Programmation dynamique: corrigé

Exercice 1. *Carré de zéros dans une matrice binaire*. On considère une matrice de taille $n \times m$, constituée de zéros et de uns. On cherche à déterminer la taille maximale d'un carré dans la matrice, constitué uniquement de zéros. Par exemple, la matrice suivante possède un carré de zéros de taille 3×3 :

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Pour $0 \leq i < n$ et $0 \leq j < m$ on note $t_{i,j}$ la taille maximale d'un carré de zéro dans la matrice, dont le coin en bas à droite est indexé par (i, j) (on a donc notamment $t_{i,j} = 0$ si le coefficient en case (i, j) de la matrice est 1).

1. Relier $t_{i,j}$ à $t_{i-1,j}$, $t_{i,j-1}$ et $t_{i-1,j-1}$ pour $i, j \geq 1$.

2. En déduire une méthode efficace pour calculer la taille maximale d'un carré de zéros dans la matrice.

3. L'implémenter en Caml et donner sa complexité.

```
#a ;;
- : int array array =
 [|[|1; 0; 0; 1; 1; 0; 1; 1; 0|]; [|0; 1; 1; 1; 0; 1; 1; 0; 0|];
   [|0; 0; 0; 1; 0; 1; 0; 0; 0|]; [|0; 1; 1; 1; 0; 0; 0; 0; 0|];
   [|0; 0; 0; 0; 0; 1; 0; 0; 0|]; [|1; 1; 0; 0; 1; 0; 1; 1; 0|]|]
#taille_zeros a ;;
- : int = 3
```

Corrigé.

1. Notons $A = (a_{i,j})_{0 \leq i < n, 0 \leq j < m}$ la matrice. On a, pour $i, j \geq 1$:

$$t_{i,j} = \begin{cases} 0 & \text{si } a_{i,j} = 1 \\ 1 + \min(t_{i-1,j}, t_{i,j-1}, t_{i-1,j-1}) & \text{sinon.} \end{cases}$$

En effet, supposons $a_{i,j} = 0$ (sinon il n'y a rien à prouver) et notons $p = \min(t_{i-1,j}, t_{i,j-1}, t_{i-1,j-1}$.

- D'une part, l'union des 3 carrés de taille $p \times p$ dont le coin en bas à droite est situé en $(i-1, j)$, $(i, j-1)$ et $(i-1, j-1)$ forme avec la case (i, j) un carré de taille $(p+1) \times (p+1)$ rempli de zéros, donc $t_{i,j} \geq p + 1$.
- . D'autre part, on sait que le carré de taille $(t_{i,j} + 1) \times (t_{i,j} + 1)$ dont le coin en bas à droite est (i, j) n'est pas constitué uniquement de zéros. Comme $a_{i,j} = 0$, l'un des trois carrrés de taille $t_{i,j} \times t_{i,j}$ dont le coin en bas à droite est $(i-1, j)$, $(i, j-1)$ et $(i-1, j-1)$ n'est donc pas constitué uniquement de zéros, et donc $p \leq t_{i,j} - 1$.

Et la relation est prouvée.

2. Il suffit de calculer incrémentalement les $t_{i,j}$ dans une matrice de même taille que A (par exemple par indices i croissants et par indice j croissants à i fixé), et de trouver le maximum.

3. En calculant le maximum en parallèle :

```
let taille_zeros a=
  let n,m=Array.length a, Array.length a.(0) in
  let t=Array.make_matrix (n+1) (m+1) 0 and maxi=ref 0 in (* petit décalage pour pas s'embeter *)
  for i=1 to n do
    for j=1 to m do
      if a.(i-1).(j-1)=0 then t.(i).(j) <- 1+ min (min t.(i-1).(j) t.(i-1).(j-1)) t.(i).(j-1) ;
      maxi:= max !maxi t.(i).(j)
    done
  done ;
!maxi
;;
```

La complexité (en temps comme en espace) est $O(nm)$. *Remarque :* il est possible d'écrire un algorithme de complexité $O(\min(n, m))$ en mémoire seulement. Comment ?

