Université Lille1 Sciences et Technologies

Info 204 - ASD

2011–2012
Licence STS - Semestre 4

**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 $\Theta$ soit une notation $\Omega$.

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 [4 points]

**Q 1.1** [2 points] Un algorithme s'exécute dans le meilleur des cas en réalisant $c_m(n) = 2n + 1$ opérations et dans le pire des cas en $c_p(n) = 2^n + 4n - 2$ opérations pour une donnée de taille $n$. Indiquez et prouvez son comportement asymptotique.

*Corrigé*

L'algorithme est en $\Omega(n)$ car $n \leq c_m(n) \leq 3n \quad \forall n > 1$ et en $\mathcal{O}(2^n)$ car $2^n \leq c_p(n) \leq 2 \times 2^n \quad \forall n > 2$.

**Q 1.2** [2 points] Le nombre d'appels récursifs d'un algorithme récursif est donné par la formule :

$$c(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ou } 1 \\ 16c\left(\frac{n}{4}\right) + n^3 & \text{sinon} \end{cases}$$

Indiquez et prouvez son comportement asymptotique.

*Corrigé*

$a = 16, b = 4, \log_4 16 = 2$ et $f(n) = n^3 = \Omega(n^2)$. De plus $16 \times (\frac{n}{4})^3 = \frac{n^3}{4} \leq \frac{n^3}{2}$ et donc $c(n) = \Theta(n^3)$. (cas 3 du théorème général)

## Exercice 2 : Calculer une complexité et améliorer un algorithme [3 points]

Dans cet exercice on considère des tableaux d'entiers. On dit qu'un élément $e$ est majoritaire dans un tableau $t$ de longueur $n$ si $t$ contient plus de $\frac{n}{2}$ occurrences de $e$. Pour simplifier, on supposera traiter le cas particulier où il existe toujours un élément majoritaire. On propose la fonction `element_majoritaire` qui calcule l'élément majoritaire de $t$ en utilisant la fonction `decompte`.

```
let decompte t e =
  let n = Array.length t
  and nb = ref 0
  in
  for i = 0 to n-1 do
    if t.(i) = e then nb := !nb + 1;
  done;
  !nb
```

```
let element_majoritaire t =
  let n = Array.length t
  and max = ref 0
  and valmax = ref 0
  in
  for i = 0 to n-1 do
    let nb = decompte t t.(i)
    in
    if nb > !max then begin
      max := nb;
      valmax :=  t.(i);
    end
  done;
  !valmax
```

**Q 2.1** [1 point] Quelle est l'opération à prendre en compte pour le calcul de la complexité en temps ?

*Corrigé*

La comparaison d'éléments du tableau `t`.

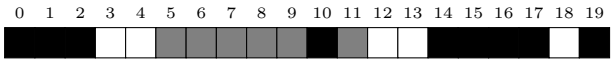
**Q 2.2** [2 points] Quelle est la complexité en temps ? Justifiez.

*Corrigé*

Le nombre de comparaisons de `decompte` est exactement $n$. Cette fonction est appelée $n$ fois sur le tableau `t` dans la recherche de l'élément majoritaire. On a donc $n^2$ comparaisons.

## Exercice 3 : Concevoir un algorithme d'une complexité donnée [6 points]

On s'intéresse à un problème d'analyse d'images : « Etant donné une ligne de pixels de différentes couleurs, on souhaite identifier la position et la longueur du plus long segment monochrome dans la ligne. » Par exemple, étant donné la ligne suivante :




notre algorithme indiquera que le plus long segment monochrome débute en position 5 et est de longueur 5. Pour simplifier, on supposera qu'une ligne de pixels est représentée par un tableau d'entiers, chaque entier codant pour une couleur. La ligne de pixels ci-dessus sera par exemple représentée par :

```
[| 56;56;56;0;0;34;34;34;34;34;56;34;0;0;56;56;56;56;0;56 |]
```

**Q 3.1** [2 points] Proposez une fonction `plsm` écrite en CAML qui résolve ce problème dont la complexité en temps soit **linéaire** (on compte le nombre de comparaisons de couleurs). La fonction prend en entrée un tableau d'entiers $t$ et retourne un couple avec la position $p$ et la longueur $\ell$ tel que $\ell$ est la longueur du plus long segment monochrome et $p$ la position à laquelle il débute.

