs instances de ce même problème à une taille plus petite. L'étape qui consiste à définir ces instances à partir du problème initial s'appelle *division* tandis que l'étape qui combine les solutions des instances en une solution du problème s'appelle *fusion*.

16.1.1 Le tri par fusion

Pour rendre plus clair le programme, il est souvent recommandé d'écrire des fonctions auxiliaires réalisant la division et la fusion, plutôt que d'en inclure le code directement dans le programme.

Question 1

Implémenter le tri par fusion, à savoir un algorithme *diviser pour régner* qui trie dans l'ordre croissant une liste de nombres en la divisant en deux listes de même taille.

16.1.2 Le tri rapide

Fonctionnement

Dans certains algorithmes, l'étape de division ou de fusion est triviale. Dans le cas du tri rapide, lorsqu'il est réalisé sur le tableau d'entrée, l'étape de fusion consiste simplement à ne rien faire. Tout le travail du tri rapide est fait par l'étape de division. Il existe de nombreuses manières de l'effectuer mais le principe est toujours le même : on choisit un pivot, puis on permute deux à

deux les éléments pour que les éléments plus petits que le pivot soient à sa gauche[1] et ceux qui sont plus grands soient à sa droite.

Question 2

Implémenter l'algorithme de division décrit ci-dessous et le tester.
L'utiliser pour écrire le tri rapide.

L'algorithme utilisé ici est le suivant : le premier élément de l'intervalle à diviser est le pivot. On place un indice i après le pivot, et j à la fin de l'intervalle. On choisit l'invariant de boucle suivant :

Tous les éléments à gauche de i sont plus inférieurs ou égaux au pivot.
Tous les éléments à droite de j sont strictement plus grands que le pivot.

On déplace i vers la droite et j vers la gauche autant que faire se peut en respectant l'invariant. Puis, on échange les éléments i et j du tableau, et on recommence. Dès que $i = j$, on introduit le pivot à la bonne position dans l'intervalle, et on renvoie sa position.

Performances

La complexité moyenne du tri rapide est de $O(n \log n)$, mais comment les cas pratiques se situent-ils par rapport à ce cas moyen et par rapport au cas le pire ? Est-il facile de s'éloigner du cas moyen pour tendre vers des complexités plus grandes ? La question 3 esquisse (littéralement) une réponse.

Question 3

Modifier les fonctions de l'exercice précédent pour obtenir une fonction `perf(t)` qui compte le nombre d'accès (en lecture ou en écriture) au tableau `t` lorsque celui-ci est trié par le tri rapide.

Ouvrir une fenêtre graphique 800x600 et dessiner pour chaque tableau `t` le point d'abscisse `vect_length t` et d'ordonnée `(perf t)/100`. Utiliser les tableaux suivants :
- En noir : 4000 tableaux de taille $1 \leq w \leq 800$, de contenu aléatoire.
- En rouge : 500 tableaux de taille $1 \leq w \leq 500$, contenant les entiers de 1 à w dans l'ordre croissant.

Les courbes tracées à l'exercice précédent représentent la complexité de l'algorithme de tri rapide. La courbe noire représente le cas moyen, de complexité $O(n \log n)$, alors que la courbe rouge représente le cas le pire, de complexité $O(n^2)$.

Ce tracé sert d'illustration à l'échelle réelle du pire cas de l'algorithme de tri rapide : les tableaux aléatoires sont groupés très près du cas moyen qui est très peu coûteux en temps. Au contraire, le pire cas se produit rarement et est beaucoup plus coûteux.

Toutes les approches qui baissent la courbe rouge (permuter aléatoirement les éléments avant de trier, choisir le pivot de manière plus sophistiquée etc) ont pour effet secondaire de faire monter la courbe noire[2].

[1] l'indice i est à gauche de l'indice j si $i < j$: on représente un tableau comme un enchaînement horizontal de cases numérotées de gauche à droite.

[2] De manière tout à fait anecdotique, ce phénomène est décrit par le théorème du *No Free Lunch* (pas de repas gratuit)

16.1.3 i-ème plus petit élément

Question 4

On se donne un tableau d'entiers, et on souhaite obtenir le i-ème élément dans l'ordre croissant. Ainsi, une méthode naïve est de trier le tableau, puis d'accéder à l'élément directement.

Utiliser la fonction de division du tri rapide pour trouver le i-ème élément en temps linéaire dans le cas moyen (on supposera, pour calculer la complexité, que dans le cas moyen le pivot n'est pas trop loin du milieu de l'intervalle).

En utilisant des moyens de partitionnement plus subtils, il est d'ailleurs possible d'obtenir une complexité linéaire dans tous les cas, plutôt que dans le cas moyen.

16.2 Programmation dynamique

On dispose d'un produit de matrices $A_1 \cdot A_2 \dots A_n$ que l'on souhaite calculer. Le problème est que les opérations de matrices sont coûteuses : c'est une opération qui est, en théorie, en temps $O(n^2)$, mais les algorithmes les plus efficaces s'approchent à peine de $O(n^{2.7})$. Pour simplifier, on considèrera que multiplier des matrices $a \times b$ et $b \times c$ prend un temps $O(a \cdot b \cdot c)$.

