Mpsi 1 et 2 - option informatique **Devoir surveillé 2**

Les algorithmes seront écrits en CAML. Ils doivent être suffisamment commentés pour faciliter la tâche du correcteur. Toute affirmation de votre part doit être justifiée.

Même s'il vous est demandé le code d'une fonction, il n'est pas interdit de créer d'autres fonctions auxiliaires.

Problème 1 - Permutations

Pour $n \in \mathbb{N}^*$, on appelle permutation de longueur n toute suite de longueur n qui contient exactement les éléments 0, 1, ..., $n-1$, dans un ordre quelconque.

On représente une permutation de longueur n par une liste d'entiers de longueur n.

Partie 1 - Reconnaître une permutation

Question 1)

a) Écrivez une fonction `app x l` de type `'a -> 'a list -> bool`, qui vérifie l'appartenance d'un objet à une liste. Quelle est sa complexité en fonction de la longueur de la liste ?

b) Écrivez une fonction `incluse l1 l2` de type `'a list -> 'a list -> bool`, qui vérifie si tous les éléments de `l1` sont aussi éléments de `l2`. Si on note n_1, n_2 les longueurs des deux listes, quelle est la complexité de cet algorithme en fonction de n_1 et n_2 ?

Question 2) Donnez une fonction qui vérifie qu'une liste d'entiers est effectivement une permutation. Quelle est sa complexité ?

Question 3) En fait, dans le cas des permutations de longueur n, comme on connaît à l'avance les objets qu'on doit trouver dans la liste, on peut faire mieux.

Soi l une liste de longueur n.
- au départ, on construit un tableau de n booléens rempli uniquement de valeurs $false$; les cases du tableau sont numérotées de 0 à $n-1$;
- on parcourt la liste d'entiers : à chaque étape, on lit un élément i de la liste et on place dans la case numéro i du tableau la valeur $true$, sauf si i n'est pas compris ... ou si cette case contient ...

Zut ! Une partie de l'algorithme a été effacée !

Pourriez-vous reconstituer l'énoncé et donner ainsi un algorithme (décrit en français courant) plus rapide que le précédent qui vérifie si la liste est une permutation ? Quelle est sa complexité ?

Question 4) Traduisez l'algorithme précédent en une fonction CAML. Pour cela vous pourrez vous servir des notations suivantes :
- si `t` est un tableau, le contenu de la case numéro `i` est noté `t.(i)`
- pour modifier la valeur d'une case, on utilise la notation `t.(i) <- x`
- pour créer un tableau de longueur `n` dont toutes les cases sont initialisées à la valeur `x`, on utilise la fonction `make_vect n x`

Partie 2 - Construire une permutation

On suppose disposer d'une fonction `hasard n` de type `int -> int`, qui donne un entier choisi aléatoirement entre 0 et $n-1$: un appel à cette fonction est supposé être une opération élémentaire. On veut créer une fonction qui construit aléatoirement une permutation de longueur n.

Question 1) Une première idée est de partir d'une liste vide L. Pour i variant de 0 à $n-1$, on tire au hasard un entier entre 0 et $n-1$ et on vérifie s'il fait partie des valeurs de la liste L : si c'est le cas, on réitère ce tirage et cette comparaison, sinon on place l'entier ainsi choisi en tête de L.

Expliquez pourquoi cette idée n'est pas efficace en donnant la complexité minimale d'un tel algorithme. La complexité moyenne est encore plus élevée...

Question 2) Une deuxième idée est récursive. Si on sait construire aléatoirement une permutation ℓ de longueur $n-1$, alors on choisit au hasard une position dans la liste ℓ et on y place n : la liste obtenue est une permutation de longueur n.

Écrivez une ou plusieurs fonctions qui mettent en œuvre cette idée. Quelle est la complexité d'une telle idée ?

Question 3) Une dernière idée.

On peut mélanger au hasard deux listes t et u en une seule :
- si les deux listes t et u ne sont pas vides, on tire au hasard un entier 0 ou 1 : si c'est 0, on place en tête la tête de t et on la retire de t, sinon on place la tête de u et on la retire de u, puis on répète cette opération ;
- si l'une des deux listes t ou u est vide, on se contente de retourner t ou la liste u

a) Écrivez le code de cette fonction de mélange.

b) Soit n un entier au moins égal à 2. On suppose connaître deux permutations p et q de longueur $n/2$ et $m = n - (n/2)$. Quelle transformation simple suffit-il de faire subir à la seconde permutation q, ce qui donne une liste q', pour que le mélange de p et q' selon le principe précédent donne une permutation de longueur n ?

c) Explicitez le principe d'un algorithme plus efficace que celui de la question 2, qui calcule une permutation de longueur n. Quelle complexité peut-on obtenir d'un tel algorithme ?

d) Écrivez en code CAML une ou plusieurs fonctions nécessaires à sa traduction.

