Lycée Faidherbe, 2020-2021

# D.S. d'informatique

**MPSI option informatique**

Version du 23 juin 2021

# Table des matières

## Devoir A

# Ensembles

Dans ce sujet nous allons donner les fonctions qui permettent de travailler avec des ensembles. Dans les exemples nous considérerons des ensembles de mots (des chaînes de caractères) mais tout type de donnée est possible, à condition qu'il soit le même pour tous les éléments d'un ensemble.

- Les ensembles seront implémenté sous formes de listes de chaînes de caractères. un ensemble $E$ sera représenté par une liste notée `e`.
- Nous utiliserons, dans les exemples, les ensembles $e_1$ et $e_2$ représentés respectivement par `e1 = ["a"; "e"; "i"; "o"; "u"]` et `e2 = ["i"; "n"; "f"; "o"]`
- Les listes qui représentent les ensembles devront être sans doublon : avant d'ajouter un élément à une liste il faudra toujours s'assurer qu'il n'en fait pas déjà partie.
- L'ordre des éléments n'a pas d'importance, en tant qu'ensemble $e_1$ peut être représenté aussi par `["a"; "i"; "u"; "o"; "e"]` ou `["u"; "o"; "i"; "a"; "e"]`.
- L'ensemble vide est représenté par `[]`.

## 1 Opérations ensemblistes

### 1.1 Premières fonctions

**Question 1**
*Écrire une fonction* `estVide e` *qui renvoie* `true` *ou* `false` *selon que l'ensemble E est vide ou non.*

```
estVide : 'a list -> bool
```

**Question 2**
*Écrire une fonction* `appartient x e` *qui renvoie* `true` *ou* `false` *selon que l'élément x appartient ou non à l'ensemble E.*

```
appartient : 'a -> 'a list -> bool
```

`appartient "n" e1` renverra `false` et `appartient "u" e1` renverra `true`.

**Question 3**
*Écrire une fonction* `ajoute x e` *qui renvoie une représentation de l'ensemble E augmenté de x. On rappelle que si x appartient à E alors la fonction doit renvoyer un ensemble avec les mêmes éléments que* `e`.

```
ajoute : 'a -> 'a list -> 'a list
```

`ajoute "ou" e1` renverra, par exemple, `["ou"; "a"; "e"; "i"; "o"; "u"]`
et `ajoute "i" e1` pourra renvoyer `["a"; "e"; "i"; "o"; "u"]`.

## 1.2 Opérations

Voici une fonction qui prend deux ensembles en argument :

```
let rec fonction e f =
  match e with
  |[] -> []
  |t::q -> if appartient t f
           then t::(fonction q f)
           else fonction q f;;

fonction : 'a list -> 'a list -> 'a list
```

**Question 4**
*Que renvoie* `fonction e1 e2` *?*
*Plus généralement que renvoie* `fonction e f` *en fonction des ensembles $E$ et $F$ ?*
*Par quel moyen peut-on démontrer le résultat ci-dessus ?*

**Question 5**
*La fonction effectue des comparaisons ; déterminer combien de comparaisons sont faites au maximum lors de l'appel de* `fonction e f` *en fonction du nombre d'éléments de $E$ et $F$.*
*Dans quel cas ce maximum est-il atteint ?*

**Question 6**
*Écrire une fonction* `union e f` *qui renvoie une représentation de l'union des ensembles représentés par* `e` *et* `f`*.*

```
union : 'a list -> 'a list -> 'a list
```

`union e1 e2` pourra renvoyer `["a"; "e"; "u"; "i"; "n"; "f"; "o"]`.

**Question 7**
*Écrire une fonction* `difference e f` *qui renvoie une représentation de l'ensemble des éléments appartenant à $E$ mais pas à $F$, $E \setminus F$.*

```
difference : 'a list -> 'a list -> 'a list
```

`difference e1 e2` pourra renvoyer `["a"; "e"; "u"]`.

**Question 8**
*En utilisant les questions précédentes écrire des fonctions courtes* `inclus e f` *et* `egaux e f` *testant si les ensembles $E$ et $F$ vérifient respectivement $E \subset F$ et $E = F$.*

```
inclus : 'a list -> 'a list -> bool
egaux : 'a list -> 'a list -> bool
```

# 2 Parties d'un ensemble

Dans cette partie nous allons construire des ensembles de sous-ensembles de $E$.
On aura donc à utiliser des listes de listes : `'a list list`.

## 2.1 Fonctions utiles

Dans cette partie on ne fera que des unions d'ensembles dont sait à l'avance qu'ils seront disjoints. Or la fonction `union` exécute beaucoup d'instructions de comparaisons pour les doublons.

**Question 9**
*Combien de comparaisons sont exécutées lors de* `union e f` *lorsque* $e$ *et* $f$ *sont deux listes de même taille* $n$ *sans point commun ?*
*On commencera par compter le nombre de comparaisons effectuées lors de l'appel de* `appartient x f` *où* `f` *est une liste de taille* $n$ *ne contenant pas* $x$*.*

On préfère donc écrire l'union à l'aide de la concaténation.

**Question 10**
*Écrire une fonction* `concat liste1 liste2` *qui produit la liste comportant tous les éléments de* `liste1` *suivis de tous les éléments de* `liste2`*.*
*On écrira sa propre fonction sans utiliser l'opérateur* `@`*.*
*Donner, en fonction du nombre d'éléments de* `liste1`*, le nombre d'assemblages, c'est-à-dire le nombre de constructions* `t::q`*, effectuées (sans démonstration).*

```
concat : 'a list -> 'a list -> 'a list
```

`concat e1 e2` devra renvoyer `["a"; "e"; "i"; "o"; "u"; "i"; "n"; "f"; "o"]`.

**Question 11**
*Écrire une fonction* `adjonction x liste` *qui reçoit une variable* `x` *et une liste de liste et qui renvoie la liste des listes qui sont les éléments de* `liste` *auxquels on a ajouté* `x`*.*

```
adjonction : 'a -> 'a list list -> 'a list list
```

`adjonction "c" [["a"; "f"]; ["e"]; []; ["g"; "u"]]` devra renvoyer
`[["c"; "a"; "f"]; ["c"; "e"]; ["c"]; ["c"; "g"; "u"]]`.

**Question 12**
*Montrer que si* `liste` *comporte* $n$ *listes alors* `adjonction x liste` *effectue* $2n$ *assemblages.*

## 2.2 Constructions

### Ensembles des parties

Pour construire la liste des parties d'un ensemble, on peut remarquer que si l'ensemble $E$, non vide, est représenté par `t::q` et si on note $E'$ le sous-ensemble représenté par `q` alors les parties de $E$ sont

- soit des parties de $E'$
- soit de la forme $\{t\} \cup F$ où $F$ est une partie de $E'$.

**Question 13**
*Écrire une fonction* `parties e` *qui prend comme argument une liste représentant un ensemble* $E$ *et renvoie une liste d'ensembles représentant l'ensemble de toutes les parties de* $E$*.*
*On notera que* $\mathcal{P}(\emptyset) = \{\emptyset\}$ *;* $\mathcal{P}(\emptyset)$ *n'est pas vide.*

```
parties : 'a list -> 'a list list
```

**Question 14**
*Montrer que si $C(n)$ est le nombre d'assemblages effectuées par* `parties e` *où* `e` *est une liste de taille $n$, alors $C(n) = 3.2^{n-1} + C(n-1)$.*
*En déduire la valeur de $C(n)$.*

Si vous trouvez $C(n) = 2.2^{n-1} + 2.C(n-1)$ c'est que vous n'avez pas respecté un principe fondamental : **ne jamais répéter les mêmes calculs**.

La construction d'une liste de taille $p$ à partir de $p$ éléments demande $p$ assemblages.

**Question 15**
*Combien d'assemblages seraient nécessaires pour construire séparément les $2^n$ listes qui représentent les sous-ensembles d'un ensemble de $n$ éléments ? On admettra que la construction d'une liste de taille $p$ demande $p$ assemblages.*
*Donner succinctement une explication à la différence entre les deux derniers résultats.*

### Parties à $p$ éléments

On note $\mathcal{P}_p(E)$ l'ensemble des parties de $E$ à $p$ éléments. $\mathcal{P}_p(E)$ est vide si on a $p > |E|$.
Pour construire $\mathcal{P}_p(E)$, on peut remarquer que si l'ensemble $E$, non vide, est représenté par `t::q` et si $E'$ est représenté par `q` alors les parties de $E$ à $p$ éléments sont

- soit des parties de $E'$ à $p$ éléments
- soit de la forme $\{t\} \cup F$ où $F$ est une partie de $E'$ à $p-1$ éléments.

**Question 16**
*Écrire une fonction* `partiesTaille e p` *qui prend comme arguments une liste représentant un ensemble $E$ et un entier et qui renvoie une liste d'ensembles représentant l'ensemble de toutes les parties de $E$ à $p$ éléments.*

```
partiesTaille : 'a list -> int -> 'a list list
```

**Question 17**
*Montrer que si $C(n,p)$ est le nombre d'assemblages effectuées par* `partiesTaille e p`
*où* `e` *est une liste de taille $n$, alors $C(n,0) = 0$, $C(n,p) = 0$ pour $0 \leqslant n < p$ et*
*$C(n,p) = 3.\binom{n-1}{p-1} + C(n-1,p-1) + C(n-1,p)$ pour $n \geqslant p > 0$.*

**Question 18**
*Calculer $C(n,1)$ pour tout $n$.*
*Puis calculer $C(n,2)$ pour tout $n$.*

**Question 19**
*Calculer $C(n,p)$ pour tout $n \geqslant p$.*

# 3 Solutions

**Question 1 (Solution)**

```
let estVide e =
  e = [];;
```

**Question 2 (Solution)**

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

**Question 3 (Solution)**

```
let ajoute x e =
  if appartient x e
  then e
  else x::e;;
```

**Question 4 (Solution)**
`fonction e1 e2` renvoie `["i"; "o"]`, c'est une représentation de l'intersection des ensembles donnés en paramètre. On le démontre par récurrence sur le nombre d'éléments de $E$.

**Question 5 (Solution)**
Chaque élément de $E$ va être comparé à tous les éléments de $F$ s'il n'appartient pas à $F$ et c'est le pire des cas. Au total on fait au plus $|E|.|F|$ comparaisons où $|E|$ est le cardinal de $E$. Ce maximum est atteint si l'intersection de $E$ et de $F$ est vide.

**Question 6 (Solution)**

```
let rec union e f =
  match e with
  |[] -> f
  |t::q -> if appartient t f
           then union q f
           else t::(union q f);;
```

**Question 7 (Solution)**

```
let rec difference e f =
  match e with
  |[] -> []
  |t::q -> if appartient t f
           then difference q f
           else t::(difference q f);;
```

**Question 8 (Solution)**

```
let inclus e f = estVide (difference e f);;

let egaux e f = (inclus e f) && (inclus f e);;
```

**Question 9 (Solution)** Comme $x$ n'est jamais trouvé, `appartient x f` compare $x$ à chaque élément de $F$, il y a donc $n$ comparaisons. `union e f` teste l'appartenance de chaque élément de $E$ dans $F$ avec un résultat `false` à chaque fois donc il y a $n^2$ comparaisons.

**Question 10 (Solution)**

```
let rec concat liste1 liste2 =
  match liste1 with
  |[] -> liste2
  |t::q -> t::(concat q liste2);;
```

Il y a un assemblage par élément de `liste1`.

**Question 11 (Solution)**

```
let rec adjonction x liste =
  match liste with
  |[] -> []
  |t::q -> (x::t)::(adjonction x q);;
```

**Question 12 (Solution)** Pour chaque liste `l` appartenant à `liste` on doit ajouter `x` puis on doit ajouter la liste `x::l` considérée comme un élément à la liste de listes : on fait donc 2 ajout par élément.

**Question 13 (Solution)**

```
let rec parties e =
  match e with
  |[] -> [[]]
  |t::q ->  let part = parties q in
            part @ (adjonction t part);;
```

**Question 14 (Solution)**
Lors de l'appel récursif la création de `part` demande $C(n-1)$ assemblages.
L'appel de `ajout x part` effectue $2.2^{n-1}$ assemblages car `part` contient $2^{n-1}$ éléments.
La concaténation `part @ (ajout x part)` effectue $2^{n-1}$ assemblages d'où le résultat.
Comme $C(0) = 0$ on obtient, par récurrence $C(n) = 3(2^n - 1)$.

**Question 15 (Solution)** Il y a $\binom{n}{p}$ listes de taille $p$ donc la construction des listes demande $\sum_{p=0}^{n} p\binom{n}{p} = \sum_{p=1}^{n} n\binom{n-1}{p-1} = n.2^{n-1}$ assemblages auxquels il faut ajouter $2^n$ assemblages pour construire la liste des sous-ensembles. On aboutit à $(n+2).2^{n-1}$ assemblages.
On a $(n+2).2^{n-1} - 3(2^n - 1) = (n-4)2^{n-1} + 3 > 0$ pour $n \geqslant 4$.
Dans la listes des sous-ensembles calculée par la fonction, beaucoup de listes ont des parties partagées, cela diminue le nombre de liens nécessaires.

**Question 16 (Solution)**

```
let rec partiesTaille e p=
  match e, p with
  |[],p -> []
  |e, 0 -> [[]]
  |t::q, p->  let part1 = partiesTaille q (p-1) in
              let part2 = partiesTaille q p in
              (adjonction t part1) @ part1;;
```

**Question 17 (Solution)**
Pour $p = 0$ on a directement le résultat, sans assemblage, donc $C(n, 0) = 0$.

On prouve par récurrence sur $n$ que `partiesTaille e p` renvoie la liste vide sans calculer d'assemblage si on a $p > n$ où $n$ est la taille de `e` : on note $P(n)$ cette propriété.

1. $P(0)$ est vraie car une liste de taille 0 est vide et `partiesTaille [] p` renvoie directement une liste vide.
2. Si $P(n-1)$ est vraie alors l'appel de `partiesTaille liste p` avec `e = t::q` de taille $n$ calcule `part1 = partiesTaille q (p-1)` et `part2 = partiesTaille q p` avec `q` de taille $n-1 < p$ et $n-1 < p-1$ donc on peut appliquer l'hypothèse de récurrence : on n'a pas effectué d'assemblage et `part1` et `part2` sont vides.
   Dans le cas de listes vides ni `adjonction` ni `@` ne font d'assemblage d'où $P(n)$ est vraie.

L'autre propriété découle directement de la formule récursive.

**Question 18 (Solution)** On a $C(0, 1) = 0$ et, pour $n \geqslant 1$,
$C(n, 1) = 3.\binom{n-1}{0} + C(n-1, 0) + C(n-1, 1) = 3 + C(n-1, 1)$ donc $C(n, 1) = 3n$.

On a alors $C(1, 2) = 0$ et, pour $n \geqslant 2$
$C(n, 2) = 3.\binom{n-1}{1} + C(n-1, 1) + C(n-1, 2) = 6(n-1) + C(n-1, 2)$
donc $C(n, 2) = 6\sum_{k=1}^{n-1} k = 3n(n-1)$ pour $n \geqslant 2$.

**Question 19 (Solution)** On a en fait $C(n, 1) = 3\binom{n}{1}$ et $C(n, 2) = 3.2\binom{n}{2}$ pour $n \geqslant 2$.
On peut imaginer et prouver par récurrence sur $n$ qu'on a $C(n, p) = 3p\binom{n}{p}$ pour $n \geqslant p$.