Le produit des matrices étant associatif, il existe plusieurs ordres dans lesquels faire ces produits. Par exemple, $(A \cdot B) \cdot C$ ou $A \cdot (B \cdot C)$. Au final, les différents ordres demandent des temps de calcul différents, et l'on souhaite trouver le temps de calcul minimal.

On veut écrire un programme qui détermine le temps de calcul minimal. On note $T(i,j)$ le temps minimal nécessaire pour calculer le produit $A_1 \cdot \ldots \cdot A_j$. Pour $i < k \leq j$:

$$\underbrace{A_i \cdot \ldots \cdot A_j}_{T(i,j)} = \underbrace{(A_i \cdot \ldots \cdot A_{k-1})}_{T(i,k-1)} \cdot \underbrace{(A_k \cdot \ldots \cdot A_j)}_{T(k,j)}$$

Il est donc possible, par un algorithme *diviser pour régner*, de déterminer pour chaque k le coût minimal pour effectuer les opérations $T(i,k-1)$ et $T(k,j)$, et en déduire le k pour lequel $T(i,j)$ se fait en un temps minimal.

Question 5

On représente les dimensions des N matrices par un tableau `a` de taille N+1, de façon à ce que les dimensions de la matrice A_i soient `a.(i)` et `a.(i+1)`.

Exprimer, en fonction de i, j, k, $T(i,k-1)$, $T(k,j)$ et `a`, le nombre minimum d'opérations nécessaires pour calculer le produit $A_i \dots A_j$ en coupant le produit en deux juste avant A_k.

Écrire une fonction récursive qui prend en argument `a`, i et j, et renvoie $T(i,j)$. Par convention, $T(i,i) = 0$.

En déduire une fonction `assoc a` qui calcule le temps minimal de calcul pour un produit de matrices. On pourra vérifier que :

```
assoc [| 30 ; 1 ; 40 ; 10 ; 25 |] = 1400
```

De par la nature même du problème, $T(i,j)$ sera utilisé pour calculer $T(k,j)$ pour $k < i$ et $T(i,k)$ pour $k > j$, ce qui représente au final une complexité déraisonnable d'environ $O(3^n)$, alors

qu'il n'y a que $O(n^2)$ valeurs à calculer. Il faut donc trouver un moyen de ne calculer chaque valeur qu'une seule fois, pour réduire la complexité.

Le principe de la programmation dynamique est stocker les résultats des calculs récursifs pour utilisation ultérieure. Pour cela, on utilise une structure de données capable d'associer à chaque sous-problème sa réponse : par exemple, si un sous-problème est décrit, comme dans le cas présent, par $0 \leq i \leq j < n$, on utilisera un tableau de tableaux ou une matrice[3].

On modifie ensuite l'algorithme récursif : celui-ci renvoie la valeur contenue dans la case correspondante de la structure de donnée (et, si elle est vide, la calcule avant de la renvoyer).

Lorsqu'on avait étudié les algorithmes récursifs, on avait vu que les ces algorithmes classaient les problèmes en deux catégories : les problèmes difficiles, qu'il faut découper en un ou plusieurs sous-problèmes résolus récursivement, et les problèmes triviaux dont la réponse est connue. En programmation dynamique, ces catégories ne sont pas fixes (elles sont dynamiques) : dès qu'un problème difficile est résolu, il devient un problème trivial, dont la réponse peut être trouvée dans la structure de données.

Question 6

Créer un tableau `t` de tableaux de dimensions $n \times n$, de façon à ce que la case `t.(j).(i)` serve à contenir la valeur de $T(i, j)$. Par convention, on notera par la valeur -1 un calcul qui n'a pas encore été effectué.

Dynamiser l'algorithme de la question précédente à l'aide de cette structure de données, et l'appeler `assoc2`.

Modifier `assoc` et `assoc2` pour qu'ils comptent le nombre d'appels récursifs non triviaux effectués (ceux qui font plus que de renvoyer une valeur). Vérifier que `assoc` effectue exactement $3^{(n-3)}$ appels, et `assoc2` $(n-2)(n-1)/2$ appels.

Il est parfois même possible de se passer entièrement d'appels récursifs. En effet, il existe des interdépendances entre les cases de la structure de données : certaines doivent être calculées avant d'autres. Il existe en particulier un ou plusieurs ordres dans lesquels il est possible de parcourir la structure de données et pour lesquels chaque case sera visitée après toutes les cases dont il faut disposer pour calculer sa valeur.

Dans le cas du produit de matrices, on visite les cases par taille croissante du produit de matrices associé, puisque chaque case n'est calculée qu'à partir de cases plus petites. Par exemple :

```
for taille = 0 to n-1 do
  for i = 0 to n - taille - 1 do
    calculer (i,i+taille)
  done
done
```

Il peut souvent être intéressant de se pencher sur cet ordre qui, par exemple dans le cas des problèmes décrits par un seul entier, est généralement très simple à identifier et correspond à une unique boucle.

