Programmation dynamique et mémoïzation

C. Charignon

Table des matières

Première partie
Cours

1 Mémoïzation et programmation dynamique

1.1 Principe de la mémoïzation

Le principe de la mémoïzation est relativement simple : dans une situation ou un algorithme naïf conduit à recalculer plusieurs fois une même valeur, on va enregistrer toutes les valeurs déjà calculées. Ceci demande plus de mémoire mais permet un gain de temps.

N.B. Attention : dans ce genre de technique on va mélanger des concepts de programmation impérative (utilisation d'un tableau ou autre structure mutable pour enregistrer les valeurs déjà calculées) et récursive (la structure de programme de base est souvent récursive.)

En général, voici comment on procède pour mémoïzer une fonction :

- Créer un objet)tableau ou autre…) appelé à retenir chaque résultat calculé. J'appellerai souvent `cache` cet objet. NB : ce tableau devra être créé hors de la fonction récursive ! Sans quoi il serait réinitialisé à chaque appel récursif.
- *au début* de chaque appel récursif, on regarde si la valeur a déjà été calculée. Si oui, on renvoie immédiatement le résultat. Sinon, on procède aux mêmes calculs que dans la version naïve.
- *à la fin* du calcul, on n'oublie pas d'enregistrer le résultat obtenu dans le tableau avant de le renvoyer.

En pratique, il y a trois opérations à réaliser :

1. Transformer la fonction initiale en une fonction auxiliaire, et créer une fonction principale s'occupant de créer le tableau puis d'appeler la fonction auxiliaire.
2. Au début de la fonction auxiliaire, tester si la valeur a déjà été calculée, et si oui la renvoyer directement. Sinon effectuer le calcul de la fonction naïve.
3. En sortie de la fonction auxiliaire, enregistrer la valeur calculée.
   Remarque : On peut alléger la lecture en utilisant une fonction comme :

   ```
   let renvoie cache arg valeur=
       (* Met valeur en cache, puis la renvoie.*)
   cache.(arg)<- valeur;
   valeur
   ;;
   ```

   Cette fonction sera alors utilisée exactement comme le `return` de Python.

Remarque : Dans certains langages, il est possible d'automatiser la mémoïzation. C'est-à-dire d'écrire une fonction `mémoïzator` qui prend en entrée une fonction quelconque et renvoie la fonction mémoïzée. En Caml, c'est peu pratique à cause du typage statique.

1.2 Premier exemple : suite de Fibonacci

On rappelle que l'algorithme naïf :

```
let rec fibonaif n=
    match n with
        |0  -> 0
        |1  -> 1
        |_  -> fibonaif (n-1) + fibonaif (n-2)
;;
```

a une complexité exponentielle, car il recalcule un grand nombre de fois les mêmes valeurs.

Voici ce qu'obtient en mémoïzant :

```
let fibo_memo n =
  let cache = Array.make (n+1) (-1) in (* -1 n'est pas possible pour un nb de Fibonacci, ainsi
    ↪  on sera sûr qu'une case contenant -1 correspond à une valeur pas encore calculée.*)

  let rec aux k =
    (* Ici on reprend la fonction naïve, avec deux différences :
       - au début de l'appel réc : regarder si la valeur pour k a déjà été calculée.
```

```
      - à la fin de l'appel : mettre la valeur calculée dans le cache.
    *)
    if cache.(k) <> -1 then
      (* valeur déjà calculée *)
      cache.(k)
    else
      (* Sinon on fait le calcul comme dans la fonction naïve *)
      if k=0 then (cache.(0)<- 0; 0)
      else if k=1 then (cache.(1)<-1; 1)
      else(
        let res = aux (k-1) + aux (k-2) in
        cache.(k)<- res;
        res
      )

  in aux n
;;
```

On peut alléger un peu le code :