On doit construire $\binom{n}{p}$ liste de taille $p$ puis les assembler en liste ce qui demande
$p\binom{n}{p} + \binom{n}{p} = (p+1)\binom{n}{p}$ assemblages au moins.
On est juste 3 fois plus lent, ce qui est acceptable.

## Devoir B

# TRANCHES

Dans ce sujet nous allons étudier le problème de la somme maximale d'une tranche en proposant plusieurs algorithmes. Ce problème consiste à se donner une suite finie de valeurs dont l'ordre est fixé et de déterminer une tranche dont la somme des termes est maximale. Un tranche est une portion de la liste formée de termes adjacents. On donnera plusieurs moyens d'arriver à une complexité linéaire.

Si on note $a_0$, $a_1$, …, $a_{n-1}$ les termes successifs de la suite, on pose $S_{i,j} = \sum_{k=i}^{j-1} a_k$ pour $0 \leqslant i \leqslant j \leqslant n$, c'est la somme de la tranche de $i$ à $j-1$ et on cherche $T = \max\{S_{i,j} \ ; \ 0 \leqslant i \leqslant j \leqslant n\}$

Par exemple, pour la suite $4, -5, 2, -1, 3, 2, -7, 3$, la tranche maximale est $S_{2,6} = a_2 + a_3 + a_4 + a_5$ : $T = 2 - 1 + 3 + 2 = 6$.

- On ne considérera que des suites d'entiers pour simplifier l'écriture des calculs mais les algorithmes fonctionnent aussi avec des flottants.
- On pose, par convention, $S_{i,i} = \sum_{k=i}^{i-1} a_k = 0$, la somme des termes d'une tranche vide est nulle. Ainsi la somme maximale d'une tranche sera toujours positive ou nulle.
- Selon les parties, les suites seront modélisées par un tableau ou par une liste. On respectera la structure de données utilisée par l'énoncé et on s'abstiendra de convertir les listes en tableaux ou réciproquement.
- Pour les calculs de complexité, on comptera le nombre d'additions et de comparaisons effectuées. Seules les comparaisons entre éléments de la suite seront comptabilisées, en particulier les instructions de gestion des indices d'une boucle `for` ne sont pas comptabilisées.
- Les parties sont de difficulté globalement croissante.
- On rappelle qu'il est toujours possible d'utiliser les fonctions d'une question, même si on n'y a pas répondu.
- `OCaml` contient une fonction `max : 'a -> 'a -> 'a` qui renvoie le maximum de deux éléments. Elle compte pour une comparaison dans les calculs de complexité.
- On peut effectuer une boucle avec des indices décroissants :

```
for i = n downto 1 do
```

- $\lfloor x \rfloor$ désigne la partie entière de $x$.
  $\lceil x \rceil$ désigne la partie entière supérieure de $x$ : c'est le plus petit entier supérieur ou égal à $x$.

# 1 Tableaux

Dans cette partie, et la suivante, nous allons modéliser la suite finie par un tableau.
L'exemple ci-dessus sera donc enregistré sous la forme

```
let t_ex = [|4; -5; 2; -1; 3; 2; -7; 3|];;
```

Ainsi $a_k$ sera la valeur `t.(k)`.

## 1.1 Algorithme naïf

On commence par une écriture directe de l'énoncé.

**Question 1**
*Écrire une fonction* `somme_tranche : int array -> int -> int -> int` *telle que* `somme_tranche t i j` *renvoie la somme* $S_{i,j} = \sum_{k=i}^{j-1} \texttt{t.(k)}$.

**Question 2**
*Écrire une fonction* `tranche_max1 : int array -> int` *telle que* `tranche_max1 t` *renvoie la somme maximale d'une tranche en calculant les sommes de toutes les tranches avec la fonction* `somme_tranche`.

**Question 3**
*Déterminer, en fonction de $i$ et $j$, le nombre d'additions effectuées lors de l'appel de* `somme_tranche1 t i j`.
*En déduire la complexité de* `tranche_max1` *en fonction de $n$, la taille de* `t`.
*On donnera le résultat sous la forme $\mathcal{O}(n^\alpha)$.*

## 1.2 Pré-traitement

Pour améliorer la complexité, il existe plusieurs pistes.
Nous allons commencer par l'utilisation d'un pré-traitement qui consiste à créer une nouvelle donnée à partir de laquelle les calculs sont plus simples.

On remarque la somme d'une tranche peut s'écrire $S_{i,j} = \sum_{k=0}^{j-1} \texttt{t.(k)} - \sum_{k=0}^{i-1} \texttt{t.(k)} = S_{0,j} - S_{0,i}$.

On rappelle que $S_{0,0} = 0$. On a donc $n+1$ sommes qui interviennent :
$S_{0,k}$ pour $0 \leqslant k \leqslant n$ ; $n$ étant la longueur du tableau.

**Question 4**
*Écrire une fonction* `cumul : int array -> int array` *telle que* `cumul t` *renvoie un tableau* `s`, *de taille $n+1$, si* `t` *est de longueur $n$, et tel que* `s.(p)` *a pour valeur $S_{0,p}$ pour $0 \leqslant p \leqslant n$ avec la convention ci-dessus pour $p = 0$.*
*Une solution de complexité linéaire (c'est-à-dire en $\mathcal{O}(n)$) sera valorisée.*

`cumul t_ex` doit renvoyer `[|0; 4; -1; 1; 0; 3; 5; -2; 1|]`.

> Le calcul de la somme d'une tranche est simplifié : c'est une différence de 2 termes plutôt qu'une somme de $j - i$ termes.

**Question 5**
*Écrire une fonction* `tranche_max2 : int array -> int` *telle que* `tranche_max2 t` *renvoie la somme maximale d'une tranche en utilisant ce calcul simplifié. La fonction devra calculer le tableau des sommes partielles en utilisant la fonction* `cumul`.

**Question 6**
*Déterminer la complexité de* `tranche_max2` *en fonction de $n$, la taille de* `t`.
*On donnera le résultat sous la forme $\mathcal{O}(n^\alpha)$.*

### 1.3 Complexité linéaire

On va maintenant améliorer encore l'algorithme ci-dessus.
En effet, pour $j$ fixé, le maximum de $S_{i,j} = S_{0,j} - S_{0,i}$ est égal à $S_{0,j} - m_j$ où $m_j$ est la valeur minimale des $S_{0,i}$ pour $i$ variant entre 0 et $j$.

**Question 7**
*Écrire une fonction* `tranche_max3 : int array -> int` *telle que* `tranche_max3 t` *renvoie la somme maximale d'une tranche en utilisant la remarque ci-dessus. Après avoir calculé le tableau des sommes partielles la fonction ne devra utiliser qu'une seule boucle* `for`*.*

**Question 8**
*Déterminer la complexité de* `tranche_max3` *en fonction de $n$, la taille de* `t`*.*
*On donnera le résultat sous la forme $\mathcal{O}(n^\alpha)$.*

## 2 Diviser pour régner

Nous allons maintenant chercher d'autres algorithmes en utilisant la méthode diviser-pour-régner. On va donc écrire une fonction auxiliaire qui recherche la somme maximale d'une tranche parmi les tranches dont les indices sont compris entre 2 bornes $i$ et $j-1$ :

$$T_{i,j} = \max\{S_{p,q}\ ;\ i \leqslant p \leqslant q \leqslant j\}$$

On aura besoin aussi de deux sommes particulières : la tranche maximale commençante, $TC_{i,j}$, et la tranche maximale finissante, $TF_{i,j}$, définies par

$$TC_{i,j} = \max\{S_{i,q}\ ;\ i \leqslant q \leqslant j\} \text{ et } TF_{i,j} = \max\{S_{p,j}\ ;\ i \leqslant p \leqslant j\}$$

Ce sont les tranches de somme maximale parmi, respectivement, les tranche commençant par $i$ (et finissant avant) et celles finissant par $j-1$ (et commençant en $i$ ou après).

### 2.1 Premier algorithme

**Question 9**
*Prouver que, pour $i \leqslant m \leqslant j$, on a $T_{i,j} = \max\{T_{i,m}, T_{m,j}, TF_{i,m} + TC_{m,j}\}$.*

**Question 10**
*Écrire des fonctions* `max_C : int array -> int -> int -> int` *et*
`max_F : int array -> int -> int -> int` *telles que* `max_C t i j` *(resp.* `max_F t i j`*) renvoie la somme de la tranche maximale commençante (resp. finissante) entre $i$ et $j-1$.*
*Le nombre d'opérations effectuées doit être de l'ordre de $j-i$.*

On peut alors écrire la fonction

```
let tranche_max4 t =
   let rec aux i j =
      if j = i + 1
      then max 0 t.(i)
      else let m = (i + j)/2 in
           let t_maxg = aux i m in
           let t_maxd = aux m j in
           let tf_maxg = max_F t i m in
           let tc_maxd = max_C t m j in
           max (max t_maxg t_maxd) (tf_maxg + tc_maxd)
   in aux 0 (Array.length t);;
```

`t_maxg` (resp. `t_maxd`) est la tranche maximale de la partie gauche (resp droite), `tf_maxg` est la tranche maximale finissante de la partie gauche et `tc_maxd` est la tranche maximale commençante de la partie droite.

On note $C(p)$ le nombre d'opérations (additions et comparaisons) effectuées lors de l'appel de `aux i j` pour $j = i + p$.

**Question 11**
*Prouver que la fonction donne la valeur maximale d'une tranche du tableau pour tout tableau de taille au moins un. (On admettra que les fonctions* `max_c` *et* `max_f` *fournissent le bon résultat.)*

**Question 12**
*Prouver que si $n \div 2$ désigne le quotient entier de $n$ par 2, on a $\lfloor \frac{n}{2} \rfloor = n \div 2$ et $\lceil \frac{n}{2} \rceil = n - n \div 2$.*

**Question 13**
*Prouver qu'on a $C(p) = C\big(\lfloor \frac{p}{2} \rfloor\big) + C\big(\lceil \frac{p}{2} \rceil\big) + 2p$.*

## 2.2 Master theorem

Lorsque l'on écrit un algorithme avec une stratégie diviser-pour-régner, la complexité vérifie souvent une propriété de la forme $C(n) \leqslant C\big(\lfloor \frac{n}{2} \rfloor\big) + C\big(\lceil \frac{n}{2} \rceil\big) + Kn^\alpha$ avec $K$ et $\alpha$ réels positifs.
C'est le cas dans la partie ci-dessus, avec $K = 2$ et $\alpha = 1$.

**Question 14**
*Prouver que si on a $2^p \leqslant n \leqslant 2^{p+1}$ alors $2^{p-1} \leqslant \lfloor \frac{n}{2} \rfloor \leqslant \lceil \frac{n}{2} \rceil \leqslant 2^p$.*

**Question 15**
*On pose $u_p = 2^{-p} \sup\big\{C(n)\ ;\ 2^p \leqslant n \leqslant 2^{p+1}\big\}$. Prouver qu'on a $u_p \leqslant u_{p-1} + 2^\alpha.K.2^{p(\alpha-1)}$.*

**Question 16**
*En déduire qu'on a, pour tout $n \geqslant 1$, en posant $A = \max\{C(1), C(2)\}$ et $B = \dfrac{2^\alpha.2^{\alpha-1}}{2^p(2^{\alpha-1}-1)}$,*

- *$C(n) \leqslant (A + K)n$ donc $C(n) = \mathcal{O}(n)$ pour $\alpha = 0$, la complexité est linéaire.*
- *$C(n) \leqslant 2Kn\log_2(n) + An$ donc $C(n) = \mathcal{O}\big(n\ln(n)\big)$ pour $\alpha = 1$.*
- *$C(n) \leqslant A.n + K.B.n^\alpha$ donc $C(n) = \mathcal{O}\big(n^\alpha\big)$ pour $\alpha > 1$.*

Ainsi la complexité de `tranche_max4` vérifie $C(n) = \mathcal{O}\big(n\ln(n)\big)$.

## 2.3 Complexité linéaire

On a trouvé un algorithme de complexité quasi-linéaire.
Pour parvenir à une complexité linéaire, il faudrait que la complexité des calculs dans la fonction en dehors des appels récursif soit bornée. Pour cela on peut espérer calculer $TC_{i,j}$ et $TF_{i,j}$ en utilisant aussi une une stratégie diviser-pour-régner.

**Question 17**
*Prouver que, pour $i \leqslant m \leqslant j$, on a $TC_{i,j} = \max\big\{TC_{i,m}, S_{i,m} + TC_{m,j}\big\}$.*
*Donner, sans démonstration, une formule analogue pour $TF_{i,j}$.*

On remarque donc qu'on doit calculer 4 valeurs entre $i$ et $j$, $T_{i,j}$, $TC_{i,j}$, $TF_{i,j}$ et $S_{i,j}$, pour les combiner récursivement.

**Question 18**
*En modifiant* `tranche_max4`*, écrire une fonction* `tranche_max5 : int array -> int` *telle que* `tranche_max5 t` *renvoie la somme maximale d'une tranche en utilisant la remarque ci-dessus.*

**Question 19**
*Déterminer la complexité de* `tranche_max5`*.*

# 3 Listes

Dans cette partie la suite des valeurs est codée en mémoire par une liste.
L'exemple du début sera donc enregistré sous la forme

```
let l_ex = [4; -5; 2; -1; 3; 2; -7; 3];;
```

On n'utilisera les listes qu'au travers d'un pattern-matching (ou éventuellement `List.hd` et `List.tl`).

```
match liste with
|[] ->
|t::q ->
```

L'usage de `List.length`, `List.nth`, ...est interdit.

La somme maximale d'une tranche d'une liste non vide, de la forme `t::q` est

- soit la somme d'une tranche incluse dans `q`, donc la somme maximale d'une tranche de `q`,
- soit la somme d'une tranche contenant `t`, la somme maximale parmi les tranches initiales.

Pour simplifier les algorithmes on inclura le cas de la tranche vide parmi les tranches initiales : le maximum des sommes des tranches initiales sera donc toujours positif.

**Question 20**
*Écrire une fonction* `max_init : int list -> int` *telle que* `max_init liste` *renvoie la somme maximale d'une tranche parmi les tranches initiales, c'est-à-dire parmi les tranches contenant le premier terme ou la tranche vide.*
*On justifiera la validité de la fonction.*

**Question 21**
*En déduire une fonction* `tranche_max6 : int list -> int` *telle que* `tranche_max6 liste` *renvoie la somme maximale d'une tranche de la liste.*

**Question 22**
*Quelle est la complexité de cet algorithme ?*

**Question 23**
*Écrire une fonction* `tranche_max7 : int list -> int` *telle que* `tranche_max7 liste` *renvoie la somme maximale d'une tranche de la liste avec une complexité linéaire.*

**Question 24**
*En s'inspirant de la méthode ci-dessus, écrire une fonction* `tranche_max8 : int array -> int` *telle que* `tranche_max8 t` *renvoie la somme maximale d'une tranche du tableau.*

# 4 Termes de la somme maximale

**Question 25**
*Modifier la fonction* `tranche_max8` *(ou la fonction* `tranche_max3`*) pour qu'elle renvoie, en plus de la somme maximale d'une tranche, deux indices $i$ et $j$ tels que cette somme maximale soit $S_{i,j}$.*
*La complexité doit rester linéaire.*

**Question 26**
*Modifier la fonction* `tranche_max7` *pour qu'elle renvoie, en plus de la somme maximale d'une tranche, la liste extraite de la liste initiale et contenant la tranche dont on a calculé la somme.*
*La complexité doit rester linéaire.*

# 5 Généralisation à 2 dimensions

On peut généraliser le problème à deux dimensions.
On se donne un tableau de dimension 2 d'entiers et on cherche le rectangle tel que la somme des termes du rectangle est maximale.
Un tableau à deux dimensions est représenté par une matrice `tt`.
Une matrice d'entiers de taille $n \times m$ est créée par

```
let tt = Array.make_matrix n m 0;;
```

C'est en fait un tableau de tableaux, `int array array`.
On accède à un terme (en lecture ou en écriture) par `tt.(i).(j)`.
Le nombre de lignes (premiers indices) est accessible par

```
let n = Array.length tt;;
```

Le nombre de colonnes (seconds indices) est accessible par

```
let m = Array.length tt.(0);;
```

On cherche donc le maximum des sommes $\rho_{i_1,i_2,j_1,j_2} = \sum_{p=i_1}^{i_2-1} \sum_{q=j_1}^{j_2-1} \texttt{tt.(p).(q)}$ avec $0 \leqslant i_1 \leqslant i_2 \leqslant n$ et $0 \leqslant j_1 \leqslant j_2 \leqslant m$ si `tt` est de taille $n \times m$.