Problème 2 - Partitions d'un entier

Dans tout le problème, k et n désignent deux entiers naturels, k non nul. On appelle k-partition de n toute suite d'entiers $(a_1, \ldots, a_k)$ telle que :
- $n = \sum_{i=1}^{k} a_i$
- $1 \leqslant a_1 \leqslant a_2 \leqslant \ldots \leqslant a_k$.

On note $P(n,k)$ l'ensemble des k-partitions de n. Par exemple, $P(6,3) = \{(1,1,4);(1,2,3);(2,2,2)\}$.

Question 1) Que vaut $P(n,k)$ si $k > n$? Que valent $P(n,1)$, $P(n,n)$? On suppose désormais que $k \leqslant n$.

Question 2) Pour $k \geqslant 2$, justifiez qu'il existe une bijection entre l'ensemble des $(k-1)$-partitions de $n-1$ avec celui des k-partitions de n dont le premier terme est 1.

Question 3) À toute k-partition de n dont le premier terme n'est pas un 1, $p = (a_1, \ldots, a_k)$ avec $a_1 \geqslant 2$, on associe le k-uplet $(a_1 - 1, \ldots, a_k - 1)$: montrez que cette application est une bijection de l'ensemble des k-partitions de n dont le premier terme n'est pas un 1 dans un autre ensemble de k-partitions.

Question 4) Expliquez comment construire l'ensemble $P(n,k)$ à partir des ensembles $P(n-1,k-1)$ et $P(n-k,k)$.

Question 5) On représente une k-partition de n par une liste à k éléments entiers ordonnés dans l'ordre croissant et un ensemble de partitions par une liste de telles listes. Par exemple, l'ensemble $P(6,3)$ est représenté par la liste `[[1; 1; 4]; [1; 2; 3]; [2; 2; 2]]`.

a) Écrivez une fonction `incrementer li` de type `int list -> int list`, qui construit à partir de la liste $[a_0; \ldots; a_{n-1}]$ la liste $[a_0 + 1; \ldots; a_{n-1} + 1]$, puis une fonction `incrementer_liste lli` de type `int list list -> int list list`, qui applique la fonction précédente à chaque liste d'une liste de listes.

b) Écrivez une fonction `partitions n k` de type `int -> int -> int list list` qui calcule la représentation de $P(n,k)$ (vous pouvez écrire diverses fonctions auxiliaires si vous en avez besoin).

Question 6) Écrivez une fonction `partitionner n` de type `int -> int list list` qui calcule la liste de toutes les partitions possibles de n.

Devoir surveillé 2 - Corrigé

Problème 1

Partie 1

Question 1)

a)
```
let rec app x l =
  match l with
  | [] -> false
  | a ::  q -> (x = a) || app x q;;
```

Si n est la longueur de la liste paramètre et $C(n)$ la complexité de cette fonction, alors on a la relation de récurrence $C(n) = C(n-1) + O(1)$, car les opérations effectuées lors de chaque appel sont élémentaires. Donc on en déduit que $C(n) = O(n)$: la fonction est de complexité linéaire.

b) On parcourt une liste : à chaque étape, on vérifie si l'élément en cours appartient à l'autre liste. Si tout se passe bien, on prouve ainsi que la première liste est incluse en tant qu'ensemble dans la seconde.

Comme vérifier qu'un objet appartient à une liste a une complexité linéaire en la longueur de la liste, on en déduit que vérifier que tous les entiers d'une liste de longueur n_1 sont dans une liste de longueur n_2 a une complexité $C(n_1, n_2) = O(n_1 n_2)$, car la relation de récurrence est $C(n_1, n_2) = C(n_1 - 1, n_2) + O(n_2)$

```
let rec incluse l1 l2 =
  match l1 with
  | [] -> true
  | x :: q1 -> app x l2 && incluse q1 l2;;
```

Question 2)

```
let rec liste n =
  match n with
  | 0 -> []
  | _ -> (n-1) :: liste (n-1);;

let verifie l =
  let n = list_length l in
  let m = liste n in
  incluse m l;;
```

On calcule la longueur n de la liste, on construit la liste des entiers de 0 à $n-1$, puis on vérifie que cette liste est incluse dans la première : comme elles ont le même cardinal, si l'une est incluse dans l'autre, alors elles sont égales.

D'après ce qui précède, la complexité est en $O(n^2)$.