- Les cas de base peuvent être traités en remplissant à l'avance le cache ;
- On peut créer une fonction `renvoi` qui sicharge d'enregistrer la valeur dans le cache avant de la renvoyer.

```
let fibo_memo n =
  let cache = Array.make (n+1) (-1) in

  (* cas de base : *)
  cache.(0)<-0;
  cache.(1)<-1;

  let renvoie k res =
    (* Met res en cache dans la case k puis le renvoie.*)
    cache.(k) <- res;
    res
  in

  (* La fonction principale *)
  let rec aux k =
    if cache.(k) <> -1 then
      cache.(k)
    else
        renvoie k  (aux (k-1) + aux (k-2))

  in aux n
;;
```

1.3 Version impérative : programmation dynamique

Sur le cas particulier de la suite de Fibonacci, comme sur beaucoup d'autres, on peut simplifier la structure de l'algorithme. En effet, on constate que pour calculer un terme F_n, on a uniquement besoin des termes *précédents*. Par conséquent, il est possible de remplir le tableau de gauche à droite, par une simple boucle pour !

```
let fibo_dyna n =
  let cache = Array.make (n+1) (-1) in
  cache.(0)<-0;
  cache.(1)<-1;

  (* On va maintenant remplir toutes les cases de cache. *)
  (* Dans quel ordre ? *)
  (* Lorsqu'on veut remplir la case cache.(k), on va utiliser cache.(k-1) + cache.(k-2).
     Il faut donc que les cases cache.(k-1) et cache.(k-2) aient déjà été remplies. *)
  (* -> Remplissons cache de gauche à droite. *)
```

```
  for k = 2 to n do
    cache.(k) <- cache.(k-1) + cache.(k-2)
  done;
  cache.(n)
;;
```

On obtient un algorithme purement impératif, plus simple. Il y a deux bémols :

- ceci n'est possible que dans les situations pas trop compliquées où on peut deviner l'ordre dans lequel faire les calculs ;
- ceci nécessite un peu de réflexion, et donc comporte plus de risques d'erreur. Au contraire, la méthode générale de mémoïzation fonctionne toujours de la même manière.

Au niveau de la complexité on a généralement le même ordre de grandeur. Une boucle peut être plus rapide à gérer par l'ordinateur que des appels récursifs. En contre-partie, il se peut que l'on calcule des termes inutile si on décide de remplir tout le tableau.

Le type de programmation utilisé pour cette version impérative s'appelle la *programmation dynamique*.

Remarque : Vocabulaire : la méthode impérative où on remplit les cases du tableau les unes après les autres s'appelle la méthode « bottom-up ». Quand on colle à la fonction récursive initiale, c'est « up-bottom ». En effet, l'appel initial est pour la valeur finale à calculer.

1.4 Deuxième exemple : coefficients binomiaux

Appliquez les méthode de mémoïzation puis de programmation dynamique à l'exemple du calcul de coefficient binomiaux par la formule de Pascal.

On utilisera comme cache un tableau à deux dimensions.

Voici une version naïve inefficace.

```
let rec cbNaif p n  =
  (* Renvoie p parmi n *)
  if p=0 then 1
  else if p>n then 0
  else cbNaif (p-1) (n-1) + cbNaif p (n-1)
;;
```

On peut la mémoïzer ainsi :

```
let cbMemo p n =

  let cache = Array.make_matrix (n+1) (p+1) (-1) in (* Rema : ce tableau n'est autre que le triangle de Pascal *)
  (* cache.(j).(i) contiendra i parmi j. C'est très perturbant d'utiliser i comme indice de colonne et j comme indice de ligne, mais la coutume est de mettre le premier argument comme indice de colonne et le second en ligne quand on dessine le triangle de Pascal... *)

  let renvoi i j res =
    (* À utiliser lorsque aux i j renvoie res. (res = i parmi j)*)
    cache.(j).(i) <- res;
    res
  in

  let rec aux i j =
    (* Renvoie i parmi j et le met dans le cache *)
    if cache.(j).(i) <> -1 then cache.(j).(i)
    else if i=0 then renvoi i j 1
    else if i>j then renvoi i j 0
    else renvoi i j
                (aux (i-1) (j-1) + aux i (j-1))
  in
```

```

  aux p n
;;
```

