

DS1 - 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 : Questions de cours [3 points] (réponses sans justification)

Q 1.1 [0.75 point] Si je prouve qu'un algorithme est en $\mathcal{O}(n^2)$, peut-il être en $\Theta(n)$ sur certains exemplaires? Répondre par oui ou non.

Corrigé

Oui bien sûr. Puisque l'algorithme est en $\mathcal{O}(n^2)$ il s'exécute dans le pire des cas en $\Theta(n^2)$, il peut donc y avoir des instances qui s'exécutent en $\Theta(n)$ puisque $n < n^2$.

Q 1.2 [0.75 point] Si je prouve qu'un algorithme est en $\Omega(n \log n)$, peut-il être en $\mathcal{O}(n)$ sur certains exemplaires? Répondre par oui ou non.

Corrigé

Non. Puisque l'algorithme est en $\Omega(n \log n)$ il s'exécute dans le meilleur des cas en $\Omega(n^2)$, il ne peut pas y avoir d'instances qui s'exécutent en $\mathcal{O}(n)$ puisque $n < n \log n$.

Q 1.3 [0.75 point] Le coût d'un ajout en queue dans une liste simplement chaînée contenant n éléments est (une seule réponse possible) :

- (a) $\mathcal{O}(n^2)$ (b) $\mathcal{O}(n)$ (c) $\Theta(n)$ (d) $\Theta(1)$?

Corrigé

$\Theta(n)$ puisqu'il faut parcourir la liste. Pas $\mathcal{O}(n)$ puisqu'on a toujours le même travail à faire, quelque soit le cas.

Q 1.4 [0.75 point] Le coût d'un ajout en queue dans une liste avec sentinelle contenant n éléments est (une seule réponse possible) :

- (a) $\mathcal{O}(n)$ (b) $\mathcal{O}(1)$ (c) $\Theta(\log n)$ (d) $\Theta(1)$?

Corrigé

$\Theta(1)$ puisque grâce à la sentinelle on a accès directement à la queue.

Exercice 2 : Multiplication de vecteurs de bits [6 points]

On souhaite réaliser la multiplication de deux nombres x et y représentés chacun sur n bits.

Une stratégie pour faire cela efficacement consiste à décomposer chacun des nombres en deux vecteurs de bits : ceux de poids fort et ceux de poids faible et d'utiliser uniquement les opérations de somme sur les entiers (+ et -), le décalage binaire noté \ll et le ou binaire noté $|$.

On notera x_L le vecteur de taille $\frac{n}{2}$ représentant les bits de poids fort de x et x_R le vecteur de taille $\frac{n}{2}$ représentant les bits de poids faible de x . Par exemple, si $x = 10011100$ alors $x_L = 1001$ et $x_R = 1100$.

En utilisant la propriété suivante :

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

on aboutit au pseudo-code suivant :

```

/* fastmultiply : vecteur de bits, vecteur de bits -> vecteur de bits */
function fastmultiply(x, y)
  n = max(size of x, size of y)
  if n = 1 then
    return xy
  else
    xL, xR = leftmost n/2, rightmost n/2 bits of x
    yL, yR = leftmost n/2, rightmost n/2 bits of y
    P1 = fastmultiply(xL,yL)
    P2 = fastmultiply(xR,yR)
    P3 = fastmultiply(xL + xR,yL + yR)
    return P1 << 2^n | (P3 - P1 - P2) << 2^(n/2) | P2

```

Q 2.1 [1 point] Donner l'équation de récurrence donnant le nombre d'appels récursifs de cet algorithme.
Corrigé

$$c_f(0) = 0; c_f(n) = 3 * c_f(n/2) + 3$$

Q 2.2 [1.5 points] En déduire le comportement asymptotique de l'algorithme `fastmultiply`.
Corrigé

$$n^{\log_b a} = n^{\log_2 3} = n^{1.59}$$

$$f(n) = 3 = \mathcal{O}(n^{1.59-0.1})$$

Cas 1 du théorème général.

$$c_f(n) = \Theta(n^{\log_2 3})$$

On souhaite maintenant convertir un entier **décimal** 10^n (1 suivi de n 0, par exemple $10^3 = 1000$ soit mille) en binaire. On propose l'algorithme suivant pour $n = 2^p$:

```

/* pwr2bin : decimal -> vecteur de bits */
function pwr2bin(n)
  if n = 1 then
    return 1010 (* dix en binaire *)
  else
    z = ???
    return fastmultiply(z, z)