**Question 27**
*En calculant les sommes des termes de tous les rectangles possibles, écrire une fonction* `rectangle_max1 : int array array -> int` *telle que* `rectangle_max1 tt` *renvoie la somme maximale des termes d'un rectangle pour les rectangles extraits du tableau représenté par* `tt`.
*Quelle est la complexité, en nombre d'additions et de comparaisons, en fonction des deux entier $n$ et $m$ si* `tt` *est de taille $n \times m$ ?*

**Question 28**
*Écrire une fonction* `cumul2 : int array array -> int array array` *telle que* `cumul tt` *renvoie un tableau* `s2`*, de taille $(n+1) \times (m+1)$, si* `tt` *est de taille $n \times m$, et tel que* `s2.(i).(j)` *a pour valeur $\rho_{0,i,0,j} = \sum_{p=0}^{i-1} \sum_{q=0}^{j-1} tt.(p).(q)$ avec la convention de somme nulle si $i = 0$ ou $j = 0$.*
*On donnera une solution de complexité en $\mathcal{O}(n.m)$.*

**Question 29**
*Exprimer $\rho_{i_1,i_2,j_1,j_2}$ en fonction de $\rho_{0,i_2,0,j_2}$, $\rho_{0,i_1,0,j_2}$, $\rho_{0,i_2,0,j_1}$ et $\rho_{0,i_1,0,j_1}$.*
*En déduire une fonction* `rectangle_max2 : int array array -> int` *de calcul de la somme maximale des termes d'un rectangle de complexité $\mathcal{O}(n^2m^2)$.*

**Question 30**
*En utilisant les idées de la partie* **1.3** *pour une des dimensions, écrire une fonction* `rectangle_max3 : int array array -> int` *de calcul de la somme maximale des termes d'un rectangle de complexité $\mathcal{O}(nm^2)$ ou $\mathcal{O}(n^2m)$.*

# 6 Solutions

**Question 1 (Solution)**

```
let somme_tranche t i j =
   let s = ref 0 in
   for k = i to (j-1) do s := !s + t.(k) done;
   !s;;
```

**Question 2 (Solution)**

```
let tranche_max1 t =
   let n = Array.length t in
   let t_max = ref 0 in
   for i = 0 to n do
      for j = i to n do
         t_max := max !t_max (somme_tranche t i j) done done;
   !t_max;;
```

**Question 3 (Solution)** On ajoute à la somme les termes `t.(k)`, pour $k$ variant entre $i$ et $j$ donc on effectue $j-i$ additions lors de de l'appel de `somme_tranche1 t i j`.
À chaque calcul de `t_max` on fait 1 comparaison et un appel de `somme_tranche`. Au total `tranche_max t` effectue $C_1(n)$ additions eavec

$$C_1(n) = \sum_{i=0}^{n}\sum_{j=i}^{n}(j-i) = \sum_{i=0}^{n}\sum_{k=0}^{n-i}k = \sum_{i=0}^{n}\frac{(n-i)(n-i+1)}{2} = \sum_{p=0}^{n}\frac{p(p+1)}{2} = \frac{n(n+1)(n+2)}{6}$$

où on a posé $k = j-i$ et $p = n-i$.

Le nombre de comparaisons est $C_2(n) = \sum_{i=0}^{n}\sum_{j=i}^{n}1 = \sum_{i=0}^{n}(n-i+1) = \sum_{k=1}^{n+1}k = \frac{(n+1)(n+2)}{2}$

La complexité totale est $\frac{(n+1)(n+2)(n+3)}{6} = \mathcal{O}(n^3)$.

**Question 4 (Solution)**

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

**Question 5 (Solution)**

```
let tranche_max2 t =
   let n = Array.length t in
   let s = cumul t in
   let t_max = ref 0 in
   for i = 0 to n do
      for j = i to n do
         t_max := max !t_max (s.(j) - s.(i)) done done;
   !t_max;;
```

**Question 6 (Solution)** Après avoir fait $n$ additions pour calculer le tableau des sommes partielles on effectue une soustraction et 2 comparaisons pour tout couple $(i,j)$ avec $0 \leqslant i \leqslant j \leqslant n$ donc moins de $n + 3(n+1)^2$ opérations : la complexité est un $\mathcal{O}(n^2)$.

**Question 7 (Solution)**

```
let tranche_max3 t =
   let n = Array.length t in
   let s = cumul t in
   let mini = ref 0 in
   let t_max = ref 0 in
   for j = 0 to n do
      mini := min !mini s.(j);
      t_max := max !t_max (s.(j) - !mini) done;
   !t_max;;
```

**Question 8 (Solution)** Après avoir fait $n$ additions pour calculer le tableau des sommes partielles on effectue $n+1$ fois une soustraction et deux comparaisons : la complexité est un $\mathcal{O}(n)$.

**Question 9 (Solution)** Il existe des entiers $p$ et $q$ tels que $TF_{i,m} = \sum_{k=p}^{m-1} a_k$ et $TC_{m,j} = \sum_{k=m}^{q-1} a_k$ donc $TF_{i,m} + TC_{m,j} = \sum_{k=p}^{q-1} a_k$. $T_{i,m}$, $T_{m,j}$ et $TF_{i,m} + TC_{m,j}$ sont donc toutes 3 des sommes de tranches d'indices compris entre $i$ et $j-1$ donc sont majorées par la somme de tranche maximale : on a $\max\{T_{i,m}, T_{m,j}, TF_{i,m} + TC_{m,j}\} \leqslant T_{i,j}$.

On a $T_{i,j} = \sum_{k=p}^{q-1} a_k$ avec $i \leqslant p \leqslant q \leqslant j$.

- Si on a $q \leqslant m$ alors $T_{i,j}$ est la somme d'une tranche d'indices compris entre $i$ et $m-1$ donc $T_{i,j} \leqslant T_{i,m}$ (il y a en fait égalité).
- De même, si on a $m \leqslant i$ alors $T_{i,j} \leqslant T_{m,j}$.
- Si on a $i < m < j$ alors $T_{i,j} = \sum_{k=p}^{m-1} a_k + \sum_{k=m}^{q-1} a_k$ avec $\sum_{k=p}^{m-1} a_k \leqslant TF_{i,m}$ et $\sum_{k=m}^{q-1} a_k \leqslant TC_{m,j}$ donc $T_{i,j} \leqslant TF_{i,m} + TC_{m,j}$.

De ces 3 majorations on déduit $T_{i,j} \leqslant \max\{T_{i,m}, T_{m,j}, TF_{i,m} + TC_{m,j}$. On conclut donc à l'égalité.

**Question 10 (Solution)**

```
let max_C t i j =
   let tc_max = ref 0 in
   let tc = ref 0 in
   for k = i to (j-1) do
      tc := !tc + t.(k);
      if !tc > !tc_max then tc_max := !tc done;
   !tc_max;;
```

```
let max_F t i j =
   let tf_max = ref 0 in
   let tf = ref 0 in
   for k = (j-1) downto i do
      tf := !tf + t.(k);
      if !tf > !tf_max then tf_max := !tf done;
   !tf_max;;
```

On effectue $j-i$ additions et $j-i$ comparaisons dans chaque fonction.

**Question 11 (Solution)** On prouve que `aux i j` fournit la somme maximale d'une tranche parmi les tranches dont les indices sont compris entre 2 bornes $i$ et $j-1$ par récurrence sur $p = j-i$.

- Pour $p = 1$ on est dans le cas où il n'y a qu'un seul terme à considérer : $a_i$.
  Si $a_i \leqslant 0$, la somme maximale est la somme vide qui vaut 0, sinon la somme maximale est $a_i$. Le résultat à renvoyer est bien $\max(0, a_i)$.
- On suppose que le résultat est valide pour tous $i$ et $j$ tels que $1 \leqslant j - i < p$ avec $p \leqslant 2$.
  Soient $i$ et $j$ tels que $j - i = p$. On prend les notations de la fonction.
  On a $j \geqslant i + 2$ d'où $2i + 2 \leqslant i + j \leqslant 2j - 2$ puis $i + 1 \leqslant \frac{i+j}{2} \leqslant j - 1$.
  On conclut qu'on a $i+1 \leqslant m \leqslant j-1$ puis $1 \leqslant m-i \leqslant j-i-1 < p$ et $1 \leqslant j-m \leqslant j-i-1 < p$.
  D'après l'hypothèse de récurrence $m_1$ est bien la somme maximale entre $i$ et $m - 1$ et $m_2$ est bien la somme maximale entre $m$ et $j - 1$. Comme les sommes commençante et finissante sont exactes elles aussi, l'exercice 9 permet de conclure que `aux i j` renvoie bien la somme maximale d'une tranche dans ce cas.

On a prouvé que `aux i (j+p)` renvoie bien la somme maximale d'une tranche par une récurrence généralisée sur $p$.

**Question 12 (Solution)** On démontre le résultat selon la parité de $n$.

- Si $n = 2m$ alors $m = n \div 2 = \frac{n}{2}$ donc $\lfloor \frac{n}{2} \rfloor = m = n \div 2$ et $\lceil \frac{n}{2} \rceil = m = n - n \div 2$.
- Si $n = 2m+1$ alors $\frac{n}{2} = m+\frac{1}{2}$ et $n \div 2 = m$ donc $\lfloor \frac{n}{2} \rfloor = m = n \div 2$ et $\lceil \frac{n}{2} \rceil = m+1 = n-n \div 2$.

**Question 13 (Solution)** Si $j + p$ alors $\frac{i+j}{2} = i + \frac{p}{2}$ donc $m - i = \lfloor \frac{p}{2} \rfloor$ donc l'appel de `aux i m` effectue $C\big(\lfloor \frac{p}{2} \rfloor\big)$ opérations et $j - m = p - \lfloor \frac{p}{2} \rfloor$ d'où l'appel de `aux i m` effectue $C\big(\lceil \frac{p}{2} \rceil\big)$ opérations.
Les appels de `max_F t i m` et `max_C t m j` effectuent respectivement $2(m - i)$ et $2(j - m)$ opérations donc $2p$ au total d'où la formule.

**Question 14 (Solution)**

- Si $n = 2m$, en divisant les inégalités de $n$ par 2,
  $2^{p-1} \leqslant m = \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil < 2^p$.
- Si $n$ est impair alors $2^p + 1 \leqslant n \leqslant 2^{p+1} - 1$ donc $2^{p-1} \leqslant \frac{n-1}{2} = \lfloor \frac{n}{2} \rfloor \leqslant 2^p - 1$ et
  $2^{p-1} + 1 \leqslant \frac{n+1}{2} = \lceil \frac{n}{2} \rceil \leqslant 2^p$.

**Question 15 (Solution)** On note $I_p$, l'intervalle $\big[2^p; 2^{p+1}\big]$.
Pour tout $n \in I_p$, on a $\lfloor \frac{n}{2} \rfloor \in I_{p-1}$ donc $C\big(\lfloor \frac{n}{2} \rfloor\big) \leqslant 2^{p-1}.u_{p-1}$ et, de même, $C\big(\lceil \frac{n}{2} \rceil\big) \leqslant 2^{p-1}.u_{p-1}$ donc
$2^{-p}C(n) \leqslant 2^{-p}\big(2^{p-1}.u_{p-1} + 2^{p-1}.u_{p-1} + K.(2^{p+1})^\alpha\big) = u_{p-1} + K.2^{\alpha+p\alpha-p}$.

**Question 16 (Solution)** On a $I_0 = [1; 2]$ donc $u_0 = \max\{C(1), C(2)\} = A$.
Pour $n \in I_p$, on a $2^p \leqslant n$ donc $p \leqslant \log_2(n)$ et $C(n) \leqslant 2^p u_p$.
On a $u_p \leqslant A + \displaystyle\sum_{k=1}^{p} K.2^\alpha.\big(2^{\alpha-1}\big)^k$.

- Pour $\alpha = 1$, on aboutit à $u_p \leqslant A + \displaystyle\sum_{k=1}^{p} K.2 = A + 2Kp$ donc
  $C(n) \leqslant 2^p.A0 + 2^p.2Kp \leqslant n\big(A + K\log_2(n)\big)$.
- Pour $\alpha = 0$, on aboutit à $u_p \leqslant A + \displaystyle\sum_{k=1}^{p} K.2^{-k} = A + K(1 - 2^{-p}) \leqslant A + K$ donc $C(n) \leqslant 2^p.(A + u_0) \leqslant n(A + u_0)$.
- Pour $\alpha > 1$, la somme géométrique de raison $r = 2^{\alpha-1} > 1$ donne
  $u_p \leqslant A + K.2^\alpha.2^{\alpha-1}\dfrac{\big(2^{\alpha-1}\big)^p - 1}{2^{\alpha-1} - 1} \leqslant A + \dfrac{K.2^\alpha.2^{\alpha-1}2^{\alpha p}}{2^p(2^{\alpha-1} - 1)}$.
  On a alors $C(n) \leqslant A.2^p + K.B.(2^p)^\alpha \leqslant A.n + K.B.n^\alpha$.

**Question 17 (Solution)** La tranche maximale commençante est $S_{i,k}$.

- Si on a $k \leqslant m$ alors $S(i, k)$ est la somme d'une tranche commençante entre $i$ et $m$ d'où $S(i, k) \leqslant TC_{i,m}$.
- Si on a $k \geqslant m$ alors $S(i, k) = S(i, m) + S(m, k)$ et $S(m, k)$ est la somme d'une tranche commençante entre $m$ et $j$ d'où $S(m, k) \leqslant TC_{m,j}$.

On a donc dans tous les cas $TC_{i,j} = S_{i,j} \leqslant \max\{TC_{i,m}, S_{i,m} + TC_{m,j}\}$.
Inversement $TC_{i,m}$ et $S_{i,m} + TC_{m,j}$ sont tous deux des sommes de tranches commençantes donc sont majorées par $TC_{i,j}$ d'où $\max\{TC_{i,m}, S_{i,m} + TC_{m,j}\} \leqslant TC_{i,j}$.
On en déduit l'égalité.
On a aussi $TF_{i,j} = \max\{TF_{i,m} + S_{m,j}, TF_{m,j}\}$.