Exercice 2. *Le problème de la portion de somme maximale.* On se donne un tableau `t` de taille n contenant des entiers, qui peuvent être positifs ou négatifs. Le but de l'exercice est de rechercher une portion du tableau, délimitée par deux indices $i \leq j$, telle que la somme `t.(i)` + `t.(i+1)` + $\cdots$ + `t.(j)` est maximale.

1. Résoudre le problème naïvement en testant toutes les possibilités : écrire une fonction `max_somme_naif t` retournant un couple d'indices (i, j) qui convient (on pourra écrire une fonction `somme_portion t i j`, et faire usage de références ensuite). Quelle est la complexité de cette approche ?

```
# let t_ex = [|-1; 2; 1; 4; -3; -5; 6; 2; -1; 6; 0; -1; -2; 2|] in max_somme_naif t_ex ;;
- : int * int = (6, 9)
```

2. Le reste de l'exercice est dévolu à la recherche d'une solution faisant usage de programmation dynamique. On note s_j une somme maximale de la forme `t.(i)` + `t.(i+1)` + $\cdots$ + `t.(j)`. Exprimer s_j en fonction de s_{j-1}.

3. En déduire un algorithme `max_somme_dyn t` permettant le calcul de tous les s_j, et renvoyant $s = \max_j s_j$. Quelle est sa complexité ?

4. Modifier l'algorithme pour qu'il retourne en plus un couple d'indice (i, j) qui convient, sans changer la complexité.

Corrigé.

1. On écrit d'abord une fonction calculant $\sum_{k=i}^{j} t.(k)$:

```
let somme_portion t i j =
  let s=ref 0 in
  for k=i to j do
    s:= !s + t.(k)
  done ;
  !s
;;
```

Puis :

```
let max_somme_naif t =
  let im, jm, sm = ref 0, ref 0, ref t.(0) in
  let n=Array.length t in
  for i=0 to n-1 do
    for j=i to n-1 do
      let s=somme_portion t i j in
      if s> !sm then (sm:=s ; im:=i; jm:=j)
    done ;
  done ;
  !im, !jm
;;
```

La complexité est $O(n^3)$: $O(n)$ pour `somme_portion`, que l'on appelle $O(n^2)$ fois.

2. $s_j = t.(j) + \max(s_{j-1}, 0)$. En effet :

- $s_j \geq t.(j)$ (il suffit de prendre la portion contenant seulement l'élément d'indice j), et $s_j \geq s_{j-1} + t.(j)$ (compléter une portion de poids s_{j-1} par $t.(j)$ donne une portion de poids $s_{j-1} + t.(j)$, d'où $s_j \geq t.(j) + \max(s_{j-1}, 0)$.
- Réciproquement, notons $i \leq j$ un indice tel que $\sum_{k=i}^{j} t.(k) = s_j$. Si $i = j$, alors $s_j = t.(j)$, et nécessairement $s_{j-1} \leq 0$, et il y a égalité dans l'inégalité précédente. Sinon, en retirant $t.(j)$ on obtient une portion de poids $s_j - t.(j) \geq 0$ terminant à l'indice $j - 1$, donc de poids au plus s_{j-1}, ainsi $s_j - t.(j) \leq s_{j-1}$ et on a encore $s_j = t.(j) + \max(s_{j-1}, 0)$.

3. Voici l'algorithme :

— En démarrant avec $s_0 = t.(0)$, calculer itérativement tous les s_j (complexité $O(n)$), en gardant l'indice j_m tel que s_{j_m} soit maximal ;

— En partant de $i = j_m$ par pas de -1, chercher i tel que $\sum_{k=i}^{j_m} t.(k) = s_{j_m}$ (temps $O(n)$ également).

— Renvoyer le couple (i, j_m) obtenu.

D'où la solution du problème en $O(n)$.