[3] On utilise, dans le cas général, des tables de hachage, car elles sont capables d'utiliser des indices non entiers.

Question 7

Un joueur dans un jeu vidéo contrôle un vaisseau spatial. Celui-ci se déplace verticalement à vitesse constante, et vers la gauche ou vers la droite suivant les décisions du joueur. Sa vitesse horizontale est telle que, partant du point A, il ne peut atteindre le point B que si $A_y < B_y$ et $|A_x - B_x| < |A_y - B_y|$
Dans l'espace flottent des bonus que le joueur doit ramasser : chacun lui rapporte un point.

On donne un tableau `pos` de couples d'entiers, triés par deuxième coordonnée croissante, décrivant les positions des divers points bonus (sauf le premier, qui est la position initiale du joueur).

– Écrire une fonction récursive qui renvoie le score maximal que le joueur peut obtenir.

– Dynamiser la fonction précédente. Peut-on évaluer sa complexité sans trop de calculs ?

– Réécrire la fonction, mais cette fois de manière itérative en parcourant les sous-problèmes dans le bon ordre. Quelle est sa complexité ?

– Écrire une fonction qui affiche dans la fenêtre graphique les bonus et la trajectoire optimale du joueur.

16.3 Correction

Question 1 : On va coder la procédure en trois étapes : d'abord on sépare les listes, puis on les recolle, et enfin on utilise les deux fonctions ainsi définies pour obtenir un véritable tri fusion :

Exercice 16.3.1: *Tri fusion*

```
# let rec separe = function
   | []      -> ([],[])
   | a::([])   -> ([a],[])
   | t::(c::q) -> let (l1,l2) = separe q in
                  (t::l1,c::l2) ;;

# let rec recolle = function
   |(l1,[]) -> l1
   |([],l2) -> l2
   |(t::q,s::r) -> if t<s then t::(recolle (q,(s::r)))
                   else s::(recolle ((t::q),r));;

# let rec tri_fusion = function
   |[] -> []
   |a::[] -> [a]
   |l -> let (l1,l2)= separe l in
         recolle ((tri_fusion l1),(tri_fusion l2));;
```

Question 2 : Une nouvelle fois nous allons découper le programme en plusieurs sous programmes : un qui trie selon un «pivot» et l'autre qui se charge de réaliser la «tri rapide» que l'on veut effectuer.

Code CAML : *pivot*

```
# let rec pivot i j t =
     let a = t.(i) in
     let r = make_vect (j-i) a and s = make_vect (j-i) a in
     let u = ref 0 and v = ref 0 in
     for k = succ(i) to j do
        if t.(k) <= a then
           (r.(!u)<- t.(k);
            incr(u))
        else
           (s.(!v)<-t.(k);
            incr(v));
        done;
     for k = 0 to pred(!u) do
        t.(i+k)<-r.(k);
     done;
     t.(i+(!u))<-a;
     for k = 0 to pred(!v) do
        t.(i+succ(!u)+k)<- s.(k);
     done;
     !u;;
```

Exercice 16.3.2: *Tri rapide*

```
# let tri_rapide t =
   let rec trirapide tableau départ fin =
      if départ < fin then
         (let a = pivot départ fin tableau in
              trirapide tableau départ (départ + a-1);
              trirapide tableau (départ+succ(a)) fin)
      in
   trirapide t 0 (pred(vect_length t));
   t;;
```

Question 3 : On va rajouter en argument une référence qui sera le nombre de lecture ou modifications du tableau que l'on aura faites et cela nous permettra de tracer les courbes de complexité.

Code CAML :

```
# let rec pivotbis i j t compteur=
     let a = t.(i) in
     incr(compteur);
     let r = make_vect (j-i) a and s = make_vect (j-i) a in
     let u = ref 0 and v = ref 0 in
     for k = succ(i) to j do
        if t.(k) <= a then
           (r.(!u)<- t.(k);
            incr(u);
            incr(compteur))
        else
           (s.(!v)<-t.(k);
            incr(v);
            incr(compteur));
        done;
     for k = 0 to pred(!u) do
        t.(i+k)<-r.(k);
     done;
     t.(i+(!u))<-a;
     for k = 0 to pred(!v) do
        t.(i+succ(!u)+k)<- s.(k);
     done;
     !u;;

# let tri_rapide t =
   let u = ref 0 in
   let rec trirapide tableau départ fin decompte=
      if départ < fin then
         (let a = pivotbis départ fin tableau decompte in
              trirapide tableau départ (départ + a-1) decompte;
              trirapide tableau (départ+succ(a)) fin decompte)
      in
   trirapide t 0 (pred(vect_length t)) u;
   !u;;
```