*Corrigé*

On parcourt le tableau, à chaque fois qu'on change de couleur on réinitialise à 1 un compteur qui va calculer le nombre de pixels successifs de la même couleur. On n'oublie pas de se souvenir du max.

```
let plsm t =
  let couleur_courante = ref t.(0)
  and compteur = ref 1
  and max = ref 1
  in
  for i = 1 to (Array.length t) - 1 do
    if t.(i) <> !couleur_courante then begin
      if max < compteur then
        max := !compteur;
      couleur_courante := t.(i);
      compteur := 1;
    end else
      compteur := !compteur + 1;
  done;
  if max < compteur then
    max := !compteur;
  !max
```

**Q 3.2** [1 point] Justifiez la complexité de votre algorithme.

*Corrigé*

Comme on a un seul parcours de tableau et qu'à chaque fois on fait une seule comparaison de couleur, on a bien un algorithme linéaire.

On souhaite maintenant obtenir les couleurs dominantes de la ligne. Le nombre de couleurs dominantes que l'on souhaite extraire dépend de l'application réalisée. Toujours sur l'exemple ci-dessus, l'extraction d'une couleur dominante retournera 56 et l'extraction des deux couleurs dominantes retournera 56 et 34.

**Q 3.3** [2 points] Ecrivez une fonction `couleurs_dominantes` en CAMLqui prend en entrée une ligne de pixels $t$, un entier $k$ et retourne un tableau avec les $k$ couleurs dominantes de $t$ (on supposera que le nombre de couleurs de $t$ est toujours supérieur ou égal à $k$). La complexité de votre fonction devra être en $\mathcal{O}(n \log n)$ (on compte toujours le nombre de comparaisons de couleurs).

*Corrigé*

On trie le tableau de pixels avec un tri fusion ou un quicksort puis on compte le nombre d'occurrences de chaque couleur puis on recherche dans ces nombres d'occurrences les $k$ plus grands. Si on considère que $k$ est une constante, alors on pouvait aussi faire le tri puis utiliser la fonction de la question précédente.

*Corrigé*

```
let couleurs_dominantes t k =
  let cmp a b = if a > b then 1 else if a = b then 0 else -1
  and tt = Array.copy t
  and n  = Array.length t
  in
  Array.sort cmp tt;
  let couleurs = Array.make n tt.(0)
  and nb_occ   = Array.make n 0
  and p = ref 0
  in
  let couleur_courante = ref tt.(0)
  and compteur = ref 1
  in
  (* on calcule pour chaque couleur son nombre d'occurrences *)
  for i = 1 to (Array.length t) - 1 do
    if tt.(i) <> !couleur_courante then begin
      nb_occ.(!p)   <- !compteur;
      couleurs.(!p) <- !couleur_courante;
      p := !p + 1;
      couleur_courante := tt.(i);
      compteur := 1;
    end else
      compteur := !compteur + 1;
  done;
  nb_occ.(!p)   <- !compteur;
  couleurs.(!p) <- !couleur_courante;
  p := !p + 1;
  (* on extrait les k couleurs dominantes *)
  let r = Array.make k 0
  in
  for i = 0 to k-1 do
    let max = ref nb_occ.(0)
    and pmax = ref 0
    in
    for j = 0 to !p-1 do
      if nb_occ.(j) > !max then begin
        max := nb_occ.(j);
        pmax := j;
      end;
    done;
    r.(i) <- couleurs.(!pmax);
    nb_occ.(!pmax) <- -1;
  done;
  r
```

*Corrigé*

ou bien

```
let couleurs_dominantes t k =
  (* fonction de comparaison de couleurs *)
  let cmp1 c1 c2 = if c1 > c2 then 1 else if c1 = c2 then 0 else -1 in
  (* fonction de comparaison de couples nb occ / couleur *)
  let cmp2 (n1,c1) (n2,c2) = if n1 > n2 then 1 else if n1 = n2 then 0 else -1
  and tt = Array.copy t
  and n  = Array.length t
  in
  Array.sort cmp1 tt;
  let couleurs = Array.make n (0,0)
  and p = ref 0
  in
  let couleur_courante = ref tt.(0)
  and compteur = ref 1
  in
  (* on calcule pour chaque couleur son nombre d'occurrences *)
  for i = 1 to (Array.length tt) - 1 do
    if tt.(i) <> !couleur_courante then begin
      couleurs.(!p) <- (!compteur,!couleur_courante);
      p := !p + 1;
      couleur_courante := tt.(i);
      compteur := 1;
    end else
      compteur := !compteur + 1;
  done;
  couleurs.(!p) <- (!compteur,!couleur_courante);
  p := !p + 1;
  (* on extrait les k couleurs dominantes *)
  Array.sort cmp2 couleurs;
  Array.sub couleurs (n-k) k
```

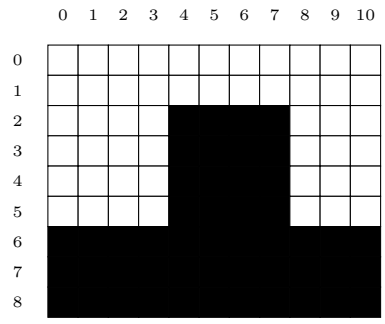
**Q 3.4** [1 point] Justifiez la complexité de votre algorithme.