4. Voici le code :

```
let max_somme_dyn t=
  let n=Array.length t in
  let s=ref t.(0) and jm=ref 0 and sm=ref t.(0) in
  for j=1 to n-1 do
    s:= t.(j) + max !s 0 ;
    if !s > !sm then (sm:= !s ; jm:=j)
  done ;
  let i=ref !jm and s=ref t.( !jm) in
  while !s < !sm do
    decr i ;
    s:= !s + t.( !i)
  done ;
  !i, !jm
;;
```

Exercice 3. *Distance de Levenshtein.* On définit la distance de Levenshtein entre deux mots u et v comme le nombre minimal $d(u, v)$ d'opérations nécessaires pour obtenir v à partir de u, les opérations autorisées étant la suppression, l'insertion, ou la modification d'une lettre. Par exemple, *mathematique* est à distance 5 de *arithmetique*, comme on le voit par la suite de transformations :

mathematique → mathmatique → mathmetique → aathmetique → arthmetique → arithmetique

1. Montrer que d est bien une distance, c'est à dire qu'elle vérifie :
$d(u, v) = 0 \Leftrightarrow u = v \qquad \forall u, v,\ d(u, v) = d(v, u) \qquad$ et $\qquad \forall u, v, w,\ d(u, v) \leq d(u, w) + d(w, v)$.

2. Pour $0 \leq i \leq |u|$ et $0 \leq j \leq |v|$, on note $d_{i,j}$ la distance de Levenshtein entre les préfixes de u et de v de longueurs respectives i et j. Déterminer une expression de $d_{i,j}$ faisant intervenir $d_{i,j-1}$, $d_{i-1,j-1}$ et $d_{i-1,j}$.

3. En déduire un algorithme de calcul de la distance de Levenshtein.

```
# dist_levenshtein "mathematique" "arithmetique" ;;
- : int = 5
```

4. Modifier l'algorithme pour obtenir un *alignement* de séquences, c'est à dire une plus courte manière de passer de u à v. Coder effectivement une fonction `alignement u v` permettant de passer de `u` à `v`. (On utilisera `print_string` pour des affichages à l'écran).

```
# alignement "mathematique" "arithmetique" ;;
mathematique
mathemetique
mathmetique
athmetique
arthmetique
arithmetique
- : unit = ()
```

Corrigé.

1.
- $d(u, u) = 0$ pour tout mot u : il n'y a aucune opération pour passer de u à lui-même ! Si $d(u, v) = 0$, alors $u = v$.
- d est clairement symétrique.
- Si on effectue $d(u, w)$ opérations pour passer de u à w, puis $d(w, v)$ opérations pour passer de w à v, on a effectué $d(u, w) + d(w, v)$ opérations pour passer de u à v, ainsi, $d(u, v) \leq d(u, w) + d(w, v)$.

2. On note $u_0, u_1, \ldots, u_{|u|-1}$ les lettres de u, de même pour v. On a :

$$d_{i,j} = \begin{cases} i & \text{si } j = 0 \\ j & \text{si } i = 0 \\ d_{i-1,j-1} & \text{si } u_{i-1} = v_{j-1} \\ 1 + \min(d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}) & \text{sinon.} \end{cases}$$

En effet :

- les deux premiers cas sont évidents ;