Exercice 16.3.3:

```
#let tableau_alea_alea ()=
   let a = random__int(800)+1 in
   let t = make_vect a 0 in
   for i = 0 to pred(a) do
      t.(i)<- random__int(1000000);
   done;
   t;;

   let tableau_alea_rangé ()=
      let a = random__int(500)+1 in
      let t = make_vect a 0 in
      for i = 0 to pred(a) do
         t.(i)<- i;
      done;
      t;;

#let complexité() =
    set_color (black);
    for i = 1 to 4000 do
       let t = tableau_alea_alea() in
       let u = tri_rapide t in
       plot (vect_length t) (u/10);
    done;
    set_color (red);
    for i = 0 to 500 do
       let t = tableau_alea_rangé() in
       let u = tri_rapide t in
       plot (vect_length t) (u/10);
    done;;
```

Question 4 : On va faire un nouvel emploi de la fonction de tri par rapport à un pivot pour obtenir le i-ème élément d'une liste.

Exercice 16.3.4: *Tri en temps moyen linéaire*

```
#let élément_numéro k t =
   let n = vect_length t in
   if n< k then
       failwith("Mais tu es un boulet")
   else
      (let u = ref 0 in
       let v = ref (n-1) in
       let a = ref (pivot 0 (n-1) t) in
       while (!u+(!a)) <>(k-1) do
          if (!u+(!a))<(k-1) then u:=!u+(!a)+1
                             else v:=!u+(!a)-1;
          a:=(pivot (!u) (!v) t);
          done;
       t.(k-1));;
```

Question 5 : Tout d'abord la relation de récurrence que l'on cherche : en coupant juste avant A_k le temps de calcul nécessaire est de

$$T(i, k-1) + T(k, j) + \mathcal{O}(i \times k \times j)$$

Exercice 16.3.5: *Complexité*

```
#let assoc a =
    let n = vect_length a in
    let rec temps i j tableau =
        if i=j then 0
        else
           (let u = ref ((temps (i+1) j tableau)+
                    tableau.(i)*tableau.(i+1)*tableau.(j+1)) in
            for k = 1 to j-i-1 do
               let a = (temps (i) (i+k) tableau)+
                        tableau.(i)*tableau.(i+k+1)*tableau.(j+1)+
                        (temps (i+k+1) j tableau) in
               if a<=(!u) then u:=a;
            done;
            !u)
     in
     temps 0 (n-2) a;;
```

Question 6 : On va «dynamiser» cet algorithme afin d'obtenir un temps de calcul décent...

Exercice 16.3.6: *Version dynamique*

```
#
let assoc2 t =
   let n = vect_length t in
   let s = make_vect (n-1) (make_vect (n-1) (-1)) in
   for i = 0 to (n-2) do
       s.(i)<-(make_vect (n-1) (-1));
       done;
   let rec temps2 i j tableau dynamique =
       if i=j then (dynamique.(i).(i)<-0;
                    0)
       else if dynamique.(j).(i)=(-1) then
            (let u = ref ((temps2 (i+1) j tableau dynamique)+
                    tableau.(i)*tableau.(i+1)*tableau.(j+1)) in
            for k = 1 to j-i-1 do
               let a = (temps2 (i) (i+k) tableau dynamique)+
                        tableau.(i)*tableau.(i+k+1)*tableau.(j+1)+
                        (temps2 (i+k+1) j tableau dynamique) in
               if a<=(!u) then u:=a;
            done;
            dynamique.(j).(i)<-(!u);
            !u)
       else dynamique.(j).(i)
    in
   temps2 0 (n-2) t s;;
```

et on voudrait aussi compter le nombre d'appels effectués :

Code CAML : *Compte du nombre d'appels*

```
#let assoccompte a =
    let n = vect_length a in
    let compte = ref 0 in
    let rec temps i j tableau décompte=
        if i=j then 0
        else
           (let u = ref ((temps (i+1) j tableau décompte)+
                    tableau.(i)*tableau.(i+1)*tableau.(j+1)) in
            for k = 1 to j-i-1 do
               let a = (temps (i) (i+k) tableau décompte)+
                        tableau.(i)*tableau.(i+k+1)*tableau.(j+1)+
                        (temps (i+k+1) j tableau décompte) in
               if a<=(!u) then u:=a;
            done;
            incr(décompte);
            !u)
     in
     let a = temps 0 (n-2) a compte in
     !compte;;

#let assoc2 t =
   let n = vect_length t in
   let s = make_vect (n-1) (make_vect (n-1) (-1)) in
   for i = 0 to (n-2) do
       s.(i)<-(make_vect (n-1) (-1));
       done;
   let compte = ref 0 in
   let rec temps2 i j tableau dynamique décompte=
       if i=j then (dynamique.(i).(i)<-0;
                    0)
       else if dynamique.(j).(i)=(-1) then
            (let u = ref ((temps2 (i+1) j tableau dynamique décompte)+
                    tableau.(i)*tableau.(i+1)*tableau.(j+1)) in
            for k = 1 to j-i-1 do
               let a = (temps2 (i) (i+k) tableau dynamique décompte)+
                        tableau.(i)*tableau.(i+k+1)*tableau.(j+1)+
                        (temps2 (i+k+1) j tableau dynamique décompte) in
               if a<=(!u) then u:=a;
            done;
            dynamique.(j).(i)<-(!u);
            incr(décompte);
            !u)
       else dynamique.(j).(i)
    in
   let a = temps2 0 (n-2) t s compte in
   !compte;;
```

