DS2 : Types et Structure de Donnée

Durée : 4 heures

1er décembre 2023

Vous pouvez réutiliser des résultats des questions précédentes pour une démonstration ou un code, même si la question n'a pas été traitée.

Vous pouvez introduire des fonctions, variables, types ou notations supplémentaires pour produire vos réponses.

Si, au cours de l'épreuve, vous repérez ce qui vous semble être une erreur d'énoncé, signalez-le sur votre copie et poursuivez votre composition en expliquant les raisons des initiatives que vous seriez amené·e à prendre.

Pour la pagination, vous numéroterez toutes les pages de votre composition en faisant figurer le numéro de la page actuelle à partir de 1, ainsi que le total de pages de votre composition. Vous ferez la pagination durant le temps de l'épreuve, et non après la fin de l'épreuve.

Votre nom devra figurer sur chacune de vos copies qui seront rendue imbriquées les unes dans les autres.

Les questions difficiles sont indiquées par une ou plusieurs étoiles (⋆). Cette difficultés est principalement indicative.

Prenez bien soin de lire les consignes et les indications fournies dans le sujet.

Bon courage, et bonne composition.

— La syntaxe fun x -> ... permet de définir une fonction anonyme.
— On peut définir une fonction de signature unit -> ... grâce à la syntaxe let f () = ... Le premier argument de cette fonction doit toujours être () et le code du corps de la fonction n'est exécuté que si f () est appelé.
— On peut définir un type construit somme avec la syntaxe :

```
type nom_du_type = Constructeur1 | Constructeur2 | ...
```

— On peut rajouter dans certains de ces constructeurs un type avec la syntaxe :

```
type nom_du_type = Constructeur1 of type1 | ...
```

— Pour créer une valeur de ces types, on doit utiliser exactement un constructeur, soit le constructeur lui-même Constructeur1, ou bien s'il est issu d'un type, Constructeur1 valeur1.
— On peut définir un type enregistrement avec la syntaxe :

```
type nom_du_type = {champs1: type1 ; champs2 : type2 ; ....}
```

— Certains de ces champs peuvent être mutables :

```
type nom_du_type = {mutable champs1: type1 ; ....}
```

— Pour créer une valeur de ce type, on utilise la syntaxe :

```
{champs1 = valeur1; champs2 = valeur2; ....}
```

— Le type construit 'a option peut être défini par type 'a option = None | Some of 'a.

Ce sujet est composé de deux problèmes indépendants. Le premier traite de permutations constructibles avec une pile, et le second traite de différentes stratégies d'évaluation pour optimiser certains appels.

Problème : Pile et permutations de pile

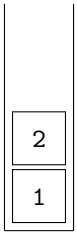
Dans ce problème on suppose qu'on dispose des nombres de 1 à n accessibles dans l'ordre ainsi que d'une pile, et la question est de savoir quelles sont les permutations de $\{1, 2, ..., n\}$ on peut obtenir.

Partie I : Dénombrement

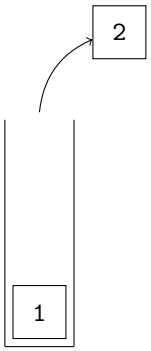
On définie une *permutation* σ de $\{1, 2, ..., n\}$ (l'ensemble des entiers de 1 à n) comme un n-uplet $(\sigma_1, \sigma_2, ..., \sigma_n)$ de sorte à ce que :
— Pour tout i, on a $\sigma_i \in \{1, 2, ..., n\}$
— pour tout $i \neq j$, on a $\sigma_i \neq \sigma_j$.
Ainsi, une permutation $\{1, 2, ..., n\}$ est l'ensemble des manières d'ordonner les entiers de 1 à n. Par exemple $(1, 3, 2, 4)$ est une permutation, mais $(1, 2, 2, 4)$ n'en est pas une.

On s'intéresse aux permutations qu'on peut obtenir avec les entiers tirés dans l'ordre et une pile. On doit prendre les entiers *par ordre croissant* et à chaque étape, on peut soit prendre le plus petit entier parmi les entiers restant et l'ajouter à la pile, soit retirer l'élément au sommet de la pile pour l'ajouter à la permutation, nous donnant successivement les valeurs de σ_1, puis σ_2, et ainsi de suite jusqu'à σ_n.
On rajoute la contrainte par ailleurs que la pile doit être vide à la fin de ces opérations.