**Question 18 (Solution)** Signification des variables :

| | partie gauche | partie droite | résultat global |
|---|---|---|---|
| tranche maximale | `tmg` | `tmd` | `tm` |
| tranche commençante maximale | `tcg` | `tcd` | `tc` |
| tranche finissante maximale | `tfg` | `tfd` | `tf` |
| somme totale | `sg` | `sd` | `s` |

```
let tranche_max5 t =
   let rec aux i j =
      if j = i + 1
      then let tm = max 0 t.(i) in tm, tm, tm, t.(i)
      else let m = (i + j)/2 in
           let tmg, tcg, tfg, sg = aux i m in
           let tmd, tcd, tfd, sd = aux m j in
           let tm = max (max tmg tmd) (tfg + tcd) in
           let tc = max tcg (sg + tcd) in
           let tf = max (tfg + sd) tfd in
           let s = sg + sd in
           tm, tc, tf, s
   in let tm, tc, tf, s = aux 0 (Array.lenght t) in tm;;
```

**Question 19 (Solution)**
Si on note $C(p)$ le nombre d'additions effectuées on a $C(p) = C\left(\lfloor \frac{p}{2} \rfloor\right) + C\left(\lceil \frac{p}{2} \rceil\right) + 8$.
Le résultat de la question 16 permet de conclure à une complexité linéaire.

**Question 20 (Solution)**

```
let rec max_init liste =
   match liste with
   |[] -> 0
   |t::q -> let tiq = max_init q in max (t + miq) 0;;
```

**Question 21 (Solution)**

```
let rec tranche_max6 liste =
   match liste with
   |[] -> 0
   |t::q -> max (tranche_max6 q) (max_init liste);;
```

**Question 22 (Solution)** `max_init` effectue une addition et une comparaison pour chaque élément de la liste : sa complexité est $2n$ si $n$ est la longueur de la liste.
Si $(n)$ est la complexité pour une liste de taille $n$, on a $C(0) = 0$ et $C(n) = C(n-1) + 2n$ donc $C(n) = n(n+1)$ : la complexité de `tranche_max6` est quadratique.

**Question 23 (Solution)**
On calcule récursivement la tranche initiale maximale et la tranche maximale.

```
let tranche_max7 liste =
   let rec aux reste =
      match reste with
      |[] -> 0, 0
      |t::q -> let t_maxq, ti_maxq = aux q in
               let ti_max = max 0 (t + ti_maxq) in
               (max t_maxq ti_max, ti_max)
   in fst (aux liste);;
```

**Question 24 (Solution)**

```
let tranche_max8 t =
   let n = Array.length t in
   let t_max = ref 0 in
   let fin = ref 0 in
   for i = 0 to (n-1) do
      fin := max 0 (!fin + t.(i));
      t_max := max !t_max !fin done;
   !t_max;;
```

Dans la pratique, pour un tableau aléatoire de 2500 valeurs, on aboutit à

```
temps pour tranche_max1 : 37.622963s
temps pour tranche_max2 :  0.103076s
temps pour tranche_max3 :  0.000696s
temps pour tranche_max4 :  0.001546s
temps pour tranche_max5 :  0.000931s
temps pour tranche_max6 :  0.078516s
temps pour tranche_max7 :  0.000609s
temps pour tranche_max8 :  0.000591s
```

**Question 25 (Solution)**

```
let indices_max3 t =
   let n = Array.length t in
   let s = cumul t in
   let mini = ref 0 in
   let t_max = ref 0 in
   let i_max = ref 0 in
   let j_max = ref 0 in
   let i_min = ref 0 in
      for j = 0 to n do
      if s.(j) <= !mini
      then (mini := s.(j); i_min := j);
      if s.(j) - !mini  > !t_max
      then (t_max := s.(j) - !mini;
            i_max := !i_min; j_max := j) done;
   !t_max, !i_max, !j_max;;
```

```
let indices_max8 t =
   let n = Array.length t in
   let t_max = ref 0 in
   let fin = ref 0 in
   let i_max = ref 0 in
   let j_max = ref 0 in
   let i_deb = ref 0 in
   for i = 0 to (n-1) do
      if !fin + t.(i) > 0
      then fin := !fin + t.(i)
      else (fin := 0; i_deb := i+1);
      if !t_max < !fin
      then (t_max := !fin; i_max := !deb; j_max := i+1) done;
    !t_max, !i_max, !j_max;;
```

```
let indices_max5 t =
   let rec aux i j =
      if j = i + 1
      then begin if t.(i) > 0
                 then t.(i), i, j, t.(i), j, t.(i), i, t.(i)
                 else 0, i, i, 0, i, 0, j, t.(i) end
      else let m = (i + j)/2 in
           let tm1,im1,jm1,tc1,fn1,tf1,db1,s1 = aux i m in
           let tm2,im2,jm2,tc2,fn2,tf2,db2,s2 = aux m j in
           let tm = ref 0 in
           let im = ref 0 in
           let jm = ref 0 in
           let tc = ref 0 in
           let fn = ref 0 in
           let tf = ref 0 in
           let db = ref 0 in
           if tf1 + tc2 > max tm1 tm2
           then (tm := tf1 + tc2; im := db1; jm := fn2)
           else begin if tm1 > tm2
                      then (tm := tm1; im := im1; jm := jm1)
                      else (tm := tm2; im := im2; jm := jm2)
                end;
           if tc1 > s1 + tc2
           then (tc := tc1; fn := fn1)
           else (tc := s1 + tc2; fn := fn2);
           if tf2 > tf1 + s2
           then (tf := tf2; db := db2)
           else (tf := tf1 + s2; db := db1);
           !tm, !im, !jm, !tc, !fn, !tf, !db, (s1 + s2) in
   let tm, im, jm, tc, fn, tf, db, s = aux 0 (Array.length t)
   in tm, im, jm;;
```

**Question 26 (Solution)**

```
let liste_max7 liste =
   let rec aux reste =
      match reste with
      |[] -> 0, [], 0, []
      |t::q -> let t_max, l_max, ti_max, l_i = aux q in
               match t + ti_max with
               |ti when ti < 0 -> t_max, l_max, 0, []
               |ti when ti < t_max -> t_max, l_max, ti, t::l_i
               |ti -> ti, t::l_i, ti, t::l_i
   in let t_max, l_max, ti_max, l_i = aux liste in t_max,
      l_max;;
```

**Question 27 (Solution)** On commence par un calcul de la somme des termes d'un rectangle.

```
let somme_rect tt i1 i2 j1 j2 =
   let s = ref 0 in
   for p = i1 to (i2-1) do
      for q = j1 to (j2 - 1) do
         s := !s + tt.(p).(q) done done;
   !s;;

let rectangle_max1 tt =
  let n = Array.length tt in
  let m = Array.length tt.(0) in
  let r_max = ref 0 in
  for i1 = 0 to n do
    for i2 = i1 to n do
      for j1 = 0 to m do
        for j2 = i2 to m do
          let s = somme_rect tt i1 i2 j1 j2 in
          if s > !r_max then r_max := s done done done done;
  !r_max;;
```

Pour les additions on dédouble les calculs de `tranche_max1` ; on effectue ainsi $\frac{n(n+1)(n+2)}{6}\frac{m(m+1)(m+2)}{6}$ additions. Le nombre de comparaisons est majoré par $(n+1)^2(m+1)^2$ donc la complexité est on $\mathcal{O}\big(n^3m^3\big)$.

**Question 28 (Solution)**

```
let cumul2 tt =
   let n = Array.length tt in
   let m = Array.length tt.(0) in
   let s2 = Array.make_matrix (n+1) (m+1) 0 in
   for i = 1 to n do
      let ligne = ref 0 in
      for j = 1 to m do
         ligne := !ligne + tt.(i-1).(j-1);
         s2.(i).(j) <- s2.(i-1).(j) + ! ligne done done;
   s2;;
```

**Question 29 (Solution)** $\rho_{i_1,i_2,j_1,j_2} = \rho_{0,i_2,0,j_2} - \rho_{0,i_1,0,j_2} - \rho_{0,i_2,0,j_1} + \rho_{0,i_1,0,j_1}$.

```
let rectangle_max2 tt =
  let n = Array.length tt in
  let m = Array.length tt.(0) in
  let s2 = cumul2 tt in
  let r_max = ref 0 in
  for i1 = 0 to n do
    for j1 = i1 to n do
      for i2 = 0 to m do
        for j2 = i2 to m do
          let s = s2.(i2).(j2) - s2.(i1).(j2)
                    - s2.(i2).(j1) + s2.(i1).(j1) in
          if s > !r_max then r_max := s done done done done;
  !r_max;;
```

**Question 30 (Solution)**

```
let rectangle_max3 tt =
  let n = Array.length tt in
  let m = Array.length tt.(0) in
  let s2 = cumul2 tt in
  let r_max = ref 0 in
    for j1 = 0 to m do
      for j2 = j1 to m do
        let mini = ref 0 in
        let r_maxj1j2 = ref 0 in
          for i = 0 to n do
            let bloc = s2.(i).(j2) - s2.(i).(j1) in
              mini := min !mini bloc;
              r_maxj1j2 := max !r_maxj1j2 (bloc - !mini) done;
          r_max := max !r_max !r_maxj1j2 done done;
    !r_max;;
```

Dans la pratique, pour un tableau aléatoire de $60 \times 60$ valeurs, on aboutit à

```
temps pour rectangle_max1 : 26.929931s
temps pour rectangle_max2 :  0.144597s
temps pour rectangle_max3 :  0.0078669s
```

## Devoir C

# DS3 : Tableaux dynamiques

Dans ce sujet nous allons définir plusieurs implémentations possibles des tableaux dynamiques. Ce sont des tableaux dans lesquels il est possible d'adjoindre un éléments, l'exemple connu étant les le type liste de Python.

On veut donc définir un type `'a tab_dyn` pour les tableaux dynamiques dont les éléments sont de type $\alpha$ (`'a`). Un tableau de taille $n$ contient $n$ éléments qui sont accessibles par leur indice qui varie entre 0 et $n-1$. On veut aussi écrire les fonctions associées.

- `creer : int -> 'a -> 'a tab_dyn`
  `creer n x` renvoie un tableau de taille $n$ dont les éléments sont tous égaux à $x$.
  C'est l'équivalent de `Array.make n x`.
- `taille : 'a tab_dyn -> int`
  `taille td` renvoie la taille du tableau dynamique `td`. C'est l'équivalent de `Array.length td`.
- `lire : int -> 'a tab_dyn -> 'a`
  `lire i td` renvoie la valeur à l'indice $i$ du tableau dynamique `td`. C'est l'équivalent de `t.(i)`.
- `ecrire : int -> 'a -> 'a tab_dyn -> unit`
  `ecrire i x td` remplace la valeur à l'indice $i$ du tableau dynamique `td` par $x$, l'ancienne valeur est perdue. C'est l'équivalent de `td.(i) <- x`.
- `adjoindre : 'a -> 'a tab_dyn -> unit`
  `adjoindre x t` ajoute un élément à `td`, il sera à l'indice $n$ si $n$ était la taille du tableau dynamique avant l'adjonction. La taille du tableau dynamique est changée, elle vaut ensuite $n+1$. C'est l'équivalent de `t.append(x)` en python.
- `enlever : 'a tab_dyn -> 'a`
  Pour un tableau dynamique `td` de taille $n$, `enlever td` renvoie l'élément l'indice $n-1$ et retire cet élément du tableau dynamique dont la taille passe à $n-1$. C'est l'équivalent de `td.pop()` en python.

Il y aura donc, comme en python, 2 manières de créer le tableau dynamique des carrés des entiers de 0 à $n-1$

```
let carres n =
   let td = creer n 0 in
   for i = 0 to (n-1) do
      ecrire i (i*i) td done;
   td;;
```

```
let carres n =
   let td = creer 0 0 in
   for i = 0 to (n-1) do
     adjoindre (i*i) td done;
   td;;
```

**Erreurs** Si $i$ est strictement négatif ou supérieur ou égal à la taille de `td`, `lire i td` ou `ecrire i td` devra renvoyer une erreur : `failwith "Indice hors des bornes"`.
Si la taille de `td` est nulle `enlever td` devra renvoyer une erreur : `failwith "Tableau vide"`.

**Tableaux OCaml** Dans le sujet on nommera **statiques** les tableau de type `'a array` de OCaml.
On rappelle que les indices varient ente 0 et $n-1$ si $n$ est la taille du tableau.
Les opérations dans un tableau statique sont rappelées dans la page précédente

**Enregistrement** Si on définit un enregistrement

```
type 'a truc = {mutable x : int;
                        y : 'a ;
                        t : 'a array};;
```

les éléments d'un objet `mon_truc` sont obtenus respectivement par `mon_truc.x` , `mon_truc.y` et `mon_truc.`. On modifie une composante **mutable**, par exemple l'incrément de `x`, par `mon_truc.x <- mon_truc.x + 1`.
Si on a défini `tableau = mon_truc.t`, toute modification de `tableau` sera répercutée dans `mon_truc` ; `tableau` n'est pas indépendant de l'enregistrement.

**Complexité** La complexité, ou le coût, d'un programme $P$ (fonction ou procédure) est le nombre d'opération élémentaires (addition, affectation, lecture d'un élément, ...) nécessaires à l'exécution de $P$. Lorsque cette complexité dépend d'un paramètre $n$, on dira que $P$ a une complexité $\mathcal{O}\big(f(n)\big)$, s'il existe $K > 0$ tel que la complexité de $P$ est au plus $Kf(n)$, pour tout $n$.
Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.
La complexité d'un algorithme est **constante** si elle est un $\mathcal{O}\big(1\big)$, c'est-à-dire si le nombre d'opération élémentaires est majoré par une constante $K$ indépendante de la taille.
La complexité d'un algorithme est **linéaire** en $n$ si elle est un $\mathcal{O}\big(n\big)$.

**Organisation** Les parties sont voulues comme ayant une difficulté croissante, cependant il est probable que la partie 3.2 (en supposant écrites les fonctions de la partie 3.1) soit plus simple que la partie 3.1.
Dans le corrigé, aucune réponse de code OCaml n'est écrite avec plus de 8 lignes, plus de la moitié des codes sont écrits avec moins de 5 lignes.

**Conseils** N'écrivez **jamais** un code sur plusieurs pages.
La concaténation pour les listes, `x::liste` **n'est pas** commutative, `x` est un objet de type `'a` et `liste` est une `'a list`
**Il est capital de ne pas confondre tableaux et listes OCaml** : en particulier `liste.(i)` n'est pas défini pour une liste et `c::tableau` n'a pas de sens pour un tableau.
La question 13 revient à écrire une fonction `List.nth`, bien entendu il est demandé de ne pas utiliser `List.nth`.
Si un code contient beaucoup de lignes ou beaucoup de lignes, outre le fait qu'il est plus susceptible d'être incorrect, il est difficile à lire. N'hésitez pas à commenter en expliquant ce que vous faites et à quoi servent les variables.
N'écrivez **jamais** un code sur plusieurs pages.

# 1 Tableaux re-dimensionnables

Une première possibilité est de gérer les première fonctions par des "vrais" tableaux. Pour pouvoir ajouter des éléments on va considérer des tableaux plus grands que nécessaires avec un paramètre indiquant quelles sont les indices utiles.

```
type 'a tab_dyn = {mutable taille : int;
                   mutable data : 'a array};;
```

Le tableau est lui même mutable car on va devoir le remplacer en cas d'adjonctions en trop grand nombre.
Dans toute cette partie les tableaux statiques devront avoir une taille non nulle.

## 1.1 Premières fonctions

La création d'un tableau dynamique demande de prévoir de la place. On peut écrire

```
let creer n x =
   let n1 = max 4 (2*n) in
   let t = Array.make n1 x in
   {taille = n; data = t};;
```

Par exemple `creer 1 "a"` donne

```
{taille = 1; data = [|"a"; "a"; "a"; "a"|]}
```

et `creer 4 "a"` donne

```
{taille = 4; data = [|"a"; "a"; "a"; "a"; "a"; "a"; "a"; "a"
   |]}
```

On rappelle que la complexité d'un tableau de taille $n$ est proportionnelle à $n$, c'est un $\mathcal{O}(n)$.