Et la « dynamiser » ainsi :

```
let cbMemo p n =

  let cache = Array.make_matrix (n+1) (p+1) (-1) in (* Rema : ce tableau n'est autre que le triangle de Pascal *)
  (* cache.(j).(i) contiendra i parmi j. C'est très perturbant d'utiliser i comme indice de colonne et j comme indice de ligne, mais la coutume est de mettre le premier argument comme indice de colonne et le second en ligne quand on dessine le triangle de Pascal... *)

  let renvoi i j res =
    (* À utiliser lorsque aux i j renvoie res. (res = i parmi j)*)
    cache.(j).(i) <- res;
    res
  in

  let rec aux i j =
    (* Renvoie i parmi j et le met dans le cache *)
    if cache.(j).(i) <> -1 then cache.(j).(i)
    else if i=0 then renvoi i j 1
    else if i>j then renvoi i j 0
    else renvoi i j
            (aux (i-1) (j-1) + aux i (j-1))
  in

  aux p n
;;
```

2 Exemples plus compliqués

2.1 Méthode générale

La programmation dynamique s'emploie souvent dans des problèmes d'optimisation (c'est-à-dire de calcul de maximum ou de minimum), dont voici quelques exemples classiques :

- Calculer le nombre minimal de pièces à utiliser pour payer une certaine somme (problème du rendu de monnaie) ;
- Calculer la manière d'agencer des calculs matriciels pour minimiser les calculs ;
- Calculer le nombre maximum d'objets qu'on peut mettre dans un récipient de taille fixée (problème du sac à dos) [1]
- Répartir plusieurs tâches entre plusieurs machines pour minimiser le temps total d'exécution

Le principe est le suivant :

1. On trouve une relation entre la solution du problème et les solutions de problèmes plus petits, ce qui permet d'écrire une fonction récursive pour calculer le minimum ou le maximum cherché.
2. On optimise cette fonction par mémoïzation.
3. On peut souvent passer à une version impérative consistant à remplir une matrice dans un certain ordre.
4. En modifiant la fonction, on peut lui faire renvoyer non seulement le minimum ou maximum cherché, mais également la solution permettant d'obtenir ce minimum ou maximum (parfois appelé l'argmax).

2.2 Exemple : distance d'édition

Appliquons ce principe sur un exemple classique : la distance d'édition (ou de Levenstein).

On cherche à mesurer la distance entre deux mots (utilisé typiquement dans les correcteurs orthographiques, mais aussi sur des textes entiers pour une recherche de plagiat par exemple). On fixe un alphabet Σ. L'ensemble des mots formés à partir des lettres de Σ est noté Σ^*. Le mot vide (contenant 0 lettre) est noté ϵ. Pour tout $u \in \Sigma^*$, son nombre de lettres sera noté $|u|$.

On définit trois opérations élémentaires :

- la substitution : remplacer une lettre par une autre ;
- l'insertion : insérer une nouvelle lettre ;
- la suppression : supprimer une lettre.

La distance d'édition entre deux mots u et v est le nombre minimal de telles opérations élémentaires permettant de transformer u en v. On la notera $d(u, v)$.

Proposition 2.1. *La fonction d ainsi définie est une distance sur Σ^*, ce qui signifie :*

- $\forall (u, v) \in (\Sigma^*)^2$, $d(u, v) = d(v, u)$ *(symétrie) ;*
- $\forall (u, v) \in (\Sigma^*)^2$, $d(u, v) = 0 \Leftrightarrow u = v$ *(caractère séparé) ;*
- $\forall (u, v, w) \in (\Sigma^*)^3$, $d(u, w) \leqslant d(u, v) + d(v, w)$ *(inégalité triangulaire).*

Remarque : On a immédiatement $d(u, v) \leqslant |u| + |v|$: on peut toujours supprimer toutes les lettres de u (donc $|u|$ suppressions) puis insérer toutes les lettres de v (soit $|v|$ insertions).

Et même plus précis, en supposant que u est le plus petit des deux mots, $d(u, v) \leqslant |u| + |v| - |u| = |v|$.

