

DS2 - documents de cours, TD, TP autorisés

Note : lorsque, dans une question, est demandée la complexité, c'est une fonction de la taille de la donnée qui est attendue. Lorsque c'est le comportement asymptotique, la réponse attendue est soit une notation \mathcal{O} , soit une notation Θ soit une notation Ω .

Vous pourrez utiliser toute fonction vue en cours, TD, TP à la condition expresse de spécifier correctement les entrées et les sorties.

Exercice 1 : Calcul avec les valeurs d'un ABR

On donne la définition d'arbre suivante :

```
type abr = Vide | Cons of noeud
and noeud = {
  valeur : int;
  filsg : abr;
  filsd : abr;
}
```

et une fonction récursive qui calcule le produit de toutes les valeurs des nœuds de l'arbre :

```
let rec produit_rec a =
  if a = Vide then
    1
  else
    let n = match a with | Cons n -> n | Vide -> failwith "arbre_vide" in
    n.valeur * (produit_rec n.filsg) * (produit_rec n.filsd)
```

Q 1.1 Ecrire en CAML une version non récursive de cette fonction. Vous pourrez utiliser toute structure de données annexe.

Corrigé

```
let produit ab =
  let p = Stack.create ()
  and r = ref 1
  in
  Stack.push ab p;
  while not (Stack.is_empty p) do
    let a = Stack.pop p
    in
    if a <> Vide then begin
      let n = match a with | Cons n -> n | Vide -> failwith "arbre_vide"
      in
      r := !r * n.valeur;
      Stack.push n.filsd p;
      Stack.push n.filsg p;
    end;
  done;
  !r
```

Exercice 2 : Suppression dans les tables de hachage

On suppose manipuler une table de hachage avec les définitions de type suivantes (les mêmes que celles du cours) :

```
type couple = {
  cle : string;
  val : int;
};
let table = couple array;
let occupe = bool array;
```

On suppose également disposer d'une fonction `h` dont la spécification est la suivante :

```
/*
  h [k] : retourne un entier positif ou nul calcule a partir de la
          chaine de caracteres [k]
  Note : cet entier n'est pas borne
*/
val h : string -> int;
```

On utilise une résolution des collisions par sondage linéaire et la même fonction d'insertion que celle du cours. La table ne peut contenir qu'une seule valeur associée à une clé donnée.

On souhaite disposer d'une fonction de suppression d'un couple (clé,valeur) de la table. La suppression aura pour effet de réaliser un décalage des couples ayant créé une collision. Par exemple, sur une table contenant 6 alvéoles :

(Clé,Valeur)	Adresse primaire		0 :		0 :
(ab,0)	1	AVANT	1 :	(ab,0)	APRES suppression de la clé "ae"
(bc,1)	5		2 :	(ae,2)	
(ae,2)	1		3 :	(cb,3)	
(cb,3)	2		4 :	(ax,4)	
(ax,4)	1		5 :	(bc,1)	
				5 :	

Q 2.1 Si la table ne contient que des clés ayant même adresse primaire que les indices de la table, décrire en français le travail qu'il y a à faire pour supprimer l'une des clés.

Corrigé

Il suffit d'indiquer que la case occupée ne n'est plus dans le tableau `occupe`.

Q 2.2 Dans le cas contraire (il y a au moins deux clés avec même adresse primaire, on souhaite en supprimer une), décrire en français ce qu'il y a à faire lorsqu'on rencontre une clé d'adresse primaire différente de celle de la clé à supprimer.

Corrigé

Ici c'est plus compliqué. Il faut décaler tous les couples sauf ceux qui sont rangés à leur adresse primaire, et s'arrêter lorsqu'on rencontre une alvéole libre. Il faut aussi mettre en place un test qui vérifie qu'on s'arrête une fois que toutes les cases ont été examinées.

Q 2.3 Ecrire le code CAML d'une fonction `supprimer [k]` qui supprime l'association de la clé `[k]` dans la table `[t]` si elle existe. Sinon elle ne fait rien.

Corrigé

```

let supprimer k =
  (* recherche l'adresse ou k est inseree *)
  let adresse = ref (h k);
  in
  (* recherche de l'alveole contenant la cle *)
  while occupe.(!adresse) && (table.(!adresse).cle <> k) do
    adresse := !adresse + 1;
  done;
  (* si k est bien dans la table *)
  if occupe.(!adresse) then begin
    let adressei = ref ((!adresse + 1) mod capacite)
    and tentative = ref 0
    in
    while occupe.(!adressei) && (!tentative < capacite) do
      if (h table.(!adressei).cle) != (!adressei) then begin
        table.(!adresse) <- table.(!adressei);
        occupe.(!adresse) <- true;
        occupe.(!adressei) <- false;
        adresse := !adressei;
        adressei := ((!adresse + 1) mod capacite);
      end else
        adressei := ((!adressei + 1) mod capacite);
        tentative := !tentative + 1;
      done;
    end;;
  end;;

```

Q 2.4 Quelle est la complexité asymptotique en temps de la fonction de suppression ? Justifier, indiquer l'opération prise en référence. (on suppose avoir inséré n clés dans la table, et que sa taille est M).