**Question 1**
*Écrire la fonction* `taille`.

**Question 2**
*Écrire la fonction* `lire`.

**Question 3**
*Écrire la fonction* `ecrire`.

**Question 4**
*Écrire la fonction* `enlever`.

**Question 5**
*Quelles sont les complexités de ces fonctions ?*

## 1.2 Adjonction

Quand on ajoute un élément à un tableau dynamique de taille $n$, il suffit de le placer à l'indice $n$ du tableau `td.data` et d'incrémenter la taille de 1. Ce sera une opération de complexité constante. Cependant cela n'est possible que lorsque la taille du tableau dynamique est strictement inférieure à celle de `td.data`. Dans le cas contraire la solution proposée est

1. recopier les éléments de `td.data` dans un tableau (statique) deux fois plus grand, on prendra la valeur à l'indice 0 (par exemple) pour remplir le tableau lors de son initialisation,
2. remplacer `td.data` par ce nouveau tableau,
3. continuer comme ci-dessus.

**Question 6**
*Écrire une fonction* `agrandir : 'a array -> 'a array` *qui reçoit un tableau statique de taille $n$ (à calculer dans la fonction) et qui renvoie un nouveau tableau de taille $2n$ dont les $n$ premiers éléments sont ceux du tableau passé en paramètre.*

**Question 7**
*Quelle est la complexité de* `agrandir` *en nombre de lectures et d'écritures, la création d'un tableau de taille $p$ comptant pour $p$ écritures ?*

**Question 8**
*Écrire la fonction* `adjoindre`*. Quelle est sa complexité en nombre de lectures et d'écritures dans le pire des cas en fonction de la taille ?*

La complexité dans le pire des cas peut sembler importante. Cependant le pire des cas n'arrive pas souvent. On peut raisonner en termes de **complexité amortie**

**Question 9**
*Combien de lectures et écritures sont nécessaires pour créer un tableau dynamique de taille 1000 en adjoignant 1000 éléments ?*

```
let n = 1000;;
let td = creer 0 0;;
for i = 0 to (n-1) do adjoindre i td done;;
```

*En moyenne, chaque adjonction à nécessité combien de lectures et d'écritures ?*

**Question 10**
*De manière générale, prouver que l'adjonction de $n$ éléments demande au plus $9n$ opérations d'écriture et de lecture.*

Ainsi la complexité moyenne d'une adjonction est majorée par 6 : le coût moyen est un $\mathcal{O}(1)$

# 2 Listes

On peut aussi implémenter le type de données sous forme d'une liste mutable.

```
type 'a tab_dyn = {mutable taille : int;
                   mutable data : 'a list};;
```

On choisit de placer les éléments "à l'envers" : si $a_i$ est l'élément d'indice $i$ d'un tableau dynamique de taille 6, celui-ci sera implémenté sous la forme

```
{taille = 6; data = [a5; a4; a3; a2; a1; a0]};;
```

## 2.1 Premières fonctions

La fonction `taille` a la même forme que dans le cas des tableaux statiques.

**Question 11**
*Écrire une fonction* **récursive** `repeter : int -> 'a -> 'a list` *telle que* `repeter n x` *renvoie une liste de taille n dont tous les éléments sont égaux à x.*
`repeter 4 "a"` *donne* `["a"; "a"; "a"; "a"]`.
*En déduire une écriture de la fonction* `creer`.

Le choix fait est similaire à l'implémentation d'une pile impérative.
Par exemple on peut écrire

```
let adjoindre x td =
   td.data <- (x :: td.data);
   td.taille <- td.taille + 1;;
```

**Question 12**
*Écrire la fonction* `enlever` *(il est recommandé d'utiliser un pattern-matching).*

## 2.2 Accès séquentiel

Bien entendu la partie difficile sera l'accès aux éléments selon leur indice.

**Question 13**
*Écrire une fonction* **récursive** `k_ieme : int -> 'a list-> 'a` *telle que* `k_ieme i liste` *renvoie l'élément de la liste à la position i, la tête étant à la position 0.*
`k_ieme 4 [2; 5; 9; 7; 3; 8]` *donne* `8`.
*On rappelle qu'on peut faire un pattern-matching sur 2 éléments :* `match x, y with`.

**Question 14**
*En déduire une écriture de la fonction* `lire` *; on n'oubliera pas le retour des messages d'erreur. On notera aussi que le terme d'indice i est à la position $n-1-i$ dans la liste des données si celle-ci est de taille n.*

**Question 15**
*Écrire une fonction* **récursive** `changer_k : int -> 'a -> 'a list-> 'a list` *telle que* `changer_k i x liste` *renvoie la liste obtenue en remplaçant le terme à la position i par x.*
`changer_k 1 6 [2; 5; 9; 7; 3; 8]` *donne* `[2; 6; 9; 7; 3; 8]`.

**Question 16**
*En déduire une écriture de la fonction* `ecrire`.

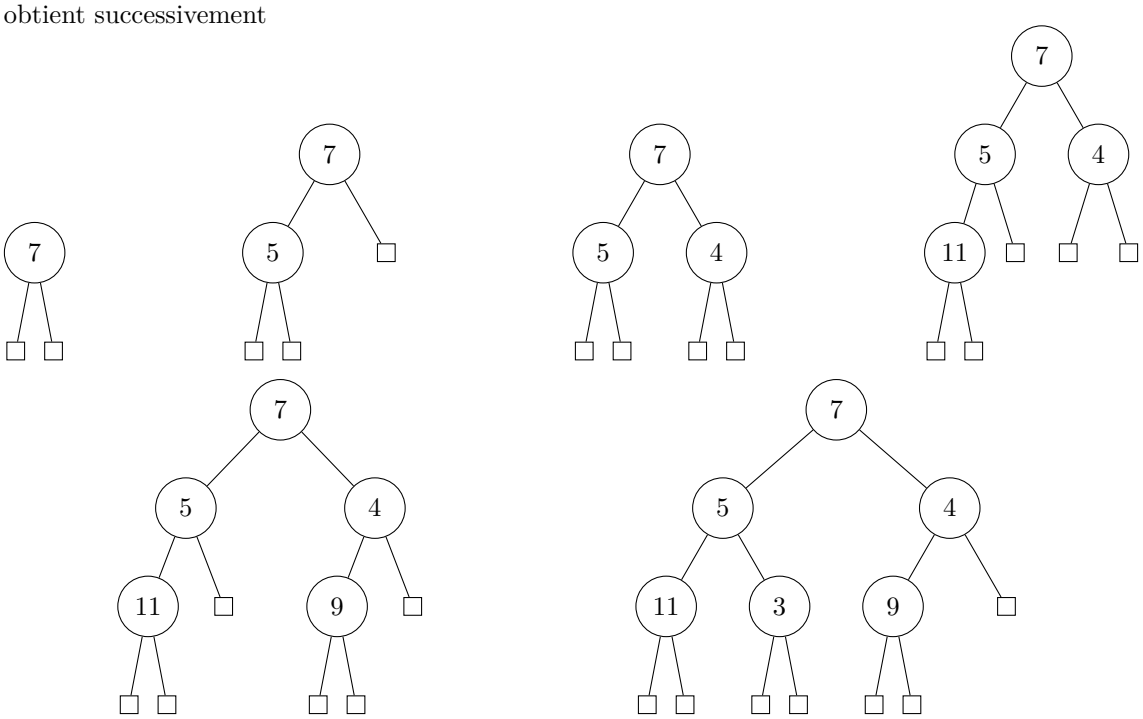
**Question 17**
*Quel est le problème de cette implémentation ?*

# 3 Arbres

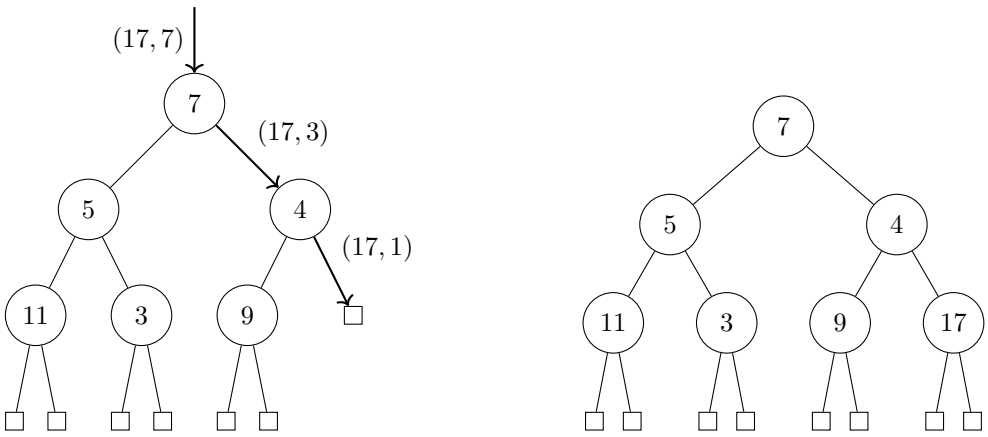
On va maintenant placer des éléments en formant un arbre binaire.
Pour que les algorithmes ne soient pas trop difficiles à lire, nous allons placer, pour l'instant, des éléments indicés à partir de 1 et non 0.
On veut construire un arbre de taille $n$ avec $a_1$, $a_2$, …, $a_n$.

- On place naturellement $a_1$ à la racine puis $a_2$ en racine du fils gauche et $a_3$ en racine du fils droit.
- Si on a placé les éléments de $a_1$ à $a_{k-1}$, $a_k$ va remplacer une feuille vide en suivant la règle suivante : pour placer un élément $x$ associé à un entier $r$ dans un nœud (ou une feuille)
  1. si $r = 0$ le nœud est une feuille et on remplace cette feuille par un nœud de valeur $x$ et de fils tous deux égaux à des feuilles
  2. si $r$ est non nul alors le nœud n'est pas une feuille,
  3. si $r$ est pair on place $x$ dans le fils gauche avec l'entier `r/2`,
  4. si $r$ est impair on place $x$ dans le fils droit avec l'entier `r/2`.

Par exemple pour placer les éléments 7, 5, 4, 11, 9, 3, on part d'un arbre réduit à une feuille en obtient successivement




Pour ajouter un septième élément de valeur 17, on lui associé son rang, 7, et on suit le chemin



On rappelle le type des arbres binaires

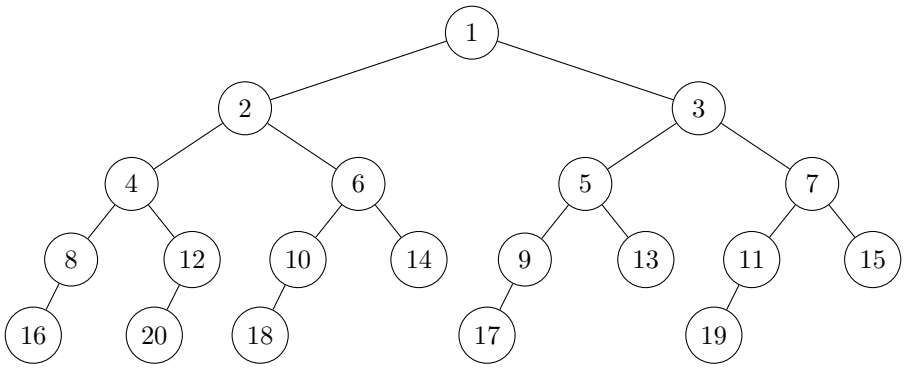
```
type 'a arbre = F | Noeud of 'a arbre * 'a * 'a arbre;;
```

On implémente la procédure ci-dessus, $k$ doit être un entier strictement positif.

```
let rec add k x arbre =
   match k, arbre with
   |1, F  -> Noeud(F, x, F)
   |1,Noeud(g, r, d) -> failwith "Ceci ne devrait pas arriver"
   |k, Noeud(g, r, d) when k mod 2 = 0
                -> Noeud(add (k/2) x g, r, d)
   |k, Noeud(g, r, d) -> Noeud(g, r, add (k/2) x d)
   |k, F -> failwith "Ceci ne devrait pas arriver";;
```

La fonction renvoie une erreur si on aboutit à un nœud avec $k = 1$ ou si on aboutit à une feuille avec $k > 1$.

On appelle **arbre de Braun** un arbre obtenu en ajoutant $n$ éléments à partir d'une feuille avec des entiers $k$ croissant de 1 à $n$ à l'aide de la fonction `add`.
On admet que si `a` est un arbre de Braun de taille $n$ alors la fonction `add (n+1) x a` donne un résultat : ce sera donc un arbre de Braun de taille $n + 1$.
Un élément placé par `add k x ab` est dit de **rang** $k$.
Voici les rang des éléments d'un arbre de Braun de taille 20 (on n'a pas dessiné les feuilles).




Dans un arbre de Braun, les éléments sont accessibles à partir de leur rang :

```
let rec read k ab =
   match k, ab with
   |1, Noeud(F, r, F)  -> r
   |1, _ -> failwith "Ceci ne devrait pas arriver"
   |k, Noeud(g, r, d) when k mod 2 = 0
                -> read (k/2) g
   |k, Noeud(g, r, d) -> read (k/2) d
   |k, F -> failwith "Ceci ne devrait pas arriver";;
```

On admet aussi que cette fonction ne renvoie pas d'erreur si `ab` est un arbre de Braun et si $k \leqslant n$ où $n$ est la taille de `ab`.

**Question 18**
*Représenter l'arbre de Braun obtenu en ajoutant successivement 12, 8, 5, 14, 3, 24, 7, 11, 19, 4 à une feuille.*

## 3.1 Autres fonctions

**Question 19**
*Écrire une fonction* `write : int -> 'a -> 'a arbre -> 'a arbre` *telle que* `write k x ab` *renvoie un nouvel arbre binaire où le terme placé au rang $k$ dans* `ab` *est remplacé par $x$.*
*La fonction n'est assurée de donner un résultat que pour un arbre de Braun de taille $n \geqslant k \geqslant 1$.*

**Question 20**
*Écrire une fonction* `remove : int -> 'a arbre -> 'a arbre` *telle que* `remove n ab` *où* `ab` *est un arbre de Braun de taille $n$ renvoie un nouvel arbre binaire où le terme placé au rang $n$ dans* `ab` *est retiré*
*La fonction n'est assurée de donner un résultat que pour un arbre de Braun de taille $n$. Dans ce cas le résultat est encore un arbre de Braun (de taille $n-1$) ; il n'est pas demandé de le prouver.*

## 3.2 Tableau dynamique

On définit de nouveau un type de tableau dynamique

```
type 'a tab_dyn = {mutable taille : int;
                   mutable data : 'a arbre};;
```

On ne s'autorise sur les arbres que les fonctions `add`, `read`, `write` et `remove`.
Ici encore la fonction `taille` est la même que lors de la question 1.