*Corrigé*

Le tri en en $\mathcal{O}(n \log n)$ et le décompte des occurrences de chaque couleur est linéaire. L'extraction des $k$ couleurs dominantes ensuite ne nécessite pas de comparaison de **couleur**.

## Exercice 4 : Mettre en œuvre un algorithme de programmation dynamique [7 points]

On s'intéresse à nouveau dans cet exercice à un problème d'analyse d'image. On considère cette fois des images en noir et blanc et le problème qu'on veut traiter est la détection du plus grand carré de cases noires dans l'image. Par exemple, pour l'image suivante :




l'algorithme indiquera que le plus grand carré noir se situe en case (4,2) [1] et est de longueur 4.

1. D'autres coordonnées sont possibles – (4,3), 4,4), (4,5) – , c'est simplement celles que mon algorithme calcule

On propose la récursion suivante pour résoudre le problème. On note $\ell(i,j)$ la longueur du plus grand carré noir se terminant en position $(i-1, j-1)$ dans l'image. On a alors :

$$\ell(i,j) = \begin{cases} 0 & \text{si la case de l'image en position } (i-1,j-1) \text{ est blanche} \\ 1 + \min(\ell(i-1,j-1), \ell(i-1,j), \ell(i,j-1)) & \text{sinon} \end{cases}$$

et

$$\ell(i,0) = \ell(0,j) = 0 \text{ pour } 0 \leq i \leq n, 0 \leq j \leq m$$

L'obtention du résultat, c'est-à-dire la plus grande longueur $L$, consiste à sélectionner le couple $(i,j)$ maximisant $l(i,j)$ :

$$L = \max_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}} \ell(i,j)$$

si on considère une image de taille $n \times m$.

Nous allons donc mettre en œuvre une résolution par programmation dynamique. Pour simplifier, on supposera qu'une image est donnée par un tableau à deux dimensions dont les cases contiennent soit 0 soit 1, 0 indiquant les cases blanches, et 1 les cases noires. On manipulera trois variables déclarées comme ci-dessous :

```
let img =
  [|
    [| 0;0;0;0;0;0;1;1;1 |];
    [| 0;0;0;0;0;0;1;1;1 |];
    [| 0;0;0;0;0;0;1;1;1 |];
    [| 0;0;0;0;0;0;1;1;1 |];
    [| 0;0;1;1;1;1;1;1;1 |];
    [| 0;0;1;1;1;1;1;1;1 |];
    [| 0;0;1;1;1;1;1;1;1 |];
    [| 0;0;1;1;1;1;1;1;1 |];
    [| 0;0;0;0;0;0;1;1;1 |];
    [| 0;0;0;0;0;0;1;1;1 |];
    [| 0;0;0;0;0;0;1;1;1 |]
  |]
;;
let n = Array.length img;;
let m = (Array.length img.(0));;
```

**Q 4.1** [1 point] Quelle est la taille de la table de programmation dynamique ?

*Corrigé*

| |
|---|
| $(n+1) \times (m+1)$ |

**Q 4.2** [1 point] Donnez le code CAMLqui réalise l'initialisation de la table.

**Q 4.3** [2 points] Donnez le code CAMLqui réalise le remplissage de la table.

**Q 4.4** [1.5 point] Donnez le code CAMLqui extrait le résultat.

*Corrigé*

```
let carre_pd img =
  let min3 a b c = (min (min a b) c)
  in
  let n = Array.length img
  and m = (Array.length img.(0))
  in
  let t = Array.make_matrix (n+1) (m+1) 0
  in
  (* initialisation *)
  (* fait avec le array.make *)
  (* remplissage *)
  for i = 1 to n do
    for j = 1 to m do
      if img.(i-1).(j-1) = 1 then
        t.(i).(j) <- 1 + min3 t.(i-1).(j-1) t.(i).(j-1) t.(i-1).(j)
      else
        t.(i).(j) <- 0
    done
  done;
  (* resultat *)
  let imax = ref 1
  and jmax = ref 1
  in
  for i = 1 to n do
    for j = 1 to m do
      if t.(i).(j) > t.(!imax).(!jmax) then begin
        imax := i;
        jmax := j
      end;
    done;
  done;
  (!imax-t.(!imax).(!jmax),!jmax-t.(!imax).(!jmax),t.(!imax).(!jmax))
```

**Q 4.5** [1.5 point] Quel est le comportement asymptotique en temps ? Justifez clairement.

*Corrigé*

Le remplissage de chaque case de la table demande 1 comparaisons de couleur et on procède au remplissage de $n \times m$ cases. Idem pour l'extraction du résultat. On est donc en $\Theta(n \times m)$.