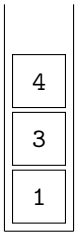
Par exemple, on peut construire la permutation $(2, 4, 3, 1)$ avec une pile.
En effet, on commence par ajouter les deux premiers entiers, c'est-à-dire 1 puis 2, à la pile pour obtenir :



On retire ensuite 2, et on a pour l'instant 2 en sortie ce qui nous donne pour l'instant la permutation $(2, .., .., ..)$.



On rajoute ensuite les deux entiers suivants, 3 puis 4, pour obtenir :



Il suffit ensuite de retirer 4, puis 3, puis 1 pour obtenir la permutation $(2, 4, 3, 1)$.

Question 1. Montrer qu'il est possible d'obtenir la permutation $(3, 2, 4, 1)$.

Question 2. Montrer qu'il n'est pas possible d'obtenir la permutation $(3, 1, 2, 4)$.

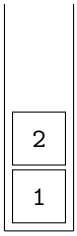
Question 3. Montrer que, pour tout n, il est possible d'obtenir la permutation $(1, 2, 3, ..., n)$ des entiers croissants de 1 à n.

Question 4. Montrer que, pour tout n, il est possible d'obtenir la permutation $(n, n-1, n-2, ..., 1)$ des entiers décroissants de n à 1.

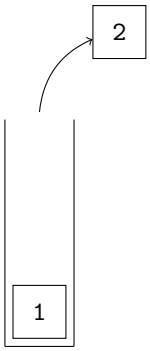
On décide de représenter une suite d'instructions à réaliser avec la pile par une séquence de A et de R. Les A correspondent au fait d'ajouter l'élément suivant dans la pile, et les R correspondent au fait de retirer un élément de la pile (et l'ajouter à la permutation). On appelle une telle séquence *recette*.

À partir d'une recette, on peut construire une permutation en suivant les instructions données par la recette.

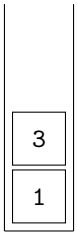
Ainsi, lorsqu'on suit la recette $AARARR$, on commence par ajouter les deux premiers entiers, 1 puis 2, dans la pile pour obtenir la pile suivante :



Puis on retire un élément qui nous permet de commencer notre permutation $(2, .., ..)$:



Puis on ajoute le prochain élément, 3, ce qui nous donne :



Enfin, on retire les deux éléments de la pile (3, puis 1), et on obtient la permutation $(2, 3, 1)$.

Réciproquement, à partir d'une permutation constructible avec une pile, on peut en écrire une recette. Par exemple, la recette de la permutation $(2, 4, 3, 1)$ est $AARAARRR$.

Question 5. À quelle permutation correspond la recette $ARAARRAR$?

Question 6. À quelle recette correspond la permutation $(3, 2, 4, 1)$?

Toute suite de A et de R n'est pas nécessairement une recette valide, par exemple la recette $ARRA$ n'est pas une recette valide puisque, lorsqu'on la suit, on doit retirer un élément alors que la pile est vide. La recette AAR n'est pas non plus valide parce que la pile n'est pas vide à la fin lorsqu'on a suivi la recette.

Question 7. Montrer qu'une recette est valide si et seulement si les propriétés suivantes sont vérifiées :
- Lorsqu'on la suit, à chaque étape, on a toujours ajouté au moins autant d'éléments qu'on en a retiré ;
- à la fin de la recette, on a retiré exactement autant d'éléments qu'on en a ajouté.

Question 8. Montrer que deux recettes valides différentes donnent nécessairement deux permutations différentes.

On pourra s'intéresser au premier moment lors duquel les deux recettes diffèrent.

Ainsi, le nombre de recettes valides de longueur $2n$ est exactement le nombre de permutations différentes que l'on peut faire de $\{1, 2, ..., n\}$ avec une pile.