**Question 21**
*Écrire la fonction* `creer` *; une fonction qui ferait appel à une boucle (for ou while) sera moins valorisée.*

**Question 22**
*Écrire la fonction* `lire`*. On notera que le rang utilisé dans les fonctions n'est pas l'indice $i$, il vaut $i+1$.*

**Question 23**
*Écrire la fonction* `ecrire`*.*

**Question 24**
*Écrire la fonction* `adjoindre`*.*

**Question 25**
*Écrire la fonction* `retirer`*.*

# 4 Solutions

**Question 1 (Solution)**

```
let taille td = td.taille;;
```

**Question 2 (Solution)**

```
let lire i td =
   if i < 0 || i>= td.taille
   then failwith "Indice hors des bornes"
   else td.data.(i);;
```

**Question 3 (Solution)**

```
let ecrire i x td =
   if i < 0 || i>= td.taille
   then failwith "Indice hors des bornes"
   else td.data.(i) <- x;;
```

**Question 4 (Solution)**

```
let enlever td =
   let n = td.taille in
   if n = 0
   then failwith "Tableau vide"
   else begin
        let x = td.data.(n-1) in
        td.taille <- n-1;
        x end;;
```

**Question 5 (Solution)** Ce sont toutes des fonctions de complexité constante.

**Question 6 (Solution)**

```
let agrandir t =
   let n = Array.length t in
   let tt = Array.make (2*n) t.(0) in
   for i = 0 to (n-1) do
      tt.(i) <- t.(i) done;
   tt;;
```

**Question 7 (Solution)** On effectue $2n + n$ lectures et $n$ écritures, la complexité est $4n$.

**Question 8 (Solution)**

```
let adjoindre x td =
   let n = td.taille in
   if n = Array.length td.data
   then td.data <- agrandir td.data;
   td.data.(n) <- x;
   td.taille <- n+1;;
```

Sa complexité est, au pire, $4n + 1$.

**Question 9 (Solution)** Le tableau statique a été dédoublé pour $i = 4$ pour arriver à un tableau de taille 8, puis pour $i = 8$, puis $i = 16$ et ainsi de suite jusqu'à $i = 512$ ; c'est à dire pour $i = 2^p$ avec $2 \leqslant p \leqslant 9$. Il y a eu 1000 écritures des éléments et $\sum_{p=2}^{9} 4.2^p = 2^4 \sum_{k=0}^{7} 2^k = 2^4.(2^8 - 1) = 4080$.
Au total il y a eu 5080 opérations donc $5,08$ opérations par adjonction en moyenne.

**Question 10 (Solution)** Pour $n \leqslant 4$ on effectue $n$ écritures.
Pour $2^m \leqslant n < 2^{m+1}$ avec $m \geqslant 2$ on effectue $n$ écritures et les appels à `agrandir` demandent $\sum_{p=2}^{m} 4.2^p = 2^4 \sum_{k=0}^{m-2} 2^k = 2^4.(2^{m-1} - 1) = 8.2^m - 16 \leqslant 8n$.
Au total, on a effectué au plus $9n$ opérations élémentaires.

**Question 11 (Solution)**

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

let creer n x =
   {taille = n; data = repeter n x};;
```

**Question 12 (Solution)**

```
let enlever td =
   match td.data with
   |[] -> failwith "Tableau vide"
   |x::reste -> td.data <- reste;
                td.taille <- td.taille - 1;
                x;;
```

**Question 13 (Solution)**

```
let rec k_ieme i liste =
   match i, liste with
   |_, [] -> failwith "Indice hors des bornes"
   |0, t::q -> t
   |k, t::q -> k_ieme (k-1) q;;
```

**Question 14 (Solution)**

```
let lire i td =
   let n = td.taille in
   if i < 0 || i >= n
   then failwith "Indice hors des bornes"
   else k_ieme (n - 1 - i) td.data;;
```

**Question 15 (Solution)**

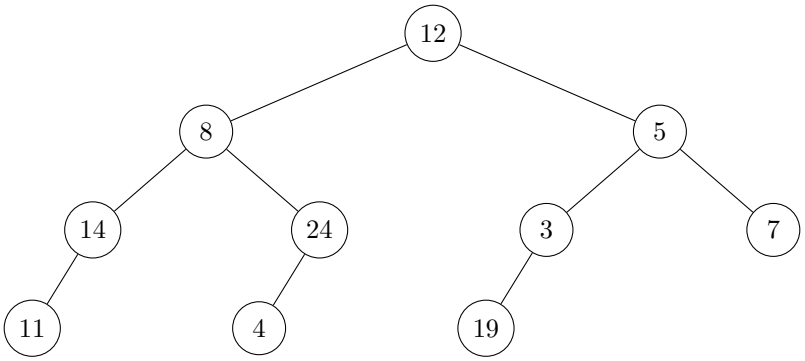
```
let rec changer_k i x liste =
   match i, liste with
   |_, [] -> failwith "Indice hors des bornes"
   |0, t::q -> x::q
   |i, t::q -> t :: (changer_k (i-1) x q);;
```

**Question 16 (Solution)**

```
let ecrire i x td =
   let n = td.taille in
   if i < 0 || i >= n
   then failwith "Indice hors des bornes"
   else td.data <- changer_k (n-1-i) x td.data;;
```

**Question 17 (Solution)** `lire` et `ecrire` ont une complexité de l'ordre de $\frac{n}{2}$.

**Question 18 (Solution)**




**Question 19 (Solution)**

```
let rec write k x ab =
   match k, ab with
   |1, Noeud(g, r, d)  -> Noeud(g, x, d)
   |1, F -> failwith "Ceci ne devrait pas arriver"
   |k, Noeud(g, r, d) when k mod 2 = 0
                -> Noeud(write (k/2) x g, r, d)
   |k, Noeud(g, r, d) -> Noeud(g, r, write (k/2) x d)
   |k, F -> failwith "Ceci ne devrait pas arriver";;
```

**Question 20 (Solution)**

```
let rec remove n ab =
   match n, ab with
   |1, Noeud(g, r, d)  -> F
   |1, F -> failwith "Ceci ne devrait pas arriver"
   |n, Noeud(g, r, d) when n mod 2 = 0
                -> Noeud(remove (n/2) g, r, d)
   |n, Noeud(g, r, d) -> Noeud(g, r, remove (n/2) d)
   |n, F -> failwith "Ceci ne devrait pas arriver";;
```

**Question 21 (Solution)**

```
let creer n x =
   let rec aux k =
       if k = 0
       then F
       else add k x (aux (k-1))
   in {taille = n; data = aux n};;
```

**Question 22 (Solution)**

```
let lire i td =
   let n = td.taille in
   if i < 0 || i>= n
   then failwith "Indice hors des bornes"
   else read (i+1) td.data;;
```

**Question 23 (Solution)**

```
let ecrire i x td =
   let n = td.taille in
   if i < 0 || i>= n
   then failwith "Indice hors des bornes"
   else begin
        td.data <- write (i+1) x td.data;
        td.taille <- n+1 end;;
```

**Question 24 (Solution)**

```
let adjoindre x td =
   let n = td.taille in
   td.data <- add (n+1) x td.data;
   td.taille <- (n+1);;
```

**Question 25 (Solution)**

```
let retirer td =
   let n = td.taille in
   td.data <- remove n td.data;
   td.taille <- (n-1);;
```

## Devoir D

# DS3+ : X-ENS 2012

On étudie dans ce problème des outils pour la combinatoire, qui peuvent être utilisés en particulier pour répondre à des questions telles que :
*combien existe-t-il de façons de paver un échiquier* $8 \times 8$ *par* $32$ *dominos de taille* $2 \times 1$ *?*"

- La partie I introduit la structure d'arbre combinatoire, qui permet de représenter un ensemble d'ensembles d'entiers.
- La partie II étudie quelques fonctions élémentaires sur cette structure.
- La partie III propose ensuite un principe de mémorisation, pour définir des fonctions plus complexes sur les arbres combinatoires.
- La partie IV utilise les résultats précédents pour répondre au problème de dénombrement ci-dessus.
- Enfin, les deux dernières parties expliquent comment construire et manipuler efficacement des arbres combinatoires, à l'aide tables de hachage.

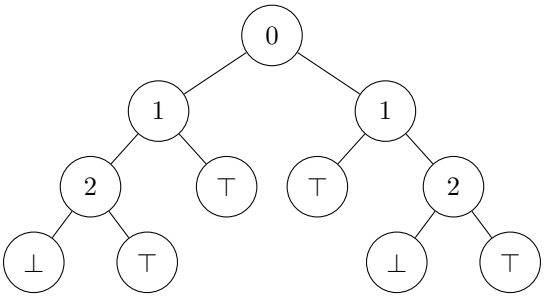
Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et fonctions introduites dans les parties précédentes.

La complexité, ou le coût, d'un programme $P$ (fonction ou procédure) est le nombre d'opération élémentaires (addition, soustraction, multiplication, division, affectation, etc...) nécessaires à l'exécution de $P$. Lorsque cette complexité dépend d'un paramètre $n$, on dira que $P$ a une complexité $O(f(n))$, s'il existe $K > 0$ tel que la complexité de $P$ est au plus $Kf(n)$, pour tout $n$. Lorsqu'il est demandé de garantir une certaine complexité, le candidat devra justifier cette dernière si elle ne se déduit pas directement de la lecture du code.

**Dans l'ensemble de ce problème, on se fixe une constante entière** $n \geqslant 1$**.**
**On note** $E$ **l'ensemble** $\{0, 1, \ldots, n-1\}$**.**

# 1 Arbres combinatoires

Dans cette partie, on étudie les arbres combinatoires, une structure de données pour représenter un élément de $\mathcal{P}\big(\mathcal{P}(E)\big)$, c'est à dire un ensemble de parties de $E$.
Un arbre combinatoire est un arbre binaire dont les nœuds sont étiquetés par des éléments de $E$ et les feuilles par $\bot$ et $\top$.
Voici un exemple d'arbre combinatoire (**exemple 1**) :



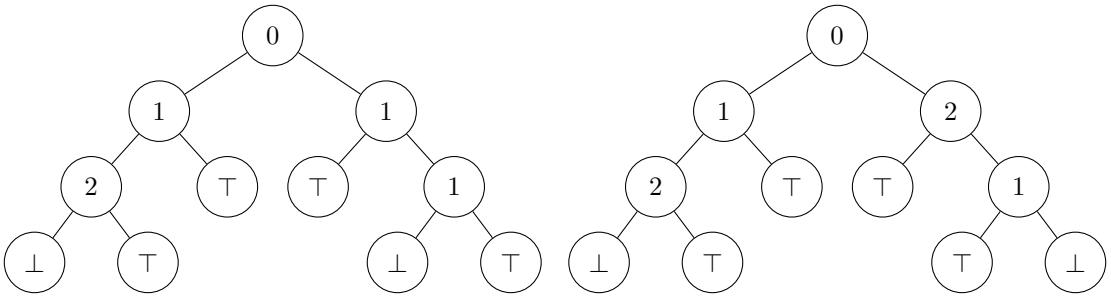

Un nœud étiqueté par $i$, de sous-arbre gauche $A_1$ et de sous-arbre droit $A_2$ sera noté $i \to A_1, A_2$.
L'arbre ci-dessus peut donc également s'écrire sous la forme

$$0 \to (1 \to (2 \to \bot, \top), \top), (1 \to \top, (2 \to \bot, \top)) \tag{1}$$

Dans ce sujet, on impose les propriétés suivantes sur tout (sous-)arbre combinatoire dont la forme est $i \to A_1, A_2$ :

$A_1$ et $A_2$ ne contiennent pas d'élément $j$ avec $j \leqslant i$ ..... *(ordre)*
$A_2 \neq \bot$ ..... *(suppression)*

Ainsi, les deux arbres




ne correspondent pas à des arbres combinatoires, car celui de gauche ne vérifie pas la condition *(ordre)* et celui de droite ne vérifie pas la condition *(suppression)*.

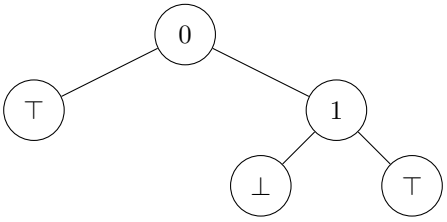
A tout arbre combinatoire $A$ on associe un ensemble de parties de $E$, noté $S(A)$, par

$$S(\bot) = \emptyset, \quad S(\top) = \{\emptyset\}, \quad S(i \to A_1, A_2) = S(A_1) \cup \big\{\{i\} \cup s \ ; \ s \in S(A_2)\big\}$$

L'interprétation d'un arbre $A$ de la forme $i \to A_1, A_2$ est donc la suivante :

- $i$ est le plus petit élément appartenant à au moins un ensemble de $S(A)$
- $A_1$ est le sous-ensemble de $S(A)$ des ensembles qui ne contiennent pas $i$
- $A_2$ est le sous-ensemble de $S(A)$ des ensembles qui contiennent $i$ auxquels on a enlevé $i$

Ainsi, l'arbre suivant est interprété comme l'ensemble $\{\emptyset, \{0, 1\}\}$.



**Question 1**
*Donner l'ensemble défini par l'arbre combinatoire de l'exemple* (1).

**Question 2**
*Donner les trois arbres combinatoires correspondant aux trois ensembles* $\{\{0\}\}$, $\{\emptyset, \{0\}\}$ *et* $\{\{0, 2\}\}$.

**Question 3**
*Soit $A$ un arbre combinatoire différent de $\bot$.*
*Montrer que $A$ contient au moins une feuille $\top$.*

**Question 4**
*Combien existe-t-il d'arbres combinatoires distincts (en fonction de $n$) ?*
*On justifiera soigneusement la réponse.*

## 2 Fonctions élémentaires sur les arbres combinatoires

On se donne le type `ac` suivant pour représenter les arbres combinatoires.