```

Q 2.3 [1 point] Quelle valeur doit-on assigner à z ?
Corrigé

$$z = \text{pwr2bin}(n/2);$$

Q 2.4 [1 point] Donner l'équation de récurrence donnant le nombre d'appels récursifs de cet algorithme.
Corrigé

Première réponse acceptée, on ne compte que les appels à `pwr2bin`. On a donc :

$$c(n) = c(n/2) + 1$$

Seconde réponse acceptée, on compte également les appels à `fastmultiply` mais c'est alors un peu plus compliqué. On approxime la taille en bits de 10^n par $\log_2 10^n$. On a alors :

$$c(n) = c(n/2) + 1 + c_f(\log_2 10^{\frac{n}{2}})$$

Q 2.5 [1.5 points] En déduire le comportement asymptotique de l'algorithme `pwr2bin`.

Corrigé

Avec la première équation :

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

$$f(n) = 1 = \Theta(n^0)$$

Cas 2 du théorème général.

$$c(n) = \Theta(n^0 \log n) = \Theta(\log n)$$

Avec la seconde équation :

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = 1 + \Theta((\log_2 10^{\frac{n}{2}})^{\log_2 3}) = \Theta\left(\left(\frac{n}{2} \log_2 10\right)^{\log_2 3}\right) = \Theta(n^{\log_2 3}) = \Omega(n^{0+\varepsilon})$$

Cas 3 du théorème général. Il faut montrer que

$$1 \times f\left(\frac{n}{2}\right) < cf(n)$$

pour $c < 1$, ce qui est vrai ici pour par exemple $c = \frac{1}{2}$. On en conclut que

$$c(p) = \Theta(n^{\log_2 3})$$

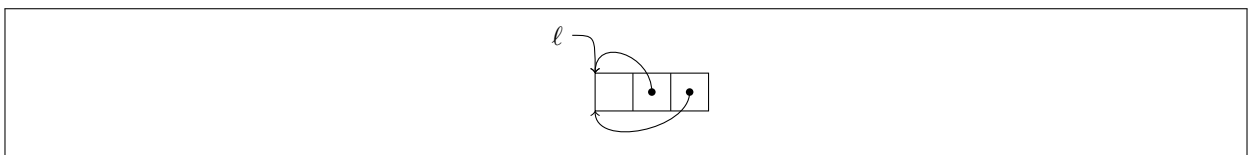
Exercice 3 : Listes circulaires [3 points]

On considère des listes circulaires avec les définitions de type suivantes :

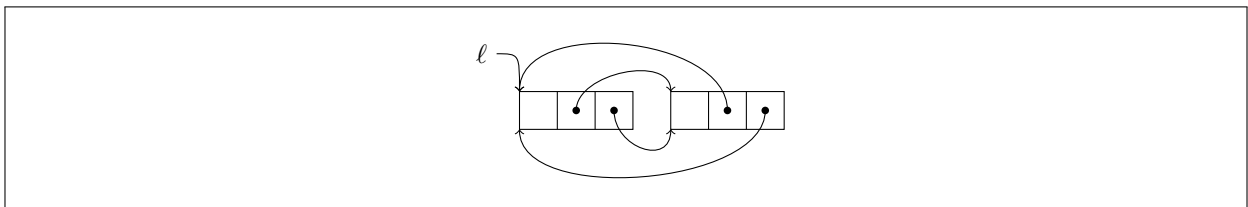
```
type liste = Vide | Cons of cellule
and cellule = {
  valeur : int;
  mutable suivant : liste;
  mutable precedent : liste;
}
```

On considère disposer d'une fonction `la_cellule` qui étant donnée une liste retourne la cellule correspondante, lève une exception `ListeVide` sinon.

Q 3.1 [0.5 point] Dessiner une représentation de la liste avec un seul élément. *Corrigé*



Q 3.2 [0.5 point] Dessiner une représentation de la liste avec deux éléments. *Corrigé*



Q 3.3 [2 points] Ecrire la fonction d'ajout en tête d'un élément e à une liste l (la liste l peut être vide).

Corrigé

```

let ajouter e l =
  try
    let c = la_cellule l
    in
    let n = { valeur = e; suivant = l; precedent = c.precedent }
    in
    (la_cellule n.precedent).suivant <- Cons n;
    c.precedent <- Cons n;
    Cons n;
  with
  | ListeVide ->
    let c = { valeur = e; suivant = Vide; precedent = Vide }
    in
    c.suivant = Cons c;
    c.precedent = c.suivant;
    c.suivant;

```

Exercice 4 : CycleSort [8 points]

L'algorithme CYCLESORT a été proposé en 1990 pour trier des tableaux d'objets auxquels on pouvait associer à chacun une clé entière dans l'intervalle $[1..n]$. Le problème du tri de tableaux d'objets peut alors se réduire au tri de tableaux d'entiers : en réalisant les mêmes opérations d'échange de valeurs sur le tableau d'objets que sur le tableau d'entiers correspondant, on aura trié les objets.