On peut démontrer que :

- Pour tout mot u, $d(u, \epsilon) = |u|$: il faut faire $|u|$ suppressions pour passer de u à ϵ.
- Pour tout mot u, $d(\epsilon, u) = |u|$: il faut faire $|u|$ insertions.
- Pour tous mots u, v et lettre a, $d(ua, va) = d(u, v)$: il suffit de transformer u en v.
- Pour tous mots u, v et lettres a, b distinctes, $d(ua, vb) = 1 + \min\big(d(ua, v), d(u, vb), d(u, v)\big)$. En effet, il existe un chemin minimal de ua à vb qui se termine par ajouter b à v ou supprimer a dans ua, ou transformer a en b.

1. Dans le dernier cas, préciser quelle est l'opération élémentaire cachée derrière le "1+", selon que le minimum est $d(ua, v), d(u, vb)$, ou $d(u, v)$.

1. variantes plus "sérieuse" : optimisation le transport de marchandise en bateau ou avion, minimiser les chutes lors de la découpe d'un matériaux....

2. Écrire une fonction récursive calculant la distance entre deux mots.

   Il pourra être plus simple d'utiliser une fonction auxiliaire `aux` telle que pour tout i, j `aux i j` calcule $d(u[:i], v[:j])$ en notation python, c'est-à-dire `aux i j` calcule la distance entre le mot formé des i premières lettres de u et celui formé des j premières lettres de v.

3. Pour tout $n \in \mathbb{N}$, on note C_n le nombre de comparaisons lorsque $|u| + |v| = n$. Montrer que $2^n = O(C_n)$. Ainsi la complexité est exponentielle.

4. Écrire une version memoïzée de la fonction précédente. (Un simple tableau suffira.)

5. Faire tourner l'algorithme à la main sur un papier sur un exemple simple. Dans quel ordre simple peut-on remplir les cases du tableau ? En déduire une version simplifiée de l'algorithme précédent.

   Quelle est la nouvelle complexité ?

6. Quel est l'espace mémoire utilisé ? Optimiser votre algorithme pour que sa complexité spatiale soit $O(\max(|u|, |v|))$.

7. Enfin, modifier le programme pour qu'il renvoie, en plus de la distance d'édition, la suite de transformations effectuées pour passer de u à v.

3 Utilisation d'un dictionnaire

3.1 Introduction : suites de Syracuse

Soit $p \in \mathbb{N}^*$. La suite de Syracuse de premier terme p est la suite u^p telle que :

$$u_0^p = p \text{ et } \forall n \in \mathbb{N}, u_{n+1} = \left| \begin{array}{ll} \frac{u_n}{2} & \text{si n est pair} \\ 3u_n + 1 & \text{sinon} \end{array} \right. .$$

On écrit facilement un programme pour calculer les termes de la suite de Syracuse :

```
let terme_suivant x =
  (* x est un terme d'une suite de Syracuse; ceci renvoie le terme suivant.*)
  if x mod 2 = 0 then
    x/2
  else
    3*x+1
;;
```

```
let rec syracuse n p =
  (* Renvoie u_n^p, càd le terme n de la suite de Syracuse avec terme initial p. *)
  if n=0 then p
  else
    terme_suivant (syracuse (n-1) p)
;;
```

Une conjecture classique affirme que quelque soit $p \in \mathbb{N}^*$, la suite u^p finit par retomber sur 1 (et à partir de là elle va boucler : 1,4,2,1,4,2,...)

Définition 3.1. *Pour tout $p \in \mathbb{N}^*$, on note $syr(p)$ le plus petit entier tel que $u^p_{syr(p)} = 1$. On l'appelle le temps de vol de u^p.*

On calcule facilement le temps de vol :

```
  …
let rec temps_de_vol p =
  if p=4 then 0
  else
   1 + temps_de_vol (terme_suivant p)
;;
```

Imaginons maintenant que nous voulions calculer plusieurs valeurs. Par exemple si nous cherchons pour $pmax \in \mathbb{N}^*$ le maximum de $\text{syr}(1), ..., \text{syr}(pamx)$.