Corrigé

En $\Omega(1)$, par exemple si la clé à supprimer est suivie d'une alvéole libre), en $\mathcal{O}(n)$, par exemple si toutes les clés ont même adresse primaire.

Exercice 3 : Comparaison de listes

Etant donné deux listes d'entiers compris entre 0 et 99, on souhaite écrire un prédicat qui teste si les deux listes de longueur respective n et m sont *équivalentes*. On dira que deux listes sont équivalentes si elles contiennent les mêmes éléments, peu importe l'ordre et le nombre d'occurrences. Par exemple :
 [5; 6; 1; 9; 5] et [1; 9; 6; 1; 1; 5] sont équivalentes,
 [5; 6; 1; 9; 5] et [6; 1; 9; 1] ne sont pas équivalentes (la valeur 5 n'apparaît pas).

On suppose que les listes utilisées correspondent à celles qui ont été vues en TP, on ne peut donc les manipuler qu'avec des itérateurs. Il est strictement interdit d'utiliser une fonction de tri pour résoudre le problème.

Q 3.1 Décrire en français la stratégie que vous allez utiliser et la(les) structure(s) de données annexes.

Corrigé

Une solution, parmi toutes celles qui utilisent un tableau annexe puisqu'on sait qu'on rencontrera au maximum 100 valeurs différentes. On pourra utiliser deux tableaux $t1$ et $t2$ d'entiers indicés de 0 à 99 initialisés à faux. Pour chaque élément i de la première liste on met à vrai la valeur de $t1.(i)$, idem pour la seconde liste avec $t2$. Il reste à comparer les deux tableaux qui doivent être identiques pour avoir deux listes équivalentes.

Q 3.2 Ecrire en CAML la fonction booléenne `equivalents` prenant en entrée deux listes et retournant vrai si les deux listes sont équivalentes, faux sinon.

Corrigé

```

let equivalents l1 l2 =
  let t1 = Array.make 100 false
  and t2 = Array.make 100 false
  and eq = ref true
  in
  let it1 = iterateur_en_debut l1
  in
  while not (est_en_fin it1) do
    t1.(valeur it1) <- true;
    avancer it1;
  done;
  t1.(valeur it1) <- true;
  let it2 = iterateur_en_debut l2
  in
  while not (est_en_fin it2) do
    t2.(valeur it2) <- true;
    avancer it2;
  done;
  t2.(valeur it2) <- true;
  for i = 0 to 99 do
    eq := !eq && (t1.(i) = t2.(i));
  done;
  !eq

```

Q 3.3 Quelle est la complexité asymptotique en temps de votre algorithme? Justifier.

Corrigé

En $\Theta(n + m)$ puisqu'on parcourt une fois chaque liste. Le parcours des tableaux à la fin est en temps constant puisque qu'on parcourt un nombre constant de cases.

Q 3.4 Quelle est la complexité asymptotique en espace de votre algorithme? Justifier.

Corrigé

La place occupée par les deux tableaux, soit $\Theta(1)$ puisque 100 est une constante.

Exercice 4 : Construction d'un ABR optimal par programmation dynamique

Etant donné un ensemble de valeurs entières distinctes croissantes $\{v_1, \dots, v_n\}$ et des fréquences de recherche de ces valeurs $\{f_1, \dots, f_n\}$ on souhaite construire un ABR \mathcal{T} contenant ces valeurs et qui minimise la fonction de coût suivante afin d'optimiser l'arbre par rapport aux recherches effectuées :

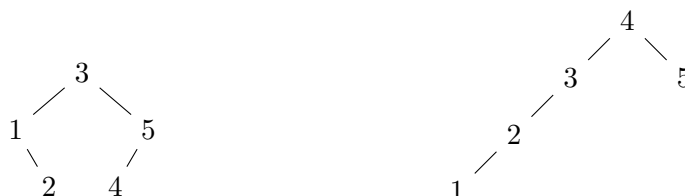
$$\text{coût}(\mathcal{T}) = \sum_{i=1}^n f_i \times p_{\mathcal{T}}(v_i)$$

où $p_{\mathcal{T}}(v_i)$ est la profondeur du nœud contenant la valeur v_i dans \mathcal{T} . On considère ici que la profondeur de la racine vaut 1, celles de ses fils 2, etc.