Si il existe une bijection entre les objets et l'intervalle $[1..n]$, on a alors à trier des tableaux particuliers que sont les permutations. On rappelle qu'une permutation de taille n est la suite des éléments de 1 à n dans n'importe quel ordre.

Ici, et afin de pouvoir manipuler les permutations dans des tableaux, on définira une permutation à n éléments comme la suite des éléments de 0 à $n - 1$ dans n'importe quel ordre. Par exemple si $n = 10$: $[|2; 4; 7; 6; 3; 9; 8; 0; 5; 1|]$ est une permutation.

CYCLESORT est basé sur la remarque que dans une permutation on peut détecter des cycles. Un cycle de taille m est une suite d'éléments $t.(i_1), t.(i_2), \dots, t.(i_m)$ qui sont tels que $t.(i_1) = i_2, t.(i_2) = i_3, \dots, t.(i_m) = i_1$. Par exemple dans la permutation ci-dessus $(2, 7, 0)$ est un cycle puisque $t.(0) = 2, t.(2) = 7, t.(7) = 0$.

Réorganisation d'un cycle Pour un cycle donné, on peut assez simplement remettre ses éléments à la bonne place dans la permutation triée : il suffit de faire « tourner » les éléments du cycle. Sur l'exemple on passera $t.(0) = 2$ dans $t.(2)$ (il sera donc bien rangé), $t.(2) = 7$ dans $t.(7)$ et $t.(7) = 0$ dans $t.(0)$.

Q 4.1 [1 point] Indiquer tous les cycles de la permutation $[|4; 7; 2; 1; 3; 8; 9; 0; 5; 6|]$.

Corrigé

```

4 -> 3 -> 1 -> 7 -> 0
2
8 -> 5
9 -> 6

```

Q 4.2 [2 points] Donner le code en CAML d'une procédure `reorganiser` qui étant donné un tableau t et un indice i réorganise le cycle qui passe par la position i . Par exemple :

```

# let t = [|2; 4; 7; 6; 3; 9; 8; 0; 5; 1|];;
# reorganiser t 0;;
# t;;
- : int array = [|0; 4; 2; 6; 3; 9; 8; 7; 5; 1|]

```

Corrigé

```

let reorganiser_bis t i =
  while t.(i) <> i do
    let tmp = t.(t.(i))
    in
    t.(t.(i)) <- t.(i);
    t.(i) <- tmp;
  done;

```

Q 4.3 [0.5 point] Si m est la taille d'un cycle, combien d'affectations de valeurs seront nécessaires pour réorganiser un cycle? Justifier. *Corrigé*

$3 \times (m - 1)$

Q 4.4 [0.5 point] Combien y a-t-il de cycles au maximum dans une permutation de taille n ? *Corrigé*

n

Q 4.5 [0.5 point] Combien y a-t-il de cycles au minimum dans une permutation de taille n ? *Corrigé*

1

Processus de tri Une procédure de tri utilisant cette propriété pourrait être :

```

pour chaque cycle c faire
  reorganiser le cycle c
fin pour

```

Q 4.6 [1 point] Est-ce un tri sur place? Justifier. *Corrigé*

Oui, les éléments du tableau sont déplacés sans création d'un nouveau tableau.

Q 4.7 [0.5 point] Indiquez dans votre code de la procédure reorganiser entre quelles lignes il faudra ajouter un incrément de compteur pour calculer le nombre d'affectations.

Q 4.8 [1 point] Quel est le meilleur des cas pour le nombre d'affectations. Expliquer. *Corrigé*

On a n cycles comme dans la permutation $[1; 2; \dots; n]$, chaque cycle contient un élément ce qui nécessite 0 affectations.

Q 4.9 [1.5 point] En identifiant le pire des cas, donner le comportement asymptotique de l'algorithme pour le nombre d'affectations?

Corrigé

Le pire des cas est lorsqu'il n'y a qu'un seul cycle de taille n comme par exemple dans la permutation $[n; 1; 2; \dots; n-1]$. On a alors $n - 1$ affectations. On est donc en $\Theta(n)$.