On note R_n le nombre de recettes distinctes valides de longueurs $2n$. On remarque que $R_0 = 1$ car il n'y a qu'une recette possible, la recette vide. $R_1 = 1$ car on ne peut qu'ajouter un élément puis le retirer. $R_2 = 2$, en effet, on a les deux recettes possibles $AARR$ et $ARAR$.

Question 9. Combien vaut R_3 ?

Question 10 (⋆). Montrer que pour tout $n \geqslant 1$, on a :

$$R_n = \sum_{k=0}^{n-1} R_k R_{n-1-k}$$

On pourra raisonner sur le premier moment dans la recette après le début pour lequel on a retiré autant d'élément qu'on a ajouté et séparer la recette en deux parties.

On remarquera que le premier élément d'une recette est toujours un ajout, et que l'action avant le premier moment pour lequel on a ajouté autant d'élément qu'on en a retiré est toujours un retrait.

Enfin, on pourra s'aider d'un schéma.

Question 11. En déduire une fonction une fonction `r` de signature `int -> int` de sorte à ce que `r n` calcule R_n.

On essayera d'avoir une complexité quadratique (en $O(n^2)$) en utilisant par exemple un tableau pour se souvenir des valeurs déjà calculées de R_k pour $k \leqslant n$. Il n'est pas nécessaire de justifier la complexité.

Correction : On utilise un tableau qui se souvient de tous les éléments calculés jusque là. Le tableau est de taille $n + 1$ pour avoir les éléments de 0 à n.

```
let r n =
    let memoire = Array.make (n+1) 1 in
    for m = 2 to n - 1 do
        let somme = ref 0 in
        for k = 0 to m - 1 do
            somme:= !somme + memoire.(k) * memoire.(m - 1 - k)
        done ;
        memoire.(m)<- !somme
    done ;
    memoire.(n)
```

Question 12. Que se passe-t-il pour les permutations qu'on peut obtenir quand on utilise une file au lieu d'une pile ?

Partie II : Implémentation

Pour représenter une permutation, on utilise une liste d'entiers.

```
type permutation = int list
```

Ainsi, la permutation $(2, 3, 1, 4)$ sera représenté par la liste `[2; 3; 1; 4]`.

Pour représenter une recette, on a un type pour les opérations peut prendre deux valeurs, et on représente la recette à réaliser par une liste d'opérations :

```
type operation = Ajouter | Retirer
```

```
type recette = operation list
```

La permutation $AARARR$ est donc représentée par la liste `[Ajouter; Ajouter; Retirer; Ajouter; Retirer; Retirer]`.

Question 13. Proposer une fonction `est_valide` de signature `recette -> bool` qui renvoie si une recette est valide, c'est-à-dire, si elle correspond à une suite d'opérations qui peut être réalisée sans entraîner de retrait dans une pile vide.

Correction : Une fonction auxiliaire prend un argument supplémentaire qui est le nombre d'éléments ajoutés jusqu'ici On utilise l'évaluation paresseuse pour arrêter le calcul si on essaye

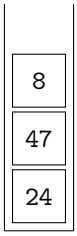
```
let est_valide r =
    let aux r nb = match r with
    | [] -> nb = 0
    | Ajouter::q -> aux q (nb + 1)
    | Retirer::q -> (nb>0)&&(aux q (nb - 1))
in aux r 0
```

Question 14. Dans cette question, on implémente une pile de sorte à pouvoir l'utiliser dans la suite. On propose d'utiliser une pile persistante, c'est-à-dire qu'on construit une nouvelle pile à chaque fois qu'on la modifie.

On propose le type pile suivant :

```
type pile = int list
```

La pile est donc une liste d'entier, et l'élément en tête de la liste est donc l'élément au sommet de la pile. Par exemple, la liste [8; 47; 24] représente la pile suivante :



Si les signatures des fonctions suivantes ne sont pas exactement celles demandées, mais qu'elles sont compatibles avec le type proposé, cela ne posera pas de problème.

1. Proposer une fonction est_vide de signature pile -> bool qui renvoie si une pile est vide.
2. Proposer une fonction empiler de signature pile -> int -> pile qui, à partir d'une pile et d'un entier, renvoie la pile qui a été modifiée en rajoutant l'entier au sommet de la pile.
3. Proposer une fonction depiler de signature pile -> int * pile qui, à partir d'une pile, renvoie l'entier au sommet de la pile et la pile dans laquelle on a retirée le sommet.