Question 7 : Et maintenant un petit jeu vidéo ! Voilà le résultat :

```
(* On résout le problème suivant: le vaisseau se trouvant sur le point i,
 * quel score maximal peut-il obtenir avec les points qu'il reste devant
 * lui?
 *
```

```
 * Cela peut s'évaluer facilement à partir de la résolution du problème
 * aux rangs [i+1 .. n] *)

(* Peut-on atteindre a à partir de b ? On utilise la division euclidienne
   pour représenter |ay - by| > |ax - bx| *)
let reach (ax,ay) (bx,by) =
  (ax - bx) / (ay - by) = 0;;



(* Résoudre le problème au rang i. b est le tableau des scores, rempli
 * à partir de l'indice i+1 *)
let solve_range (a) (b) (i) =
  for j = i + 1 to vect_length a - 1 do
    if reach a.(i) a.(j) then
      b.(i) <- max b.(i) (b.(j) + 1)
  done;;



(* Résoudre intégralement le problème. *)
let solve (a) =
  let b = make_vect (vect_length a) 0 in
  for i = vect_length a - 1 downto 0 do
    solve_range (a) (b) (i)
  done;
  b.(0);;



(* Dessiner une solution *)
let draw (a) =
  let b = make_vect (vect_length a) 0 in
  for i = vect_length a - 1 downto 0 do
    solve_range (a) (b) (i)
  done;

  (* Dessiner en se servant de b *)
  let s = ref 0 in

  for i = 1 to vect_length b - 1 do

    (* On peut continuer dans cette direction *)
    if reach a.(i) a.(!s) && b.(!s) - 1 = b.(i) then begin
      let (x,y) = a.(!s) in moveto x y;
      let (x,y) = a.(i) in lineto x y;
      s := i;
    end

    (* On ne peut pas continuer par ici *)
    else
```

```
        let (x,y) = a.(i) in plot x y
    done;;



set_color black;
let points = init_vect (600) (fun i -> (random__int 400,i)) in
clear_graph ( );
draw points;;
```

CHAPITRE 17

TP7 : Logique propositionnelle

Les énoncés des travaux pratiques seront disponibles sur internet quelques jours avant la séance correspondante. Les corrigés seront mis en ligne peu après. Tous ces documents se trouveront à l'adresse suivante :

`http ://www.nicollet.net/colles/`

17.1 Vérification

Un professeur d'informatique a l'intention de faire passer à ses élèves un examen sur machines. Il a pour cela écrit trois énoncés différents, ce qui lui assurait qu'aucun élève ne puisse copier sur ses voisins. Malheureusement, il apprend le matin de l'examen que les machines sont connectées en réseau et que les élèves peuvent communiquer entre eux même s'ils sont assis à des endroits opposés de la pièce.

Fin psychologue, le professeur sait que par *esprit concours*, tous ne sont pas prêts à s'aider les uns les autres, et il établit rapidement une fonction `collabore(i,j)` qui lui indique si les élèves `i` et `j` collaborent. Il lui reste deux heures avant l'examen pour écrire un programme qui, à partir de `collabore`, calcule une répartition des trois énoncés qui lui assure qu'il sera impossible de copier.

L'objectif de cette partie est de construire à partir de `collabore` une proposition logique qui est vraie si et seulement si la répartition des énoncés (caractérisée par des variables) ne permet pas aux élèves de copier.

Question 1

Notre ami professeur souhaite représenter des propositions logiques comme x_1, $x_1 \vee x_2$, $(x_1 \wedge \neg x_2) \vee x_2 \vee x_3$ et bien d'autres encore. Choisir un type parmi les suivants et l'utiliser pour représenter les trois exemples ci-dessus :

```
type log = Var of string
  | Not of log
  | And of log
  | Or of log ; ;

type log = Not of log
  | And of log * log
  | Or of log * log ; ;

type log = Var of int
  | Not of log
  | And of log list
  | Or of log list ; ;

type log = Var
  | Not
  | And
  | Or ; ;
```

Question 2

Les expressions manipulées ne sont pas très lisibles. Écrire une fonction `show` qui prend en argument une proposition, et qui l'affiche de manière plus compréhensible. Utiliser des parenthèses lorsque c'est utile, ainsi que les chaînes `" v "` (lettre v), `" ^ "` (circonflexe) et `string_of_char (char_of_int 172)` (pour le caractère ¬).