Une version naïve est la suivante :

```
 (* Ecrivons une fonction pour calculer le temps de vol max d'une suite de Syracuse *)
let rec temps_de_vol_max pmax=
    (* temps de vol max pour p entre 1 et pmax *)
    if pmax=1 then 1
    else
      max (temps_de_vol pmax)
  (temps_de_vol_max (pmax-1))
;;

```

Là il serait judicieux d'enregistrer les valeurs précédemment calculées. Mais la suite de Syracuse étant complètement imprévisible, on ne sait pas à l'avances de quelles valeurs on aura besoin ! Par exemple on a vu que $u_1^3 = 10$, donc pour calculer $\text{syr}(3)$ on passe par $\text{syr}(10)$.

Dans cette situation, l'utilisation d'un tableau est malcommode car on ne peut même par savoir combien de cases prévoir au moment de créer le tableau. Éventuellement un tableau redimensionnable, mais il reste le défaut qu'il y a aura de nombreuses cases créées inutilement.

La structure la mieux adaptée semble ici le dictionnaire.

```
let temps_de_vol_max_memo pmax =
    let cache = Hashtbl.create pmax in

    let renvoie p t=
        Hashtbl.add cache p t;
        t
    in

    let rec temps_de_vol p=
    try
        Hashtbl.find cache p
    with
      |Not_found -> (* pas de valeur associée à p dans le cache *)
        if p=4 then renvoie p 0
        else renvoie p (1+ temps_de_vol (termeSuivant p))
    in

    let rec temps_de_vol_max pmax=
    (* temps de vol max pour p entre 1 et pmax *)
    if pmax=1 then 1
    else
    max (temps_de_vol pmax)
       (temps_de_vol_max (pmax-1))
  in

  temps_de_vol_max pmax;;



temps_de_vol_max_memo 9;;

let debut=Sys.time() in
let res= temps_de_vol_max 200000 in
res, Sys.time() -. debut;;

let debut=Sys.time() in
let res= temps_de_vol_max_memo 200000 in
res, Sys.time() -. debut;;
```

3.2 Mémoïzation systématique

N'importe quelle fonction peut être mémoïzée au moyen d'un dictionnaire, et ce de manière quasi automatique. Pour mémoïzer une fonction f quelconque :

```
let f_memo arg =

    let cache = Hashtbl.create 42 in

    let renvoie cache cle res=
        Hashtbl.add cle res;
        res
    in


    let rec aux arg =
    if Hashtbl.mem cache cle then
        Hashtbl.find cache cle
    else
        (* Recopier le code de f, en passant par la fonction "renvoie" pour chaque renvoi. *).
```

Deuxième partie

Exercices

Exercices : programmation dynamique

1 Programmation dynamique

Exercice 1. $S(n, k)$ = nombre de partitions de $[\![0, n[\![$ en k parties.
$S(n, k) = S(n-1, k-1) + kS(n-1, k)$.
Naïf puis dynamique.

Exercice 2. ** Appel (E3A 2017)
Dans une classe de n élèves, les élèves sont numérotés de 0 à $n-1$. Un professeur souhaite faire l'appel, c'est à dire déterminer quels élèves sont absents.

Une salle de classe est décrite par un tableau à n entrées. Si `tab` est un tel tableau et `i` un entier de $[\![0, n[\![$, alors `tab.(i)` donne le numéro de l'élève assis à la place i, ou -1 si cette place est vide.

1. Écrire une fonction `asseoir : int list` $\rightarrow$ `int` $\rightarrow$ `int vect`, qui prend en argument une liste non vide d'entiers distincts et un entier n, et renvoie un tableau représentant une salle de classe pour n élèves où chaque élève de la liste à été assis à la place numérotée par son propre numéro. Les entiers supérieurs ou égaux à n seront ignorés.
2. En déduire une fonction `absent2 : int list` $\rightarrow$ `int` $\rightarrow$ `int list` qui étant donné une liste non vide d'entiers distincts et un entier n, renvoie la liste des entiers de $[0; n-1]$ qui n'y sont pas. Les entiers supérieurs ou égaux à n seront ignorés.
3. En notant k la longueur de la liste donnée en argument, quelle est la complexité en nombre de lectures et d'écritures dans un tableau (en fonction de n et k) de la fonction précédente ?