   Dans le cas où la pile est vide, on lévera une erreur grâce à l'expression suivante :

   ```
   failwith "Pile vide."
   ```

Correction : Voir cours.

Question 15. Proposer une fonction calculer de signature recette -> permutation qui renvoie la permutation obtenue en évaluant une recette passée en entrée.

On supposera que la recette est valide.

Correction : On utilise un accumulateur qui sauvegarde la permutation à l'envers dans une fonction auxiliaire, et on ajoute un argument pour le prochain élément à ajouter et la pile en cours.

```
let calculer recette =
    let rec aux r permut pile n = match r with
    | [] -> permut
    | Ajouter::q -> aux q permut (empiler n pile) (n+1)
    | Retirer::q -> let el, nouvelle_pile = depiler q in
        aux q (el::permut) nouvelle_pile n
in List.rev(aux recette [] 1)
```

Question 16. Proposer une fonction `donner_recette` de signature `permutation -> recette` qui renvoie la recette d'une permutation passée en argument.

On supposera que la permutation en entrée est bien une permutation, et qu'il existe une recette.

Correction : On utilise le fait qu'il existe une recette pour construire la recette de la manière suivante : on utilise un argument pour se souvenir du dernier élément ajouté, et on retire de la pile seulement si celui-ci est au moins égal à l'élément que nous cherchons :

```
let donner_recette permutation =
    let rec aux perm recette n = match perm with
        | [] -> recette
        | p::q when p>n -> aux perm (Ajouter::recette) (n+1)
        | p::q -> aux q (Retirer::recette) n
in List.rev (aux permutation  [] 0)
```

Problème : Stratégies d'évaluation

On s'intéresse dans ce problème aux stratégies d'évaluation. Lors du calcul d'une fonction, on peut calculer l'intrégralité des arguments de cette fonction pour ensuite calculer la fonction, ce que fait OCaml, mais on peut aussi différer le calcul en ne calculant la valeur des arguments que si nécessaire.

OCaml fait toujours des appels par valeur : c'est-à-dire que lors de l'appel d'une fonction `f a`, OCaml calcule la valeur de `a` puis appelle `f` sur cette valeur. On peut cependant utiliser des subterfuges pour faire des appels différés. Cela demande cependant quelques changements dans notre syntaxe de base.

Partie I : Tableau dynamique

Un tableau dynamique est une structure de donnée linéaire à taille variable pour laquelle il est rapide d'accéder à des éléments quelqu'en soit l'indice.

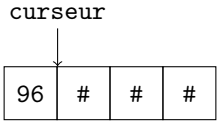
On propose le type suivant :

```
type 'a dynamique = {mutable memoire : 'a array ; mutable curseur : int}
```

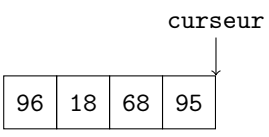
L'idée est la suivante : on dispose d'une mémoire (un tableau) de taille fixe et d'un curseur qui indique jusqu'à où les éléments sont importants.

Lorsque le tableau dynamique est vide, aucun élément n'est important, et le curseur pointe vers l'indice 0 (la dièse indique que l'élément peut être n'importe quel élément du bon type) :

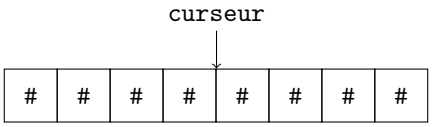
Quand on ajoute un élément, on modifie l'élément sous le tableau, et on déplace le curseur. Ainsi, en ajoutant 96 :



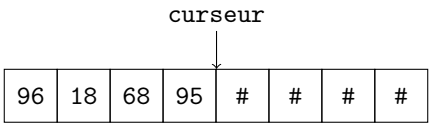
Ainsi, si on ajoute 18, 68 puis 95, on obtient le tableau dynamique suivant :