Aussi surprenant que cela puisse être, la fonction `collabore` correspond à une relation de divisibilité. Deux élèves peuvent copier l'un sur l'autre si et seulement si l'indice de l'un divise l'indice de l'autre, à l'exception de l'élève 1 qui n'aide personne :

```
let collabore i j =
  if i = 1 || j = 1 then false
  else if i < j then j mod i = 0
  else i mod j = 0 ; ;
```

Pour construire une proposition caractérisant une bonne répartition des énoncés, il faut traduire une répartition par des variables booléenes. Ainsi, pour représenter le fait que l'élève $1 \leq i \leq N$ a reçu l'énoncé $1 \leq j \leq E$, on utilise la variable $x_{k(i,j)}, 1 \leq k(i,j) \leq N \cdot E$ avec $k(i,j) = (i-1)E + j$. Ainsi, il faut traduire par la proposition logique les trois contraintes suivantes :

- Pour tout $1 \leq i \leq N$, $j \neq j'$: $x_{k(i,j)} = 0$ ou $x_{k(i,j')} = 0$ (un élève ne reçoit pas deux énoncés).
- Pour tout $1 \leq i \leq N$: il existe j tel que $x_{k(i,j)} = 1$ (chaque élève reçoit un énoncé).
- Pour tout $1 \leq j \leq E$, $i \neq i'$: si `collabore i i' = true`, alors $x_{k(i,j)} = 0$ ou $x_{k(i',j)} = 0$ (deux élèves qui peuvent collaborer ne doivent pas recevoir le même énoncé).

Question 3

Écrire la fonction `k(i,j)`, puis une fonction `contraintes n e` qui renvoie la proposition logique correspondant aux contraintes pour N élèves et E énoncés.

Le professeur s'attend à avoir 9 élèves à son examen. Il souhaite leur distribuer les énoncés A, B, C de la manière suivante :

1	2	3	4	5	6	7	8	9
A	B	C	A	B	C	B	C	A

Il souhaite utiliser la proposition logique `contraintes 9 3` pour déterminer si cette répartition permet ou non de tricher.

Question 4

On représente les valeurs des variables par un tableau de booléens. La valeur de x_i est alors donnée par la case `x.(i-1)`. Suivant cette convention, créer un tableau représentant la distribution envisagée par le professeur. Écrire également une fonction `inverse n e x` qui affiche un tableau comme des associations élève-énoncé.

Écrire une fonction récursive `eval x p` qui renvoie la valeur de la proposition `p` lorsque les valeurs des variables sont données par `x`.

Répondre à la question du professeur : cette distribution est-elle ou non correcte ?

17.2 Table de vérité

Le professeur souhaiterait utiliser le programme pour déterminer une répartition correcte. Il va donc tenter d'établir la table de vérité de la proposition `contraintes 9 3` et choisir une répartition pour laquelle elle est vraie. Il faut donc commencer par énumérer toutes les répartitions possibles, donc tous les ensembles de valeurs pouvant être pris par les variables. Autrement dit, il faut énumérer tous les tableaux `x` possibles.

On choisit pour cela l'algorithme suivant :

```
let next x =
  let i = ref 0 in
  while !i < vect_length x && x.(!i) do
    x.(!i) <- false;
    incr i
  done;
  if !i < vect_length x then (x.(!i) <- true; true) else false;;
```

On a donc choisi un ordre dans lequel énumérer les tableaux possibles (tous, et une seule fois chacun), et la fonction `next` transforme un tableau en le tableau suivant dans l'ordre choisi. Il faut commencer l'énumération par le tableau initialisé à `false`, et la fonction `next` renvoie `false` si et seulement si l'énumération est terminée (et revenue au point de départ).

Question 5

En utilisant la fonction `next`, écrire une fonction `table p n` qui affiche la table de vérité de la propriété `p` à `n` variables.

Commencer par écrire une fonction pour afficher un tableau de booléens suivi d'une valeur booléenne. Afficher les tables de vérité de $\neg$, $\wedge$ et $\vee$.

Une manière naïve de trouver une répartition correcte est donc de commencer à construire une table de vérité valeur par valeur jusqu'à rencontrer une répartition pour laquelle la proposition est vraie.

Question 6

Modifier la fonction de la question précédente pour obtenir une fonction `satisfaire p n` qui renvoie un arrangement de `n` variables satisfaisant la propriété `p` (ou `failwith "Pas de solution !"`), et qui affiche le nombre de combinaisons essayées.

Satisfaire `contraintes 4 2`, et essayer de satisfaire `contraintes 8 3`. A-t-on une chance de satisfaire `contraintes 15 4` avec cette approche ?

On constate le problème : les propositions étudiées ont beaucoup de variables (respectivement 8, 24 et 60), et il y a donc $2^8 = 256$, $2^{24} \sim 17 \cdot 10^6$ et $2^{60} \sim 10^{18}$ possibilités. Ces nombres sont beaucoup trop grands : il faut donc trouver un moyen plus intelligent de déterminer une solution.

17.3 Une approche plus efficace

Il existe une deuxième façon de voir la recherche par table de vérité. On peut considérer que l'algorithme de recherche est en réalité récursif, et résout le problème suivant : étant données les

valeurs des variables x_1 à x_k, existe-t-il des valeurs de $x_{k+1} \dots x_n$ pour lesquelles la proposition est vraie ?