Exercice 3. ** ! Parenthésage optimal pour un produit de matrices
Soit $n \in \mathbb{N}$ et $A_1, \dots, A_n$ n matrices de formats tels que le produit $A_1 \times \cdots \times A_n$ soit bien défini. On rappelle que le produit matriciel est associatif : les parenthèses peuvent être placées comme on le souhaite dans ce produit.

On désire calculer comment placer les parenthèses pour minimiser le nombre de multiplications à effectuer. On notera pour tout $i \in [\![1, n]\!]$, l_i et c_i le nombre de lignes et colonnes de A_i.

1. Quelle est la condition sur les c_i, l_i pour que le produit soit bien défini ?
   En pratique, les fonctions à suivre prendront en entrée le vecteur `[|c0; ...; cn|]` tel que pour tout i où cela a du sens, c_i est le nombre de lignes de la $(i+1)$-ème matrice et aussi le nombre de colonnes de la i-ème.
2. Soit $i \in [\![1, n]\!]$. Quel est le nombre de multiplications scalaires pour calculer $A_i \times A_{i+1}$?
3. Plus généralement, soit $(d, k, f) \in [\![1, n]\!]^3$ tel que $d \leqslant k \leqslant f$. Quel est le nombre de multiplications scalaires pour calculer la multiplication matricielle $\Big(A_d \dot{A}_k)\Big) \times \Big(A_{k+1} \dots A_f\Big)$?
4. Soit $(d, f) \in [\![1, n]\!]^2$ tel que $d \leq f$. Pour calculer le parenthésage optimal pour $(A_d \times \cdots \times A_f)$, le principe est d'essayer pour tout $k \in [\![d, f-1]\!]$ de faire la k-ème multiplication en dernier (c'est-à-dire un parenthésage de type $(A_d \times \dots \times A_k) \times (A_{k+1} \times \dots \times A_f))$ et de garder le minimum.
   Ainsi, si on note pour tout $(d, f) \in [\![1, n]\!]^2$ tel que $d \leqslant f$, $N_{d,f}$ le nombre minimal de multiplications scalaires pour calculer $A_d \dots A_f$. Écrire la formule basée sur le principe ci-dessus permettant d'exprimer $N_{d,f}$.
5. Écrire une fonction récursive basée sur ce principe.
6. Démontrer que votre fonction termine.
7. Montrer que sa complexité est exponentielle.
8. Mémoïzer cette fonction.
9. écrire une version impérative de cette fonction.
   Quelle est sa complexité ?
10. Écrire une version de cette fonction qui renvoie la liste des produits à effectuer, dans le bon ordre, pour que le calcul du produit matriciel soit optimal. Pour simplifier la programmation, on mettra la dernière multiplication à effectuer en tête de liste.

Exercice 4. * Stratégie gagnante pour un jeu de dépilage**[2]
On considère une pile d'entiers positifs `p` de longueur n et deux joueurs A et B.

- Les deux joueurs jouent l'un après l'autre. Chacun doit dépiler un certain nombre d'éléments de la pile, celui qui ne peut plus (car la pile est vide) perd.

2. Variante du jeu de Nim

- La règle est la suivante : à son tour un joueur a le choix entre :
  - ◇ dépiler un élément
  - ◇ dépiler k éléments, où k est l'entier actuellement au sommet de la pile. Opération autorisée uniquement s'il reste au moins k éléments dans la pile bien sûr.

A joue en premier. Le but est d'écrire une fonction qui calcule s'il existe une stratégie gagnante pour A.

On pourra maintenir un tableau de booléens `G` tel que pour tout i, `G.(i)` indique s'il est possible au joueur à qui c'est le tour de gagner lorsqu'il reste i éléments dans la pile.

Écrire une formule de récurrence permettant de remplir `G` (tout simplement de la case 0 à la case n), et programmer une fonction répondant au problème.

Exercice 5. ** Plus longue sous-séquence strictement croissante**

Écrire une fonction qui prend en entrée un tableau d'entiers `t` et qui renvoie la longueur de la plus longue sous-suite d'éléments de `t` strictement croissante.