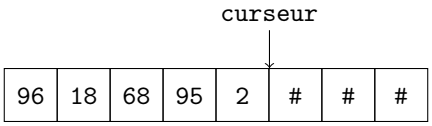
Mais si on doit ensuite ajouter un autre nombre (par exemple 2), il faut modifier le tableau qui sers de mémoire. On crée donc un tableau de taille doublée (si le tableau de la mémoire était vide, on crée un tableau de taille 1) :



On recopie les éléments qui étaient dans le précédent tableau :



Enfin, on peut rajouter l'élément que l'on voulait ajouter en mettant à jour le curseur :



Question 17. On part d'un tableau vide (la taille de la mémoire est zéro, elle est égale à [|||), et on ajoute successivement les éléments 4, 8, 15, 16, 23 puis 42.

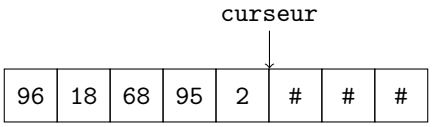
1. À quoi ressemble la mémoire avant et après chaque ajout ?
2. Proposer une valeur OCaml de type `int dynamique` pour représenter ce tableau dynamique dans son état final.

Question 18. Proposer une fonction `creer` de signature `unit -> 'a dynamique` qui crée un nouveau tableau dynamique vide.

On prendra soin d'initialiser correctement la mémoire de sorte à ce que le type du tableau soit bien le bon.

Question 19. Proposer une fonction `taille` de signature `'a dynamique -> int` qui renvoie la taille d'un tableau dynamique, c'est-à-dire le nombre d'éléments qui sont dans ce tableau dynamique.

On prendra soin de renvoyer la taille du tableau dynamique, et non la taille de la mémoire totale. Par exemple, le tableau dynamique suivant est de taille 5 :



Question 20. Proposer une fonction `acceder` de signature `'a dynamique -> int -> 'a option` de sorte à ce que `acceder t k` renvoie le k-ième élément x du tableau dynamique `t` sous la forme `Some x` si le tableau est défini jusqu'à k, et `None` sinon.

Question 21. Proposer une fonction `modifier` de signature `'a dynamique -> int -> a -> unit` de sorte à ce que `modifier t k x` modifie le k-ième élément de `t` pour qu'il soit égal à `x`.

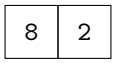
On supposera que l'indice est valide, et qu'il est bien inférieur strictement au nombre d'éléments pertinents dans la mémoire.

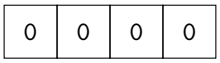
Question 22. Dans cette question, on traite de tableau classique au sens d'OCaml.

Proposer une fonction `copier` de signature `'a array -> 'a array -> unit` qui recopie les éléments du premier tableau en argument dans le second tableau en argument.

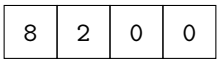
On supposera que le premier tableau est de taille inférieur au second. Il faudra donc copier tous les éléments du premier tableau, mais une partie des éléments du second tableau pourront ne pas être modifiés.

Par exemple, lorsqu'on exécute le code sur les deux tableaux suivants :





le second tableau doit ensuite être égal à :



Question 23.

1. Quelle valeur a le curseur juste avant un ajout qui entraîne un redimensionnement de tableau ?
2. Proposer une fonction `ajouter` qui ajoute un élément à la fin du tableau dynamique. On prendra soin de redimensionner la mémoire si cela est nécessaire.

Correction : Voir cours.

Partie II : Appel par nom

Dans un premier temps, on essaye de cacher le calcul d'une valeur derrière une fonction que l'on évaluera qu'au moment nécessaire. On propose un type qui propose de présenter le nom d'une variable comme la fonction qu'il faut exécuter pour l'obtenir. Cela nous donne, en OCaml, le type suivant :

```
type 'a nom = unit -> 'a
```

Ainsi, grâce à ce type, on peut évaluer les arguments d'une fonction seulement si nécessaire. On dit que la valeur contenue dans ce nom est le résultat de l'exécution de la fonction.