- Supposons $u_{i-1} = v_{j-1}$. Étant donnée une chaîne d'opérations de longueur $d_{i,j}$ permettant de passer de $u_0 \cdots u_{i-1}$ à $v_0 \cdots v_{j-1}$, on obtient une chaîne d'opérations de longueur inférieure permettant de passer de $u_0 \cdots u_{i-2}$ à $v_0 \cdots v_{j-2}$ en supprimant les dernières lettres des mots apparaissant dans la chaîne et les éventuelles répétitions de mots identiques dans la chaîne obtenue, d'où $d_{i,j} \geq d_{i-1,j-1}$. La réciproque s'obtient en rajoutant u_{i-1} à la fin de tous les mots d'une chaîne d'opérations permettant de passer de $u_0 \cdots u_{i-2}$ à $u_0 \cdots u_{j-2}$.
- Si $u_{i-1} \neq v_{j-1}$, on montre de même que $d_{i,j} \leq 1 + \min(d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j})$. Par exemple, si le minimum est atteint pour $d_{i-1,j-1}$, il suffit d'effectuer la transformation $u_0 \cdots u_{i-1} \to u_0 \cdots u_{i-2}v_{i-1}$ suivie des transformations permettant de transformer $u_0 \cdots u_{i-2}$ et $v_0 \cdots v_{i-2}$ pour obtenir une chaîne de transformation de $u_0 \cdots u_{i-1}$ en $v_0 \cdots v_{i-1}$ de longueur $1 + d_{i-1,j-1} = 1 + \min(d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j})$ (les deux autres cas se traitent de manière similaire). Réciproquement, considérons une chaîne de transformations de longueur $d_{i,j}$ permettant de passer de $u_0 \cdots u_{i-1}$ à $v_0 \cdots v_{j-1}$. Considérons, en remontant depuis la fin, le premier moment où la lettre v_{j-1} apparaît à la fin des mots. Si la transformation est un ajout, on obtient en supprimant cette transformation et toutes les dernières v_{j-1} apparaissant dans les mots à partir de cette transformation une suite de transformations de longueur $d_{i,j} - 1$ permettant de passer de $u_0, \dots, u_{i-1}$ à $v_0, \dots, v_{i-2}$, d'où $d_{i,j-1} \leq d_{i,j} - 1$ dans ce cas, ainsi $\min(d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}) \leq d_{i,j} - 1$. On traite les deux autres cas possibles (suppression, modification) de manière similaire.

3. Voici :

```
let dist_levenshtein u v =
  let n, m = String.length u, String.length v in
  let d = Array.make_matrix (n+1) (m+1) 0 in
  for i=0 to n do
    d.(i).(0) <- i
  done ;
  for j=0 to m do
    d.(0).(j)<- j
  done ;
  for i=1 to n do
    for j=1 to m do
      if u.[i-1] = v.[j-1] then
        d.(i).(j) <- d.(i-1).(j-1)
      else
        d.(i).(j) <- 1 + min (min  d.(i-1).(j-1) d.(i).(j-1)) d.(i-1).(j)
    done
  done ;
  d.(n).(m)
;;
```

4. Voici une solution : c'est un peu long. En Ocaml, les chaînes sont désormais immuables, il faut utiliser le type `Bytes` (octets) pour avoir un objet mutable. L'expression `Bytes.set s i x` remplace le caractère `i` de `s` par `x`, et les fonctions `Bytes.of_string` et `Bytes.to_string` permettent la conversion entre les types `Bytes` et `String`.

```
let retire_lettre u i=
  (* retire la lette d'indice i de u *)
  String.sub u 0 i^String.sub u (i+1) (String.length u - i -1)
;;

let change_lettre u i c=
  (* change la lette d'indice i de u en c*)
  let b=Bytes.of_string u in Bytes.set b i c ; Bytes.to_string b
;;

let affiche u= print_string u ; print_string "\n" ;;

let alignement u v =
  let n, m=String.length u, String.length v in
  let d=Array.make_matrix (n+1) (m+1) 0 in
  for i=0 to n do
    d.(i).(0) <- i
  done ;
  for j=0 to m do
    d.(0).(j) <- j
  done ;
  for i=1 to n do
    for j=1 to m do
      if u.[i-1] = v.[j-1] then
```

```
        d.(i).(j) <- d.(i-1).(j-1)
      else
        d.(i).(j) <- 1 + min (min  d.(i-1).(j-1) d.(i).(j-1)) d.(i-1).(j)
    done
  done ;
  print_string "\n" ;
  let rec aux i j u v =
    if d.(i).(j) = 0 then affiche v
    else if i>0 && j>0 && u.[i-1] = v.[j-1] then aux (i-1) (j-1) u v
    else if i>0 && d.(i-1).(j) = d.(i).(j) - 1 then (affiche u ; aux (i-1) j (retire_lettre u (i-1)) v)
    else if d.(i).(j-1) = d.(i).(j) - 1 then (aux i (j-1) u (retire_lettre v (j-1)) ; affiche v)
    else (affiche u ; aux (i-1) (j-1) (change_lettre u (i-1) v.[j-1]) v)
  in aux n m u v
;;
```

L'idée est par contre assez simple : il suffit de créer un chemin entre u et v via le tableau d, en passant par une case telle que $d.(i).(j)$ soit égal à 0 (associé à une chaîne w). Le code affiche les changements de u à w (avant l'appel récursif à `aux`) et de w à v (après l'appel récursif).