```
type ac = Zero | Un | Comb of int * ac * ac;;
```

Le constructeur `Zero` représente $\bot$ et le constructeur `Un` représente $\top$.

Dans les questions suivantes, une partie $E$ est représentée par la liste de ses éléments, triée par ordre croissant. On note `ensemble` le type correspondant, c'est à dire

```
type ensemble = int list;;
```

On définit le constructeur (provisoirement) par

```
let cons i a b = Comb(i,a,b);;
```

**Question 5**
*Écrire une fonction* `un_elt : ac -> ensemble` *qui prend en argument un arbre combinatoire $A$, supposé différent de $\bot$, et qui renvoie un ensemble $s \in S(A)$ arbitraire.*
*La complexité doit être linéaire en la hauteur de $A$.*

**Question 6**
*Écrire une fonction* `singleton : ensemble -> ac` *qui prend en argument un ensemble $s \in \mathcal{P}(E)$ et qui renvoie l'arbre combinatoire représentant le singleton $\{s\}$.*
*La complexité doit être linéaire en le nombre d'éléments de $s$.*

**Question 7**
*Écrire une fonction* `appartient : ensemble -> ac -> bool` *qui prend en argument un ensemble $s \in \mathcal{P}(E)$ et un arbre combinatoire $A$ et qui teste si $s$ appartient à $S(A)$.*
*La complexité doit être linéaire en le nombre d'éléments de $s$.*

**Question 8**
*Écrire une fonction* `cardinal : ac -> int` *qui prend en argument un arbre combinatoire $A$ et qui renvoie $\big|S(A)\big|$, le cardinal de $S(A)$. On évaluera sa complexité.*

# 3 Principe de mémorisation

## 3.1 Taille d'un arbre combinatoire

On définit l'ensemble des sous-arbres d'un arbre combinatoire $A$, noté $\mathcal{U}(A)$, par

$$\mathcal{U}(\bot) = \{\bot\}, \quad \mathcal{U}(\top) = \{\top\}, \quad \mathcal{U}(i \to A_1, A_2) = \{i \to A_1, A_2\} \cup \mathcal{U}(A_1) \cup \mathcal{U}(A_2)$$

La taille d'un arbre combinatoire $A$, notée $T(A)$, est définie comme le cardinal de $\mathcal{U}(A)$, c'est à dire comme le nombre de ses sous-arbres **distincts**.

**Question 9**
*Quelle est la taille de l'arbre combinatoire de l'exemple* (1) *?*

## 3.2 Principe de mémorisation

Pour écrire efficacement une fonction sur les arbres combinatoires, on va mémoriser tous les résultats obtenus par cette fonction, de manière à ne pas refaire deux fois le même calcul. Pour cela, on suppose donnée une structure de table d'association indexée par des arbres combinatoires. Plus précisément, on suppose donné un type `table1` représentant une table associant à des arbres combinatoires des valeurs d'un type quelconque et les quatre fonctions suivantes :

- `cree1()` renvoie une nouvelle table, initialement vide
- `ajoute1(t,a,v)` ajoute l'association de la valeur $v$ à l'arbre $a$ dans la table $t$
- `present1(t,a)` renvoie un booléen indiquant si l'arbre $a$ est associé à une valeur dans la table $t$
- `trouve1 (t,a)` renvoie la valeur associée à l'arbre $a$ dans la table $t$, en supposant qu'elle existe

On suppose que les trois fonctions `ajoute1`, `present1` et `trouve1` ont toutes un coût constant.
On suppose de même l'existence d'un type `table2` représentant des tables d'association indexées par des couples d'arbres combinatoires et quatre fonctions similaires `cree2`, `ajoute2`, `present2` et `trouve2` également de coût constant.
*Les parties V et VI expliqueront comment de telles tables peuvent être construites.*

**Question 10**
*Réécrire la fonction* `cardinal : ac -> int` *de la question 8 à l'aide du principe de mémorisation pour garantir une complexité* $\mathcal{O}\big(T(A)\big)$.
*Justifier soigneusement la complexité du code proposé.*

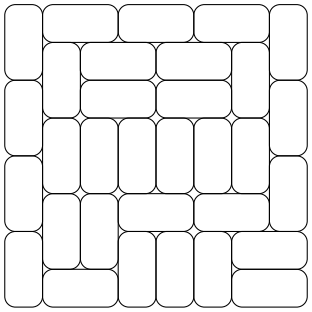
**Question 11**
*Écrire une fonction* `inter : ac -> ac -> ac` *qui prend en argument deux arbres combinatoires* $A_1$ *et* $A_2$ *et qui renvoie l'arbre combinatoire représentant leur intersection, c'est à dire l'arbre* $A$ *tel que* $S(A) = S(A_1) \cap S(A_2)$.

**Question 12**
*Montrer que, pour tous arbres combinatoires* $A_1$ *et* $A_2$*, on a*

$$T(\texttt{inter}(A_1, A_2)) \leqslant T(A_1) \times T(A_2)$$

# 4 Application au dénombrement

On en vient maintenant au problème de dénombrement évoqué dans l'introduction.
Soit $p$ un entier pair supérieur ou égal à 2. On cherche à déterminer le nombre de façons de paver un échiquier de dimension $p \times p$ avec $\frac{p^2}{2}$ dominos de taille $2 \times 1$.
Voici un exemple de tel pavage pour $p = 8$.



Pour cela, on va construire un arbre combinatoire $A$ tel que le cardinal de $S(A)$ est exactement le nombre de pavages possibles.

**Question 13**
*Combien existe-t-il de façons différentes de placer un domino* $2 \times 1$ *sur l'échiquier ?*

Dans ce qui suit, on suppose que $n$ est égal à la réponse à la question précédente, et que chaque élément $i \in E$ représente un placement possible de domino. Chaque case de l'échiquier est représentée par un entier $j$ tel que $0 \leqslant j < p^2$, les cases étant numérotées de gauche à droite, puis de haut en bas. On se donne une matrice de booléens `m` de taille $n \times p^2$. Le booléen `m.(i).(j)` vaut `true` si et seulement si la ligne $i$ correspond à un placement de domino qui occupe la case $j$.
**On suppose avoir rempli ainsi la matrice `m`, qui est une variable globale.**

Un élément $s$ de $\mathcal{P}(E)$ représente un ensemble de lignes de la matrice $m$. Il correspond à un pavage si et seulement si chaque case de l'échiquier est occupée par exactement un domino, *i.e.* si et seulement si pour toute colonne $j$, il existe une unique ligne $i \in s$ telle que `m.(i).(j)=true`.
On parle alors de **couverture exacte** de la matrice `m`.

**Question 14**
*Écrire une fonction* `colonne : int -> ac` *qui prend en argument un entier* $j$ *avec* $0 \leqslant j < p^2$*, et qui renvoie un arbre combinatoire* $A$ *tel que, pour tout* $s$*,*
$s \in S(A)$ *si et seulement si il existe un unique* $i \in s$ *tel que* `m.(i).(j) = true`*.*
*On garantira une complexité* $O(n)$*.*

**Question 15**
*En déduire une fonction* `pavage : unit -> ac` *qui renvoie un arbre combinatoire* $A$ *tel que le cardinal de* $S(A)$ *est égal au nombre de façons de paver l'échiquier, et majorer le coût de* `pavage` *en fonction de* $n$*.*

# 5 Tables de hachage

Dans cette partie, on explique comment réaliser les structures de données `table1` et `table2`, qui ont notamment permis d'obtenir des fonctions `inter` et `cardinal` efficaces. L'idée consiste à utiliser des *tables de hachage*.

On abstrait le problème en considérant qu'on cherche à construire une structure de table d'association pour des clés d'un type `clé` et des valeurs d'un type `valeur`, ces deux types étant supposés déjà définis. On se donne un entier $H > 1$ et on suppose l'existence d'une fonction `hache` de coût constant, des clés vers les entiers, telle que pour toute clé $k$, $0 \leqslant \texttt{hache}(k) < H$.
L'idée consiste alors à utiliser un tableau de taille $H$ et à stocker dans la case $i$ les entrées correspondant à des clés $k$ pour lesquelles $\texttt{hache}(k) = i$. Chaque case du tableau est appelée un *seau*. Comme plusieurs clés peuvent avoir la même valeur par la fonction `hache`, un seau est une liste d'entrées, c'est à dire une liste de couples *(clé,valeur)*. On adopte donc le type suivant :

```
type table = (clé * valeur) list array;;
```

On suppose par ailleurs qu'on peut comparer deux clés à l'aide d'une fonction `egal` à valeurs dans les booléens, également de coût constant, telle que pour toutes clés $k$ et $l$,

(1) $$\texttt{egal(k,l)} \Rightarrow \texttt{hache(k)= hache(l)}$$

**Question 16**
*Écrire une fonction* `ajoute : table -> clé -> valeur -> unit` *qui prend en argument une table de hachage $t$, une clé $k$ et une valeur $v$, et ajoute l'entrée $(k, v)$ à la table $t$. On ne cherchera pas à tester si l'entrée $(k, v)$ existe déjà dans $t$ et on garantira une complexité constante.*

**Question 17**
*Écrire une fonction* `present : table -> clé -> bool` *qui prend en argument une table de hachage $t$ et une clé $k$, et qui teste si la table $t$ contient une entrée pour la clé $k$.*

**Question 18**
*Écrire une fonction* `trouve : table -> clé -> valeur` *qui prend en argument une table de hachage $t$ et une clé $k$, et qui renvoie la valeur associée à la clé $k$ dans, en supposant qu'elle existe.*

**Question 19**
*Sous quelles hypothèses sur la valeur de $H$ et la fonction* `hache` *peut-on espérer que le coût des fonctions* `ajoute`, `present` *et* `trouve` *soit effectivement $O(1)$ ?*

# 6 Construction des arbres combinatoires

Il reste enfin à expliquer comment réaliser une fonction de hachage, une fonction d'égalité et une fonction `cons` sur les arbres combinatoires, qui soient toutes les trois de complexité $O(1)$.

L'idée consiste à associer un entier unique à chaque arbre combinatoire $A$, noté `unique(A)`, et à garantir la propriété suivante pour tous arbres combinatoires $A$ et $A$ :

$$A = B \Leftrightarrow \texttt{unique(A)= unique(A)} \tag{2}$$

Pour cela, on pose `unique(Zero)=0` et `unique(Un)=1`. Pour un arbre $A$ de la forme $i \to A_1, A_2$, on choisira pour `unique(A)` une valeur arbitraire supérieure ou égale à 2, stockée dans le nœud de l'arbre. On modifie donc ainsi la définition du type `ac` :

```
type ac = Zero | Un | Comb of unique*int * ac * ac;;
```

On propose alors la fonction `hache` suivante sur les arbres combinatoires :

- $\texttt{hache}(\bot) = 0$
- $\texttt{hache}(\top) = 1$
- $\texttt{hache}(i \to A_1, A_2) = (19^2 \times i + 19 \times \texttt{unique}(A_1) + \texttt{unique}(A_2)) \bmod H$

Le choix de cette fonction, et du coefficient 19 en particulier, relèvent de considérations pratiques uniquement.

De même, on propose la fonction `egal` suivante sur les arbres combinatoires :

- $\texttt{egal}(\bot, \bot) = \texttt{true}$
- $\texttt{egal}(\top, \top) = \texttt{true}$
- $\texttt{egal}\big((i_1 \to L_1, R_1), (i_2 \to L_2, R_2)\big) = \texttt{true}$ si $i_1 = i_2$, et $\texttt{unique}(L_1) = \texttt{unique}(L_2)$ et $\texttt{unique}(R_1) = \texttt{unique}(R_2)$
- $\texttt{egal}(A_1, A_2) = \texttt{false}$ dans les autres cas.

**Question 20**
*Montrer que les fonctions* `hache` *et* `egal` *ci-dessus vérifient bien la propriété* (1).

**Question 21**
*Proposer un code pour la fonction*

```
cons : int -> ac -> ac -> ac
```

*qui garantisse la propriété* (2)*, en supposant que les arbres combinatoires sont exclusivement construits à partir de* `Zero`*,* `Un` *et de la fonction* `cons`*. On garantira un coût en* $O(1)$ *en utilisant une table globale de type* `table1` *contenant les arbres combinatoires déjà construits. (On suppose que le type* `table1` *et ses opérations ont été adaptés au nouveau type* `ac`*.)*

Pour résoudre le problème de pavage de la partie IV, on construit au total 22518 arbres combinatoires. Si on prend $H = 19997$ et la fonction de hachage proposée ci-dessus, la longueur des seaux dans la table utilisée pour `cons` n'excède jamais 7. Plus précisément, les arbres se répartissent dans cette table de la manière suivante :

| $l$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $N(l)$ | 6450 | 7340 | 7080 | 1617 | 400 | 96 | 11 | 3 |

où $N(l)$ est le nombre de seaux de longueur $l$.

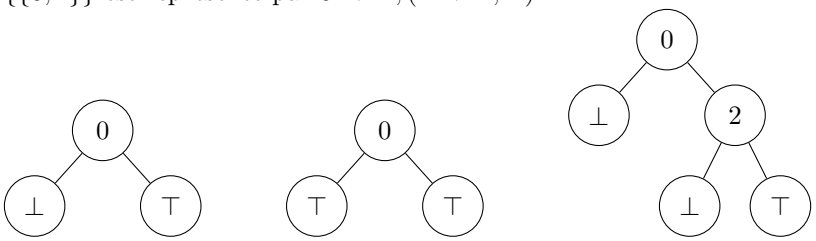
**Question 22**
*Quel est, dans cet état, le nombre moyen d'appels à la fonction* `egal` *réalisés par un nouvel appel à la fonction* `cons`

- *dans le cas où l'arbre doit être construit pour la première fois ;*
- *dans le cas où l'arbre apparaissait déjà dans la table ?*

# 7 Solutions

**Question 1 (Solution)** L'arbre combinatoire de l'exemple (1) définit $\{\{0\},\{1\},\{2\},\{0,1,2\}\}$.

**Question 2 (Solution)** $\{\{0\}\}$ est représenté par $0 \to \bot, \top$
$\{\emptyset,\{0\}\}$ est représenté par $0 \to \top, \top$
$\{\{0,2\}\}$ est représenté par $0 \to \bot, (2 \to \bot, \top)$




**Question 3 (Solution)** On montre par récurrence la propriété $\mathcal{P}(h)$ :
si la profondeur maximale d'une feuille d'un arbre combinatoire différent de $\bot$ est $h$ au plus alors l'arbre contient au moins une feuille $\top$.

- Si $h = 0$ l'arbre est une feuille qui ne peut $\bot$, c'est donc $\top$ : l'arbre contient une feuille $\top$. Ainsi $\mathcal{P}(h)$ est vérifiée.
- Si $\mathcal{P}(h)$ est vérifiée on considère un arbre de profondeur maximale $h+1$. Ce ne peut être une feuille donc il est de la forme $A = i \to A_1, A_2$.
  $A_2$ est un arbre combinatoire de profondeur maximale $h'$ avec $h' \leqslant h$ et distinct de $\bot$ par propriété de suppression. D'après l'hypothèse de récurrence, il contient une feuille $\top$. Ainsi $A$ contient une une feuille $\top$.
  $\mathcal{P}(h+1)$ est vérifiée.

**Question 4 (Solution)** On généralise la notion d'arbre combinatoire à un ensemble $F = \{a_1, a_2, \ldots, a_n\} \subset \mathbb{N}$ avec $a_1 < a_2 < \cdots < a_n$. La condition d'ordre s'écrit alors :
pour $A = a_i \to A1, A2$, ni $A_1$ ni $A_2$ ne contient d'élément $a_j$ avec $a_j \leqslant i$.
La bijection $i \mapsto a_i$ se prolonge en une bijection entre l'ensemble des arbres combinatoires sur $E$ vers l'ensemble des arbres combinatoires sur $F$.
On note $N_k$ le nombre d'arbres combinatoires sur un ensemble de cardinal $k$.
On a $N_0 = 2$, i $F$ est vide les seuls arbres combinatoires sont $\top$ et $\bot$.
Si $F$ est de cardinal $k+1$ on a $F = \{a\} \cup F'$ avec $F'$ de cardinal $k$ et $a < x$ pour tout $x \in F'$.
Les arbres combinatoires sur $F$ sont alors les arbres combinatoires sur $F'$ ou bien les arbres contenant $a$ mais alors ils doivent être de la forme $a \to A1, A_2$ avec $A_1$ et $A_2$ arbres combinatoires sur $F'$ et $A_2 \neq \bot$.
On a donc $N_{k+1} = N_k + N_k.(N_k - 1) = N_k^2$. Par récurrence on en déduit $N_k = 2^{2^k}$.

**Question 5 (Solution)** Pour trouver un élément de $S(A)$, c'est-à-dire une partie de $E$ on va prendre les éléments des racines et les ajouter aux racines des fils droits. On est ainsi certain de parvenir à un dernier fils droit égal à $\top$.

```
let rec un_elt a =
   match a with
   |Zero -> failwith ("Partie vide")
   |Un -> []
   |Comb(i,_,a) -> i::(un_elt a);;
```

L'ordre croissant de la liste est respecté car la racine a une étiquette inférieure à celles des nœuds de ses fils. La complexité est la hauteur de la descendance par la droite, elle est majorée par la hauteur de l'arbre.

**Question 6 (Solution)** On fabrique l'arbre en descendant.