Question 3) On initialise un tableau de booléens de longueur n, contenant des $false$ initialement. Puis on parcourt la liste : on note i l'élément en cours d'analyse ; si on trouve un entier en dehors de $\{0, \ldots, n-1\}$, alors on s'arrête car la liste n'est pas une permutation ; sinon on regarde le contenu de la case i du tableau, si elle est égale à "vrai" alors on s'arrête car on a déjà trouvé l'entier avant, sinon on remplace la case i par "vrai" et on passe à l'élément suivant de la liste.

Soit on s'est arrêté sur un cas "faux" précédent, soit on a parcouru toute la liste et dans ce cas, c'est bien une permutation.

Comme on ne parcourt la liste d'une fois et que chaque étape est de complexité constante, la complexité est cette fois-ci en $O(n)$.

Question 4)

```
let verifie l =
  let n = list_length l in
  let t = make_vect n false in
  let rec verifaux l =
    match l with
    | [] -> true
    | i :: q -> (0 <= i) && (i <= n-1)
             && (if not t.(i) then
                  begin
                    t.(i) <- true;
                    verifaux q
                  end
                  else false)
  in
  verifaux l;;
```

Partie 2

Question 1) Pour tester si un entier a déjà été tiré au sort, il faut parcourir toute la liste, soit i valeurs à tester, et ceci doit être répété pour i variant de 0 à $n-1$, ce qui donne $1+2+...+(n-1) = \frac{n(n-1)}{2}$ tests au minimum. Et on ne compte pas les éventuels retirages, ce qui peut aléatoirement augmenter la complexité. La complexité minimale est donc au moins quadratique.

Question 2)

```
let rec inserer x l k =
  if k = 0 then x :: l
  else match l with
       | [] -> [x]
       | a :: q -> a :: inserer x q (k-1);;

let rec perm n =
  if n = 0 then []
  else let p = perm (n-1) in
    let h = hasard n in
    inserer n p h;;
```

La fonction `inserer x l k` insère l'élément `x` en position `k` dans la liste `l` dans le cas où `k` est inférieur ou égal à la longueur de `l` ou en dernière position sinon.

La fonction `perm` traduit l'idée de l'énoncé.

La fonction d'insertion est de complexité linéaire en la longueur de la liste (on parcourt une fois la liste et les opérations effectuées sotn élémentaires) et la complexité $C(n)$ de la deuxième fonction vérifie la relation $C(n) = C(n-1) + O(n)$ donc $C(n) = O(n^2)$.

Question 3)

a)

```
(* melanger deux listes *)
let rec melanger t u =
  match t, u with
  | [],[] -> []
  | [],_ -> u
  | _, [] -> t
  | a :: t', b :: u' ->
      let h = hasard 2 in
      if h = 0 then a :: melanger t' u
      else b :: melanger t u';;
```

b) Il suffit d'ajouter $n/2$ à chaque élément de q : en effet, q contient les entiers $0, 1, \ldots, m-1$, donc q' est alors une liste contenant les entiers $n/2, n/2+1, \ldots, m+n/2-1$. Or $m+n/2-1 = n-1$, donc si on mélange q' aléatoirement avec la liste p qui contient les entiers $0, \ldots, n/2-1$, alors on obtient une liste à n éléments, qui contient les entiers $0, \ldots, n-1$, donc c'est une permutation de longueur n.

c) L'ajout d'un entier à tous les éléments d'une liste est de complexité linéaire en la longueur de la liste. Le mélange précédent est de complexité linéaire en la somme des longueurs des listes.

Donc si on applique le principe précédent, on se retrouve avec un algorithme utilisant la méthode « diviser pour régner », dont la complexité vérifiera la relation de récurrence $C(n) = 2C(n/2)+O(n)$, donc d'après le th. du cours, la complexité est en $O(n \log n)$:

— l'étape de division est simplement diviser n par 2 : coût $O(1)$
— le calcul des deux permutations est l'étape de règne ;
— la fusion consiste à ajouter $n/2$ à la deuxième permutation de longueur m (coût $O(m) = O(n)$) et à mélanger les deux listes (coût $O(n)$, car la somme des longueurs des listes est égale à n).

d)

```
(* ajouter un entier a chaque entier d'une liste *)
let rec ajoute p l =
  match l with
  | [] -> []
  | a :: q -> (a + p) :: ajoute p q;;

let rec perm2 n =
  if n = 1 then [0]
  else
  let m = n / 2 in
  let t = perm2 m and u = ajoute m (perm2 (n - m)) in
  fusion t u;;
```

Problème 2

Question 1) Si $p = (a_1, \ldots, a_k)$ est une k-partition de n, alors $k \leqslant a_1 + \ldots + a_k = n$, donc $P(n,k) = \varnothing$ si $k > n$.