Exercice 4. *Rendu de monnaie*. On se donne $v_0, \dots, v_{n-1}$ des valeurs de pièces de monnaie, avec $v_0 = 1$. On peut supposer les v_i croissantes. On se donne une somme d'argent S à décomposer en les v_i (il y a au moins une solution car $v_0 = 1$), et on cherche à minimiser le nombre de pièces impliquées. En clair, on cherche à résoudre le problème :

$$\text{Trouver } (\alpha_0, \dots, \alpha_{n-1}) \text{ tel que } S = \sum_{i=0}^{n-1} \alpha_i v_i \quad \text{et} \quad \sum_{i=0}^{n-1} \alpha_i \text{ minimal.}$$

Dans un système monétaire usuel, il suffit de procéder en rendant le plus possible de pièces v_{n-1}, puis de pièces v_{n-2}, etc... Ce n'est pas le cas par exemple pour $(v_0, v_1, v_2) = (1, 3, 4)$, où pour $S = 6$ il vaut mieux utiliser deux pièces 3 que trois pièces 1, 1 et 4. Chercher une méthode de résolution du problème avec une complexité $O(Sn)$.

Corrigé. Notons $N_{i,j}$ le nombre minimal de pièces nécessaires pour obtenir la somme j, avec seulement les pièces d'indice au plus i (inclus), pour $0 \leq i < n$ et $0 \leq j \leq S$. On a larelation suivante :

$$N_{i,j} = \begin{cases} j & \text{si } i = 0 \\ N_{i-1,j} & \text{si } v_i > j \\ \min(N_{i-1,j}, 1 + N_{i,j-v_i}) & \text{sinon.} \end{cases}$$

En effet, les deux premiers cas sont immédiats. Pour le troisième :

- $N_{i,j} \leq N_{i-1,j}$ est clair, de même que $N_{i,j} \leq 1 + N_{i,j-v.(i)}$ car on peut obtenir une décomposition de i en $1 + N_{i,j-v.(i)}$ pièces à partir d'une décomposition de $j - v.(i)$ à $N_{i,j-v.(i)}$ pièces en lui rajoutant la pièce de valeur $v.(i)$. Donc $N_{i,j} \leq \min(N_{i-1,j}, 1 + N_{i,j-v_i})$.
- Réciproquement, si on possède une décomposition de j en $N_{i,j}$ pièces, soit celle-ci ne contient pas la pièce de valeur $v.(j)$ et $N_{i-1,j} = N_{i,j}$ soit elle la contient au moins une fois, et l'enlever fournit une décomposition de $j - v.(i)$ à $N_{i,j} - 1$ pièces (et donc $N_{i,j-v.(i)} \leq N_{i,j} - 1$). Dans les deux cas $N_{i,j} \geq \min(N_{i-1,j}, 1 + N_{i,j-v_i})$, d'où la relation.

Le code ressemble ensuite beaucoup à des choses déja faites : on calcule les $N_{i,j}$ dans un premier temps, puis une décomposition optimale sous forme de listes (un tableau indiquant pour chaque valeur de pièces le nombre choisi était également possible).

```
let decomposition v s=
  let n=Array.length v in
  let m=Array.make_matrix n (s+1) 0 in
  for j=0 to s do
    m.(0).(j) <- j
  done ;
  for i=1 to n-1 do
    for j=1 to s do
      if v.(i) > j then
        m.(i).(j) <- m.(i-1).(j)
      else
        m.(i).(j) <- min m.(i-1).(j) (1+m.(i).(j-v.(i)))
    done
  done ;
```

```
  let rec aux i r = match r, i with
    | 0,_ -> []
    | _,0 -> 1::aux i (r-1)
    | _ when r < v.(i) || m.(i-1).(r)<1+m.(i).(r-v.(i)) -> aux (i-1) r
    | _ -> v.(i) :: aux i (r-v.(i))
  in aux (n-1) s
;;
```