Application : L'avenue des flots bleues est perpendiculaire à la plage. Le long de cette avenues s'élèvent des immeubles dont le nombre d'étages est 2,3,4,1,2,5,10,11,9,8. Quels immeubles faut-il raser pour que les immeubles restant aient tous vue sur la mer ?

Exercice 6. * Alignement de séquences génomiques**

Un génome est un mot formé à l'aide des lettres 'A', 'T', 'G', 'C'. Voici une méthode fréquemment utilisée pour estimer la différence entre deux génomes, très proche du calcul de la distance d'édition.

On introduit une nouvelle lettre, '-'.

Soient deux génomes u et v. Un « alignement » de u et v est une matrice de deux lignes dont la première contient u dans lequel sont éventuellement insérés un ou plusieurs '-', et la seconde ligne est v, dans lequel sont également éventuellement insérés un ou plusieurs '-'. Aucune colonne ne peut contenir deux '-'.

Par exemple $\begin{vmatrix} \text{A} & \text{G} & \text{-} & \text{A} & \text{-} & \text{-} \\ \text{A} & \text{-} & \text{G} & \text{C} & \text{T} & \text{A} \end{vmatrix}$ est un alignement entre "AGA" et "AGCTA".

Lorsque les deux lettres de la colonne sont identique, on dit qu'il y a appariement, lorsqu'il y a un '-' on dit qu'il y a une délétion, et lorsqu'il y a deux lettres différentes, on dit qu'il y a mésappariement.

On définit le « score » d'un alignement ainsi : un appariement vaut 4, une délétion ou un mésappariement vaut -1. Pour tout $(x, y) \in \{A, G, C, T, -\}^2$, on notera $\delta(x, y)$ le score entre les deux lettres x et y.

Le score total de l'alignement est obtenu en sommant le score des deux lettres de chaque colonne. Par exemple, le score de l'alignement ci-dessus est -1.

Le but est de déterminer l'alignement entre deux mots u et v qui réalise le score total maximal.

On notera pour tout $(i, j) \in [\![0, |u|]\!] \times [\![0, |v|]\!]$, $s_{i,j}$ le score maximal entre les sous-mots $u_1...u_i$ et $v_1...v_j$. On convient que $s_{0,0} = 0$.

1. Programmer la fonction δ.
2. Soit $i \in [\![1, |u|]\!]$, que vaut $s_{i,0}$? De même, soit $j \in [\![1, |v|]\!]$, que vaut $s_{j,0}$?
3. Justifier que $(i, j) \in [\![1, |u|]\!] \times [\![1, |v|]\!]$,
$$s_{i,j} = \max\left\{s_{i-1,j} + \delta(u_i, -), \quad s_{i,j-1} + \delta(-, v_j), \quad s_{i-1,j-1} + \delta(u_i, v_j)\right\}.$$
Dans quel cas est atteint chacun des 3 nombres dans le calcul du maximum ?
4. En déduire une fonction pour calculer si, j.

Quelques indications

7 Vérifier que $\forall n \in \mathbb{N}$, $C_n \geq 2C_{n-1}$.

9 remplir le tableau de base en haut.

5 Maintenir un tableau `t` tel que pour tout `k`, `t.(k)` est le dernier élément d'une plus longue sous-suite croissante de longueur de k.

Quelques solutions

2

1 Pour tout $i \in [\![1, n-1]\!]$, il faut que $c_i = l_{i+1}$.

2 Le produit $A_i \times A_{i+1}$ coûte $c_{i-1} \times c_i \times c_{i+1}$ multiplications entre coefficients de la matrice.

3 $c_{d-1} \times c_k \times c_f$

4

$$\forall (d, f) \in [\![1, n]\!]^2 \text{ tq } d < f,\ N_{d,f} = \min_{k \in [\![d, f-1]\!]} N_{d,k} + N_{k+1,f} + c_{d-1} c_k c_f$$

4 On trouve $G.(0) = faux$ et pour tout $i \in [\![1, n]\!]$, $G.(i) = \text{ non } (G.(i-1)) \text{ ou } \text{ non } (G.(i - p.(i)))$ si $p.(i) \leqslant i$ et $G.(i) = \text{ non } (G.(i-1))$ sinon.

5

6