```
let rec singleton liste =
   match liste with
   |[] -> Un
   |a::l -> cons  a Zero (singleton l);;
```

On n'introduit pas de feuille $\bot$ à droite donc la condition de suppression est respectée.
La liste est croissante donc les étiquettes croissent vers le bas, la condition d'ordre est respectée.
La complexité est le nombre d'éléments de la liste donc majorée par $n$.

**Question 7 (Solution)** Pour un ensemble non vide le plus petit élément doit être supérieur ou égal à l'étiquette de la racine. Si le plus petit élément est égal à la racine, le reste doit appartenir à l'ensemble défini par le fils droit, sinon l'ensemble doit appartenir au fils gauche.

```
let rec appartient l a =
   match a,l with
   |Zero, l -> false
   |Un, [] -> true
   |Un, _ -> false
   |Comb(_, a1, _), [] -> appartient [] a1
   |Comb(i, a1, a2), k::ll when k < i -> false
   |Comb(i, a1, a2), k::ll when k = i -> appartient ll a2
   |Comb(i, a1, a2), _  -> appartient l a1;;
```

Chaque appel descend la hauteur de l'arbre passé en paramètre d'au moins 1, la complexité est majorée par la hauteur de l'arbre donc par $n$.

**Question 8 (Solution)** Si $A = i \to A_1, A_2$ alors $S(A) = S(A_1) \cup \big\{\{i\} \cup s \ ; \ s \in S(A_2)\big\}$ donne $\big|\big(S(A)\big)\big| = \big|\big(S(A_1)\big)\big| + \big|\big(S(A_2)\big)\big|$ d'où la fonction

```
let rec cardinal a =
   match a with
   |Zero -> 0
   |Un -> 1
   |Comb(i,a1,a2) -> cardinal a1 + cardinal a2;;
```

La complexité est majorée par le nombre de nœuds de l'arbre, il y en a au plus $2^n$.
On peut aussi remarque que chaque appel détermine au moins un éléments des ensembles qui composent $S(A)$, on peut donc majorer la complexité par $\displaystyle\sum_{X \in S(A)} |X|$.

**Question 9 (Solution)**
L'arbre $A$ de l'exemple (1) contient les sous-arbres distincts $\big(\bot\big)$, $\big(\top\big)$, $\big(2 \to \bot, \top\big)$, $\big(1 \to (2 \to \bot, \top), \top\big)$, $\big(1 \to \top, (2 \to \bot, \top)\big)$ et $A$. Sa taille est donc 6.

**Question 10 (Solution)** On reprend la solution de l'exercice 8 en testant si un arbre combinatoire n'a pas déjà été calculé auparavant. On écrit donc une fonction auxiliaire après avoir initialisé une table.

```
let cardinal a =
   let t = cree1() in
   let rec card ac =
      if not (present1 t ac)
      then begin match ac with
           |Zero ->  ajoute1 t Zero 0
           |Un ->  ajoute1 t Un 1
           |Comb(_, a1, a2) -> ajoute1 t ac (card a1 + card a2
              ) end;
      trouve1 t ac in
   card a;;
```

On ne fait un appel récursif à `card` que pour les fils des arbres non encore calculés. Le nombre d'appels récursifs est donc majoré par $2T(A)$, la complexité est bien linéaire en $T(A)$.

**Question 11 (Solution)** L'arbre représentant l'intersection de $A_1$ et $A_2$ se calcule selon les valeurs des arbres.

- Si $A_1 = \texttt{Zero}$ ou $A_2 = \texttt{Zero}$ alors $S(A_1) \cap S(A_2) = \emptyset$ et $\texttt{inter}(A_1, A_2) = \texttt{Zero}$.
- Si $A_1 = \texttt{Un}$ alors $S(A_1) = \{\emptyset\}$ et l'intersection est égale à $\{\emptyset\}$ ou $\emptyset$ selon que $\{\emptyset\}$ appartient ou non à $S(A_2)$ ce qui se voit dans le fils gauche de $A_2$ car les éléments provenant de $A_2$ contiennent $i$.
- Si $A = i \to A_1, A_2$ et $B = j \to B_1, B_2$ avec $i < j$ alors les éléments de $S(A) \cap S(B)$ ne contiennent pas $i$ car aucun élément de $S(B)$ ne contient $i$ en raison de la condition d'ordre. On a donc $\texttt{inter}(A, B) = \texttt{inter}(A_1, B)$
- Si $A = i \to A_1, A_2$ et $B = i \to B_1, B_2$ alors les éléments de $S(A) \cap S(B)$ qui ne contiennent pas $i$ sont ceux de $S(A_1) \cap S(B_1)$ et ceux qui contiennent $i$ sont associés aux éléments de $S(A_2) \cap S(B_2)$. Ainsi $\texttt{inter}(A, B) = i \to \texttt{inter}(A_1, B_1), \texttt{inter}(A_2, B_2)$. En raison de la propriété de suppression il faudra tester si $\texttt{inter}(A_2, B_2) = \perp$ car ce ne peut être un fils droit.

```
let rec inter a b =
   match a, b with
   |Zero, _ -> Zero
   |_, Zero -> Zero
   |Un, Un ->  Un
   |Un, Comb(_, ba, _) -> inter Un ba
   |Comb(_, aa, _), Un -> inter Un aa
   |Comb(i, aa, ab), Comb(j, ba, bb) when i < j -> inter aa b
   |Comb(i, aa, ab), Comb(j, ba, bb) when j < i -> inter a ba
   |Comb(i, aa, ab), Comb(j, ba, bb) -> let c = inter ab bb in
                                        if c = Zero
                                        then inter aa ba
                                        else cons i (inter aa
                                           ba) c;;
```

Bien que cela ne soit pas indiqué, il semble indispensable de ne pas calculer plusieurs fois les intersections déjà calculées. On utilisera une méthode semblable à la précédente.

```
let inter a b =
   let t = cree2() in
   let rec intct a b =
      if not (present2 t (a, b))
      then begin
         match a, b with
         |Zero, _ -> ajoute2 t (a, b) Zero
         |_, Zero -> ajoute2 t (a, b) Zero
         |Un, Un ->  ajoute2 t (a, b) Un
         |Un, Comb(_, ba, _) -> ajoute2 t (a, b) (inter Un ba)
         |Comb(_, aa, _), Un -> ajoute2 t (a, b) (inter Un aa)
         |Comb(i, aa, ab), Comb(j, ba, bb) when i < j
            -> ajoute2 t (a, b) (inter aa b)
         |Comb(i, aa, ab), Comb(j, ba, bb) when j < i
            -> ajoute2 t (a, b) (inter a ba)
         |Comb(i, aa, ab), Comb(j, ba, bb)
            -> let c = inter ab bb in
               if c = Zero
               then ajoute2 t (a, b) (inter aa ba)
               else ajoute2 t (a, b) (cons i (inter aa ba) c)
                  end;
      trouve2 t (a,b)
   in intct a b;;
```

**Question 12 (Solution)** Dans la question ci-dessus $T(\texttt{inter}(A_1, A_2))$ est le nombre de cases définies dans le tableau `t`. Or chacun des appels qui définit une case se fait à partir d'un couple de sous-arbres donc le nombre d'appel est majoré par le nombre se sous-arbres de $A_1$ multiplié par le nombre se sous-arbres de $A_2$ : $T(\texttt{inter}(A_1, A_2)) \leqslant T(A_1) \times T(A_2)$.

**Question 13 (Solution)** Pour placer un domino à l'horizontale il suffit de placer la partie gauche : on choisit une des $p$ lignes et l'une des $p-1$ premières colonnes. On trouve donc $p(p-1)$ positions. Il y en a, par symétrie, autant pour le placer à la verticale.
Il existe $2p(p-1)$ façons différentes de placer un domino $2 \times 1$ sur un échiquier.

**Question 14 (Solution)** On note $\mathbb{E}^1$ l'ensemble considéré, ce sont les parties $E \subset \{0, 1, 2, \ldots, n-1\}$ telles qu'il existe un unique $i \in E$ avec `m.(i).(j) = true`.
On note $A = 0 \to A_1, A_2$, l'arbre tel que $S(A) = \mathbb{E}^1$.

- Si `m.(0).(j) = false`, $A_1$ et $A_2$ sont égaux et ils représentent l'ensemble des parties $E \subset \{1, 2, \ldots, n-1\}$ telles que `m.(i).(j) = false` pour tout $i \in E$ sauf un.
- Si `m.(0).(j) = true`, $A_1$ est la partie décrite ci-dessus et $A_2$ représente l'ensemble des parties $E \subset \{1, 2, \ldots, n-1\}$ telles que `m.(i).(j) = false` pour tout $i \in E$.

On introduit les notations :

- $\mathbb{E}_k^0$ l'ensemble des parties $E \subset \{k, k+1, \ldots, n-1\}$ telles que `m.(i).(j) = false` pour tout $i \in E$.
- $A_k^0$ est l'arbre combinatoire qui représente $\mathbb{E}_k^0$.
- $\mathbb{E}_k^1$ l'ensemble des parties $E \subset \{k, k+1, \ldots, n-1\}$ telles que `m.(i).(j) = false` pour tout $i \in E$ sauf un.
- $A_k^1$ est l'arbre combinatoire qui représente $\mathbb{E}_k^1$, on cherche $A_0^1$.

On vient de voir qu'on a $A_0^1 = 0 \to A_1^1, A_1^0$ ou $A_0^1 = 0 \to A_1^1, A_1^1$ selon que `m.(0).(j) = true` ou `m.(0).(j) = false`.
On peut généraliser : si `m.(k).(j) = true`, $A_k^1 = k \to A_{k+1}^1, A_{k+1}^0$, $A_k^0$ n'est jamais égal à $\perp$ car $\mathbb{E}_k^0$ contient toujours $\emptyset$ ; si `m.(k).(j) = false`, $A_k^1 = k \to A_{k+1}^1, A_{k+1}^1$,
sauf lorsque $A_{k+1}^1 = \perp$, dans ce cas $A_k^1 = \perp$, on a alors $\mathbb{E}_k^1 = \emptyset$.

On peut construire de même les $A_k^0$ : si `m.(k).(j) = true`, $A_k^0 = A_{k+1}^0$,
si `m.(k).(j) = false` , $A_k^0 = k \to A_{k+1}^0, A_{k+1}^0$.
On a ainsi un algorithme de construction des $(A_k^0, A_k^1)$ à partir de $(A_n^0, A_n^1) = (\top, \bot)$.

```
let colonne j =
   let rec col k =
    if k = n
    then Zero, Un
    else let a0, a1 = col (k+1) in
         if m.(k).(j)
         then a0, (cons k a1 a0)
         else if a1 = Zero
              then  cons k a0 a0, Zero
              else  cons k a0 a0, cons k a1 a1  in
 col 0;;
```

L'appel récursif, qui n'ajoute que des calculs à temps constant, se fait pour chaque entier de 0 à $n-1$. La complexité est donc un $\mathcal{O}(n)$.
On notera ensuite que la taille de `colonne j` est au plus $2n$, en effet à chaque étape on ajoute au plus deux sous-arbres dans le résultat.

**Question 15 (Solution)** Les pavages correspondent à un ensemble des positions de dominos $I$ tel que, pour toute case $j$ du damier il existe une unique position $i \in I$ telle que `m.(i).(j0)=true`.
On doit donc avoir $I \in S(A_j)$ pour tout $j$ avec $A_j =$`colonne j`.
On calcule donc l'intersection des $A_j$.

```
let pavage () =
   let dim = Array.length m.(0);
   let rec pave k a =
      if k < 0 then a
               else inter a (colonne k) in
   pave (dim-2) (colonne (dim-1));;
```

La valeur de `dim` est $p^2$ ; on fait donc $p^2$ fois appel à `colonne` ce qui donne une complexité de $np^2 = \%\!\!\%p^4$.
On fait $p^2$ fois appel à `inter` ; si on note $B_j$ les différentes intersections calculées la complexité d'un calcul est de l'ordre de $T(\texttt{inter}(A_j, B_j)) \leqslant T(A_j) \times T(B_j) \leqslant 2nT(B_j)$. On a ainsi $T(B_{j+1}) \leqslant 2nT(B_j)$ donc $T(B_j) \leqslant (2n)^j$ donc la complexité des appels à `inter` est majorée par

$$\sum_{j=0}^{p^2-1} (2n)^{j+1} = \frac{(2n)^{p^2+1} - 1}{2n-1} + 1 = \mathcal{O}\big((2n)^{p^2}\big) = \mathcal{O}\big((4p^2)^{p^2}\big) = \mathcal{O}\big((2p)^{2p^2}\big)$$

Ce terme est un infiniment grand par rapport au premier : la complexité totale est en $\mathcal{O}\big((2p)^{2p^2}\big)$.

**Question 16 (Solution)**

```
let ajoute t k v =
   let c= hache k in
   t.(c) <- (k,v)::t.(c);;
```

**Question 17 (Solution)**

```
let present t k =
   let rec pres liste =
      match liste with
      |[] -> false
      |(k1,v)::ll -> egal k k1 || pres ll in
   pres t.(hache k);;
```

**Question 18 (Solution)**

```
let trouve t k =
   let rec trouv liste =
      |[] -> failwith "Clé non présente"
     |(k1,v)::ll -> if egal k k1
                    then v
                    else trouv  ll in
   trouv t.(hache k);;
```

**Question 19 (Solution)** `ajoute` est à temps constant car `hache` l'est.
`present` et `trouve` explorent une liste les autres opérations étant à temps constant, pour que la complexité soit un ‰1 il faut donc que les seaux aient une taille limitée par un entier $K$. Pour cela il faut que $KH$ soit supérieur au nombre d'objets à classer et que la fonction de hachage répartisse régulièrement les objets dans les seaux.

**Question 20 (Solution)** `hache` est calculée, dans le cas d'un arbre distinct de `Un` et `Zero`, par les valeur de $i$, $A_1$ et $A_2$ donc le cas d'égalité par `egal` implique l'égalité des clé de hachage.
On ne teste pas l'égalité par la valeur de `unique` de l'arbre car dans la suite on calcule celle-ci après avoir calculé la valeur de `hache`.

**Question 21 (Solution)** L'idée est de donner une valeur provisoire à `unique` de voir si l'arbre a déjà été construit (d'où l'intérêt de ne pas utiliser cette valeur) et de changer la valeur de `unique` si elle existait déjà.
On commence par les initialisations :

```
let t_unique = Array.make h [];;
ajoute t_unique Zero 0;;
ajoute t_unique Un 1;;
let compteur = ref 2;;
```

On travaille ensuite avec ces objets globaux :

```
let cons i a b =
   if present t_unique Comb(0, i, a, b)
   then let uni = trouve t_unique Comb(0, i, a, b) in
        comb(uni, i, a, b)
   else begin
        let uni = !compteur in
        incr compteur;
        ajoute t-unique Comb(uni, i, a, b) uni;
        Comb(uni, i, a, b) end;;
```

**Question 22 (Solution)** On note $n_k$ le nombre de seaux de longueur $k$.

- Si l'arbre est construit pour la première fois on n'utilise `egal` que dans l'instruction `present` et on doit parcourir la liste associée en entier parce qu'on ne le trouve pas. Dans la situation où est le tableau la longueur moyenne d'une liste est $\frac{\sum_{k=0}^{7} k n_k}{\sum_{k=0}^{7} n_k} \sim 1,13$.
- Si l'arbre apparaissait déjà dans la table, `egal` est appelée deux fois, dans `present` et dans `trouve`. Pour un seau donné de longueur $k$, les $k$ arbres demandent $\sum_{i=1}^{k} i$ recherches.

  Ainsi la moyenne du nombre d'appels à égal est à $\frac{\sum_{k=1}^{7} \left(n_k \sum_{i=1}^{k} i\right)}{\sum_{k=1}^{7} k n_k} \sim 3,11$.