La résolution consiste, si $k = n$, à regarder si la proposition est vraie ou, dans le cas contraire, à résoudre le problème au rang $k + 1$ pour les deux valeurs possibles de x_{k+1}.

Question 7

Réécrire l'algorithme de recherche de solution de manière récursive. Indication : cela ne doit pas prendre plus de huit lignes.

Vu sous cet angle, on constate tout de suite une inefficience dans l'algorithme : en effet, il commence par donner une valeur à toutes les variables avant de regarder la valeur de la proposition. Or, il se pourrait que les deux premières hypothèses soient contradictoires, et donc que tout le travail fait par la suite soit inutile !

Nous allons donc modifier l'algorithme en ajoutant une ligne supplémentaire au début de chaque appel, qui regarde si les hypothèses faites jusque-là sont ou non contradictoires (et abandonne immédiatement la piste en cours si c'est le cas). Pour cela, il faut d'abord déterminer si des hypothèses sont contradictoires.

Les valeurs des variables sont données par un tableau `x` contenant :

```
type tern = True | False | Maybe
```

Autrement dit, il s'agit de logique ternaire où une troisième valeur, `Maybe` (ou ?), indique qu'aucune hypothèse n'a été faite sur une variable. Les opérations booléennes se comportent de la manière suivante : si Maybe apparaît dans les opérandes, alors le résultat est Maybe, sauf pour $0 \wedge ? = ? \wedge 0 = 0$ et $1 \vee ? = ? \vee 1 = 1$.

Question 8

Écrire une fonction `evalt x p` qui évalue en logique ternaire une proposition `p` (facultatif : utiliser de l'évaluation paresseuse, comme celle des operateurs binaires `&&` et `||`).

L'interprétation est la suivante : une proposition en logique ternaire a pour valeur `True` (respectivement `False`) si et seulement si elle aurait la même valeur en logique binaire en remplaçant chaque variable égale à `Maybe` par `True` ou `False`. Par conséquent, `evalt x p = False` permet de déterminer si l'ensemble d'hypothèses `x` est contradictoire.

Question 9

Modifier l'algorithme de recherche récursif pour interrompre la recherche dès qu'une contradiction est détectée.

Constater les performances très supérieures sur `contraintes 8 3`, puis résoudre `contraintes 15 4` et `contraintes 40 5`.

17.4 Conclusion

Ce système de résolution est *a priori* peu efficace et assez difficile à comprendre. D'ailleurs, à titre de **Question 10**, écrivez une fonction `Caml Light` qui résout le problème étudié (cela prend huit lignes et fonctionne en temps quadratique).

Cependant, des modes de résolution plus ou moins généralisés existent, et leur puissance dépasse celle de la seule logique propositionnelle. On obtient alors des langages de programmation comme

Prolog, dans lequels on ne décrit pas l'algorithme permettant de trouver la solution (ce qui est parfois très difficile), mais on se contente de décrire les contraintes que l'on impose à cette solution. Le langage produit alors une solution vérifiant les contraintes, ou bien l'ensemble de toutes les solutions, ou encore la solution maximisant une certaine fonction, et ainsi de suite.

Pour la culture générale, comment peut-on placer N reines sur un échiquier $N \times N$ sans qu'aucune ne puisse être prise par une autre en un mouvement ? Question subsidiaire : serait-il plus rapide d'écrire un programme `Caml Light` répondant à cette question pour tout N, ou de traduire ce problème en une proposition logique et d'utiliser le système de résolution ci-dessus pour trouver une solution ?

17.5 Correction

Troisième partie

Contrôles

CHAPITRE 18

Contrôle n° 1

18.1 correction

18.1.1 Tableaux doubles

On pourrait imaginer utiliser des tableaux dans des tableaux mais quand on donne une valeur à un case d'un tableau, mais tous les tableaux de toutes les cases sont liés, donc la valeur passe dans toutes les cases pour compenser on doit faire une boucle *for* qui crée un tableau par case. On va plutôt utiliser des fonctions de deux variables.

Code CAML : *Question 1*

```
>>let rec trouve e f a b =
   if b<a then
      (false,0)
   else let m=(a+b)/2 in
      if e = f(m) then (true,m)
      else if e>f(m) then trouve e f m+1 b
      else trouve e f a m-1;;
```

La complexité est en $\mathcal{O}(\log(b-a))$.