Dans la fonction suivante qui multiplie deux entiers, la valeur contenue dans y n'est calculée que si x a une valeur différente de 0. Dans le cas où x est évalué à 0, on sait que la réponse sera 0 dans tous les cas et on ne calcule pas la valeur contenue dans y.

```
let multiplier (x: int nom) (y: int nom) =
    let val_x = x () in
    if val_x = 0
        then 0
        else let val_y = y () in
            val_y * val_x
```

Ici `multiplier` est de type `int nom -> int nom -> int`, mais on peut vouloir modifier f de sorte à ce que f renvoie un type `int nom` :

```
let multiplier (x: int nom) (y: int nom): int nom =
fun () -> (
    let val_x = x () in
    if val_x = 0
        then 0
        else let val_y = y () in
            val_y * val_x)
```

Question 24. Proposer une fonction `evaluer` de signature `'a nom -> 'a` qui à partir d'une variable de type `'a nom` renvoie la valeur contenue dans ce nom.

Correction :

```
let evaluer n = n ()
```

Question 25. Proposer une fonction renvoyer_nom de signature 'a -> 'a nom qui à partir d'une valeur renvoie le nom type 'a nom qui contient cette valeur.

Correction :

```
let renvoyer_nom a = fun () -> a
```

Grâce à ces deux fonctions on peut réécrire la fonction multiplier de la manière suivante :

```
let multiplier (x: int nom) (y: int nom) =
    renvoyer_nom
        (let val_x = evaluer x in
        if val_x = 0 then 0
        else let val_y = evaluer y in
        val_x * val_y
    )
```

Question 26. Justifier que evaluer (renvoyer_nom x) est égal à la valeur de x.

Question 27. Proposer une fonction ou de signature bool nom -> bool nom -> bool nom qui renvoie le ou logique de deux variables par nom.

On procédera de manière paresseuse : si le premier argument permet de conclure quant à la valeur que doit renvoyer le résultat, on n'évaluera pas la valeur du deuxième argument.

On utilisera evaluer x et renvoyer_nom a quand c'est possible.

Correction :

```
let ou x y =
    renvoyer_nom (
        if evaluer x
            then true
            else evaluer y
    )
```

Question 28. Combien de fois doit on calculer la valeur de x () dans le calcul de l'expression suivante ?

```
(evaluer x) + (evaluer x)
```

C'est un souci, et on aimerait bien ne pas calculer plusieurs fois la valeur d'une même variable, car potentiellement, ce calcul peut être long.

Partie III : Appel par nécessité

De sorte à ne faire le calcul au plus qu'une fois, on construit un type de variable paresseuse qui contient à la fois le calcul qui peut être effectué pour en obtenir la valeur, mais aussi une mémoire pour cette valeur.

On modifie notre type pour qu'il contienne d'une part un champ qui correspondait au type précédent de la fonction à appeler pour obtenir la valeur qui nous intéresse, mais aussi un champ mutable qui contienne soit None, soit Some a où a est la valeur que prend la fonction f quand elle est évaluée.

```
type 'a paresse = {f: unit -> 'a; mutable mem: 'a option}
```

Ainsi, lorsque le calcul a déjà été fait, le champs mem contiendra Some a où a est la valeur contenue dans f, et None sinon. Grâce à cela, on peut ne faire le calcul qu'une seule fois au plus.

Question 29. Proposer une version modifiée evaluer_paresse de signature 'a paresse -> 'a qui utilise la mémoire contenue dans le type paresse si cela est possible, ou évalue la fonction contenue et met à jour la mémoire.

Correction :

```
let evaluer2 x = match x.mem with
| Some a -> a
| None -> let res = evaluer x.f in x.mem <- res ; res
```

Question 30. Proposer une version modifiée renvoyer_paresse de signature 'a -> 'a paresse de sorte à ce que renvoyer_paresse a renvoie la variable paresseuse contenant a pour ce nouveau type.

Correction :

```
let renvoyer_paresse x = {f = renvoyer x ; mem = Some x}
```

Question 31. Combien de fois réalise-t-on le calcul de x.f () dans le calcul de l'expression suivante ?

```
(evaluer_paresse x) + (evaluer_paresse x)
```

Partie IV : Suite paresseuse récurrente

On s'intéresse désormais à une suite évaluée paresseusement définie par une formule de récurrence. Le problème, c'est qu'on ne peut pas utiliser de formule de récurrence dans la fonction f sans rendre opaque l'accès aux appels récursifs.