Si $k = 1$, alors la seule 1-partition de n est la suite (n), donc $P(n,1) = \{(n)\}$. Si $k = n$, alors la seule n-partition de n est la suite de longueur n constante égale à 1 : $P(n,n) = \{(1, \ldots, 1)\}$.

Question 2) A toute k-partition de n commençant par un 1, $p = (1, a_2, \ldots, a_k)$, on associe la $(k-1)$-partition de $n-1$ $(a_2, \ldots, a_k)$: en effet, la liste $(a_2, \ldots, a_k)$ est bien une liste d'entiers naturels non nuls ordonnée en sens croissant et $\displaystyle\sum_{i=2}^{k} a_i = \left(\sum_{i=1}^{k} a_i\right) - a_1 = n-1$.

On définit donc une application de $P'(n,k)$, ensemble des k-partition de n commençant par un 1, dans $P(n-1, k-1)$, en associant à $p = (1, a_2, \ldots, a_k)$, la partition $F(p) = (a_2, \ldots, a_k)$.

F est une bijection de $P'(n,k)$ dans $P(n-1,k-1)$, car elle a pour réciproque l'application $(a_2, \ldots, a_k) \mapsto (1, a_2, \ldots, a_k)$.

Question 3) A toute k-partition de n ne commençant pas par un 1 (donc au moins par un 2), $p = (a_1, \ldots, a_k)$ avec $a_1 \geqslant 2$, on associe la k-partition de $n-k$ $(a_1 - 1, \ldots, a_k - 1)$: en effet, la liste $(a_1 - 1, \ldots, a_k - 1)$ est une liste d'entiers naturels non nuls ordonnée en sens croissant et $\displaystyle\sum_{i=1}^{k} (a_i - 1) = \left(\sum_{i=1}^{k} a_i\right) - k = n - k$.

On définit donc une application de $P''(n,k)$, ensemble des k-partition de n commençant par un $a_1 \geqslant 2$, dans $P(n-k, k)$, en associant à $p = (a_1, a_2, \ldots, a_k)$, la partition $G(p) = (a_1 - 1, \ldots, a_k - 1)$.

G est une bijection de $P''(n,k)$ dans $P(n-k,k)$, car elle a pour réciproque l'application $(a_1, \ldots, a_k) \mapsto (a_1 + 1, a_2 + 1, \ldots, a_k + 1)$.

Question 4) L'ensemble $P(n,k)$ est la réunion disjointe des deux ensembles précédents $P'(n,k)$ et $P''(n,k)$. Le premier est obtenu à partir de $P(n-1,k-1)$ en ajoutant à chaque $k-1$-partition de $n-1$ le nombre 1 en

tête. Le second est obtenu à partir de $P(n-k,k)$ en additionnant 1 à chaque élément de chaque k-partition de $n-k$.

En termes d'applications, on a l'égalité $P(n,k) = F^{-1}\big(P(n-1,k-1)\big) \cup G^{-1}\big(P(n-k,k)\big)$.

Ceci donne une définition récursive de $P(n,k)$, car dans les deux cas, la somme $n+k$ décroît strictement dans $\mathbb{N}$, ensemble bien fondé. Les cas de base sont les cas où $k=1 : P(n,1)$ est connu, et où $k > n$ (idem).

Question 5)

a) On crée donc ici la fonction G^{-1}.

```
let rec incrementer li =
  match li with
  | [] -> []
  | a :: q -> (a+1) :: incrementer q;;

let rec incrementer_liste lli =
  match lli with
  | [] -> []
  | li :: qli -> incrementer li :: incrementer_liste qli;;
```

b) On crée une autre fonction qui sert à ajouter un 1 en tête de chaque liste d'une liste de listes (c'est la fonction F^{-1}). Puis on applique le principe récursif détaillé ci-dessus.

```
let rec ajouter_un lli =
  match lli with
  | [] -> []
  | li :: qli -> (1 :: li) :: ajouter_un qli;;

let rec partitions n k =
  if k = 1 then [[n]]
  else if n < k then []
  else let p = partitions (n-1) (k-1) in
       let q = partitions (n-k) k in
       ajouter_un p @ incrementer_liste q;;
```

Question 6) L'idée est plutôt itérative : on part de l'ensemble vide et pour k variant de 1 à n, on réunit progressivement cet ensemble avec les ensembles $P(n,k)$. Pour traduire cette idée récursivement, on introduit donc une fonction récursive terminale encapsulée, avec un paramètre accumulateur.

```
let partitionner n =
  let rec partaux k p =
  if k = 0 then p
  else let lli = partitions n k in
       partaux (k-1) (lli @ p)
  in
  partaux n [];;
```