Code CAML : *Question 2*

```
>>let trouve2c =
   let déjàtrouvé = ref false and où = ref (0,0) and k = ref 1 in
   while (!k<=Nx) && (not !déjàtrouvé) do
      let g(x) = f(!k,x) in
      let (r,m) = trouve e g 1 Ny in
      if r then (déjàtrouvé:=true ; où := (!k,m));
      incr k;
   done;
   (!déjàtrouvé,fst(!où),snd(!où));;
```

la complexité est en $\mathcal{O}(N_x(\log(N_y)+1))$

Code CAML : *Sur les lignes*
```
>>let trouve2l =
   let déjàtrouvé = ref false and où = ref (0,0) and k = ref 1 in
   while (!k<=Ny) && (not !déjàtrouvé) do
      let g(x) = f(x,!k) in
      let (r,m) = trouve e g 1 Nx in
      if r then (déjàtrouvé:=true ; où := (m,!k));
      incr k;
   done;
   (!déjàtrouvé,fst(!où),snd(où));;
```

la complexité est en $\mathcal{O}(N_y(\log(N_x)+1))$. il vaut mieux que le plus grand des deux N soit sous le log : cela se prouve en étudiant les variations de la fonction $\frac{\log(x)}{x}$ qui est décroissante à partir de 2.

Code CAML : *Question 3*
```
>>let trouve e f =
   let rec util f a b =      cherche parmi les f(x,y) sachant que x>=a et y<=b
     if a>Nx then (false,0,0)
     else if b<1 then (false,0,0)
     else if e = f(a,b) then (true,a,b)
     else if e>f(a,b) then util e f (a+1) b
     else util e f a (b-1)
     in
   util e f 1 Ny;;
```

18.1.2 Les records

Code CAML : *Les records*
```
>>let record l =
   let rec util l a =      a := meilleure performance
     match l with
      []->[]
     |t::q -> if t<=a then util q a
              else t::(util q t) in
     match l with
      []->[]
     |a::q-> a::(util q a);;
```

18.1.3 Maximisons au maximum

Code CAML : *Décomposition en base 2*

```
>>let décomp k n =
   let res = make_vect n 0 in
   let a = ref k in
   for i = 0 to n-1 do
      res.(i)<-(!a mod 2)
      a:=!a/2;
   done;
   res;;
```

Code CAML :

```
>>let f a I =
   let n = ref(0.) and p= ref(1.) in
   for i = 0 to n-1 do
      if I.(i)=1 then
         (s:=!s+.a.(i);p:=!p*.a.(i));
      done;
   !s/.!p;;
```

Code CAML :

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

>>let pbénoncé a =
   let n = vect_length a in
   let m = ref 0 and s = ref 0. in
   for k = 1 to (puissance n) do
      let u = f(décomp k n) in
      if us then (m:=k;s:=n);
   done;
   décomp(!m);;
```

18.1.4 Jouons à la marchande

1. Code CAML : *Programmons la marchande*

```
>>let rend_monnaie k p =
   let n = vect_length p in
   let res = make_vect n 0 in
   let k' = ref k in
   fot i = 0 to n-1 do
      res.(i)<-!(k')/p.(i);
      k':=!k'-res.(i)*p.(i);
   done;
   res;;
```

2. rend_monnaie 16 va renvoyer une pièce de 12 et 4 pièces de 1, soit 5 pièces. Mais ce n'est pas optimal car on pourrait faire la même chose avec deux pièces.

3. p est optimal si pour tout n le rendu de pièces optimal est celui qui est effectué par rend_monnaie. Si $n = \sum d.(i)p.(i)$ par ce programme vérifie que $\sum_{i>k} d.(i)p.(i) < p.(k) \qquad (\star)$.
   Rend_monnaie fournit l'unique rendu de pièces vérifiant $(\star)$.
   Montrons que $(a^k)_{k\in[\![0,n]\!]}$ est optimal, alors une somme de termes avec $k > 2$ est inférieure à a sinon on peut grouper a pièces de valeur a^k en une seule de valeur a^{k+1}.
   On a alors
   $$\forall k, \sum_{i>k} d.(i).a^i \leq \sum_{i>k} (a-1)a^i = a^k - 1 < p.(k)$$

4. soit p le tableau des pièces.
   Si $n = \sum_{i=0}^{5} d.(i)p.(i)$ est un rendu optimal alors $d.(1) \leq 2$ car $d.(1) \geq 3 \Rightarrow 20 + 20 + 20 = 50 + 10$. De plus $d.(2) \leq 1, d.(3) \leq 1, d.(4) \leq 2, d.(5) \leq 1$.
   - $d.(1) = 2 \wedge d.(2) = 1$ impossible car $20 + 20 + 10 = 50$
   - $d.(4) = 2 \wedge d.(5) = 1$ impossible car $2 + 2 + 1 = 5$
   - $d.(1)p(1) + d.(2)p.(2) \leq 40$ et $d.(4)p.(4) + d.(5)p.(5) \leq 4$
   - Il suffit de comparer toutes les sommes de i à 5...

On peut noter quelques conseils : ne pas faire agir des sous procédures sur des objets extérieurs à celle-ci : on devrait pouvoir la réutiliser en faisant un simple copier-coller dans un autre programme. Ne jamais renvoyer, dans un «if» quelque chose qui n'est ps du même type pour chaque cas : si on renvoie un float dans le cas où le test est positif, ne pas renvoyer un integer dans le cas où le test est négatif...