Pour résoudre ce problèmes on va utiliser un combinateur qui va s'occuper de faire la partie récursive du calcul, et la fonction f prendra en argument la fonction qu'elle doit appeler pour faire les appels récursifs, faisant appel à ce combinateur de manière détournée.

On propose la fonction combinateur suivante :

```
let rec combinateur f x = f (combinateur f) x
```

La signature de combinateur est (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b.

Cette fonction permet de définir une fonction qui boucle à partir d'une fonction non récursive.

On se donne la fonction récursive suivante :

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

À partir de celle-ci, on peut écrire la fonction factorielle à l'aide de la fonction combinateur de la manière suivante :

```
let f cont n = match n with
| 0 -> 1
| _ -> n * cont (n-1)

let factorielle n = combinateur f n
```

Question 32. Quels sont les appels à f et combinateur réalisés dans le calcul de factorielle 2 ?

On donnera à chaque fois les arguments avec lesquels ces fonctions sont appelées ainsi que l'ordre dans lequel les appels ont lieu.

Question 33. On définit la suite de Fibonacci par la suite définie par $u_0 = 0$, $u_1 = 1$, et pour tout $n \in \mathbb{N}$, $u_{n+2} = u_{n+1} + u_n$.

Proposer une fonction g telle que combinateur g n calcule le terme u_n. On s'attend à ce que g soit de signature (int -> int) -> int -> int

On ne se soucira pas de la complexité de la fonction obtenue.

Correction :

```
let g cont n = match n with
| 0 | 1 -> n
| _ -> cont (n-1) + cont (n-2)
```

On se donne un type qui permette d'exploiter cette manière d'utiliser la récurrence. La suite paresseuse récurrente se souviendra de la fonction f et permettra de calculer la fonction fun n -> combinateur f n.

```
type 'a suite_paresseuse_recurrente = {f: (int -> 'a) -> int -> 'a ; mem: 'a option
    array}
```

Attention, la fonction f ne donne pas directement le résultat de la suite qui est calculée, il faudra passer par la fonction combinateur.

Question 34. Proposer une fonction creer_suite_paresseuse_recurrente de signature ((int -> 'a) -> int -> 'a) -> int -> 'a suite_paresseuse_recurrente qui crée une suite paresseuse recurrente à partir de sa fonction f et d'un entier n la taille du tableau mem.

Correction :

```
let creer_suite_paresseuse_recurrente f n =
    {f = f; mem = Array.make n None}
```

De sorte à pouvoir exploiter le tableau mem, on doit modifier la fonction combinateur.

Question 35. Completer la fonction combinateur2 de sorte à ce que combinateur2 mem f n calcule le même résultat que combinateur f n mais en vérifiant dans mem avant de faire le calcul si la valeur est déjà connue à n donné, et en mettant à jour éventuellement mem.

```
let rec combinateur2 mem f x = match mem.(x) with
| None -> ...
| Some res -> ...
```

Correction :

```
let rec combinateur2 mem f x = match mem.(x) with
| None -> let res = f (combinateur2 mem f) x in mem.(x) <- res ; res
| Some res -> res
```

Question 36. Proposer une fonction evaluer_suite de signature 'a suite_paresseuse_recurrente -> int -> 'a qui utilise combinateur2 pour évaluer la valeur d'une suite paresseuse récurrente en un entier n donné.

Correction :

```
let evaluer_suite s n = combinateur2 s.mem s.f n
```

Question 37. On reprend la fonction g définie à la question 33. On définit la fonction fibonacci de la manière suivante :

```
let fibonacci n =
    let s = creer_suite_paresseuse_recurrente g (n+1) in
    evaluer_suite s n
```

Montrer que la complexité temporelle de cette fonction est en $O(n)$.

Question 38. Pour l'instant, le tableau mem est de taille fixe. Quelle structure de donnée pourrait-on utiliser pour pouvoir utiliser une taille variable tout en conservant un accès rapide à chaque élément de la mémoire ?