Supposons par exemple qu'on construise un arbre avec les valeurs et les fréquences de recherche suivantes :

v_i	1	2	3	4	5
f_i	0.1	0.1	0.5	0.2	0.1

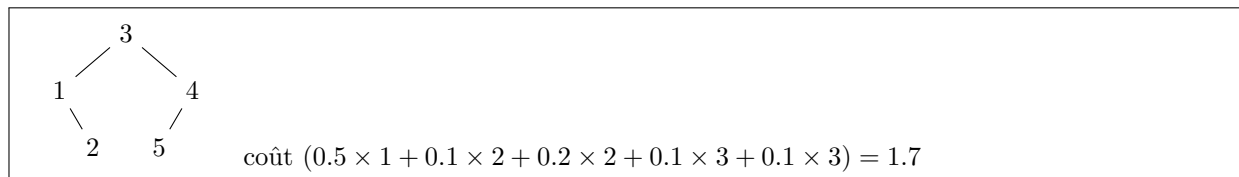
et si on considère les deux ABR suivants :



l'arbre de gauche aura pour coût $(0.5 \times 1 + 0.1 \times 2 + 0.1 \times 2 + 0.1 \times 3 + 0.2 \times 3) = 1.8$ alors que l'arbre de droite aura pour coût $(0.2 \times 1 + 0.5 \times 2 + 0.1 \times 2 + 0.1 \times 3 + 0.1 \times 4) = 2.1$

Q 4.1 Proposer un arbre de meilleur coût (dessiner l'arbre et détailler le calcul du coût comme ci-dessus).

Corrigé



On peut voir le problème récursivement : quelle est la meilleure valeur v_k à mettre à la racine qui minimise le coût des sous-arbres gauche contenant les valeurs v_1 à v_{k-1} et droit contenant les valeurs v_{k+1} à v_n ?

Si on note $\mathcal{T}_{i,j}$ l'arbre optimal pour les valeurs v_{i+1} à v_j , $w_{i,j} = \sum_{k=i+1}^j f_k$, et $c_{i,j}$ le coût de l'arbre $\mathcal{T}_{i,j}$ ($c_{i,i} = 0$ pour l'arbre vide $\mathcal{T}_{i,i}$), alors on peut prouver que¹ :

$$c_{i,j} = w_{i,j} + \min_{k \in \{i+1..j\}} (c_{i,k-1} + c_{k,j})$$

Il suffit donc de calculer $c_{0,n}$ pour obtenir le coût minimum.

Cette expression récursive permet de mettre en œuvre une résolution par programmation dynamique. On supposera disposer d'une fonction w à deux paramètres permettant de calculer $w_{i,j}$.

Q 4.2 Quelle est la dimension de la table de programmation dynamique ?

Corrigé

Deux.

Q 4.3 Quelle est la taille de la table de programmation dynamique ?

Corrigé

$(n + 1) \times (n + 1)$

Q 4.4 Ecrire le code permettant d'initialiser la table sachant que les fréquences sont contenues dans un tableau \mathbf{f} .

Corrigé

En suivant ce qui est écrit dans l'énoncé, on sait que $c_{i,i} = 0$. Voir le code complet pour le code CAML.

Q 4.5 Grâce à un dessin, indiquer toutes les cases qu'il faut calculer avant de calculer $c_{i,j}$.

Corrigé

En développant un calcul de $c_{i,j}$ on se rend compte qu'il faut connaître les cases de la colonne 0 au dessus de la ligne j ainsi que les cases de la ligne j entre les position $j + 1$ et i .

Par exemple :

$$c_{0,3} = w_{0,3} + \min \left\{ \begin{array}{l} c_{0,0} + c_{1,3} \\ c_{0,1} + c_{2,3} \\ c_{0,2} + c_{3,3} \end{array} \right\}$$

Il faut donc d'abord calculer $c_{0,1}$, $c_{0,2}$, $c_{1,3}$ et $c_{2,3}$.

1. ne cherchez pas à prouver cette formule !

Q 4.6 Ecrire le code permettant de remplir la table.

Corrigé

En pseudocode, voici la réponse :

```
Cout-ABR-Optimal(p, n)
// initialisation
pour i allant de 0 a n faire
  c(i,i) = 0
// remplissage
pour l allant de 1 a n faire
  pour i allant de 0 a n - l faire
    j = i+l
    soit m la valeur de k telle que c(i,k-1) + c(k,j) soit minimum
    c(i,j) = wi,j + c(i,m-1) + c(m,j)
```

et la traduction en CAML :

```
let cout_optimal () =
  let n = Array.length v
  in
  let t = Array.make_matrix (n+1) (n+1) 0.0
  in
  (* initialisation *)
  for i = 0 to n do
    t.(i).(i) <- 0.0
  done;
  (* remplissage *)
  for j = 1 to n do
    for i = j-1 downto 0 do
      let meilleur_c = ref (t.(i).(i) +. t.(i+1).(j))
      in
      for k = i+2 to j do
        let c1 = t.(i).((k-1))
        and c2 = t.(k).(j)
        in
        if ((c1 +. c2) < !meilleur_c) then
          meilleur_c := c1 +. c2;
        done;
      t.(i).(j) <- (w i j) +. !meilleur_c;
    done;
  done;
  t.(0).(n)
```

Q 4.7 Sur le même dessin, indiquer la case où se trouve le résultat final.

Corrigé

Lé réponse est dans l'énoncé : $c_{0,n}$.

Q 4.8 Quelle est la complexité en temps du calcul de $c_{(0,n)}$? Justifier, indiquer l'opération prise en référence.

Corrigé

Le calcul d'une case nécessite la recherche du max et est donc en $\mathcal{O}(n)$. Il y a $\Theta(n^2)$ cases à calculer. La complexité totale est donc en $\mathcal{O}(n^3)$.