TD 4 - PROGRAMMATION DYNAMIQUE

Exercice 1

1. Le nombre de sous-problèmes est card $[\![1,\ n]\!] = n$.
2. Le nombre de sous-problèmes est card$([\![1,\ n]\!] \times [\![1,\ m]\!]) = nm$.
3. Le nombre de sous-problèmes est card$\{(i,\ j)\ :\ 1 \leqslant i \leqslant j \leqslant n\} = \sum_{j=1}^{n} \underbrace{\text{card } [\![1,\ j]\!]}_{=j} = \frac{n(n+1)}{2}$.

Exercice 2

On note pour $n, p \in \mathbb{N}$: $\mathscr{N}(n,p)$ le nombre de chemins de A à B dans un quadrillage de taille $n \times p$.

- Si $n = 0$, le quadrillage se limite à une ligne horizontale, donc on ne peut se déplacer que vers la droite : il n'y a qu'un seul chemin pour toute valeur de p.

$$\forall p \in \mathbb{N},\ \mathscr{N}(0,p) = 1.$$

- De même on a

$$\forall n \in \mathbb{N}, \mathscr{N}(n,0) = 1.$$

- Le problème est fondamentalement récursif . Si $n, p \in \mathbb{N}^*$, alors depuis le point A on a le choix entre trois possibilités : aller vers le haut, vers la droite ou en diagonale. Chacun de ces choix nous amène à un nouveau point A_1 qui délimite avec B un quadrillage plus petit que celui de départ, dans lequel on devra poursuivre le dénombrement des chemins de A_1 à B. On a

$$(\mathscr{R})\ \ \forall n, p \in \mathbb{N}^*,\ \mathscr{N}(n,p) = \underbrace{\mathscr{N}(n-1,p)}_{\text{vers le haut}} + \underbrace{\mathscr{N}(n,p-1)}_{\text{vers la droite}} + \underbrace{\mathscr{N}(n-1,p-1)}_{\text{en diagonal}}.$$

Si on applique naïvement l'équation de récursion pour résoudre le problème, on va avoir une complexité qui vérifie l'équation

$$(\mathscr{R}')\ \ \forall n, p \in \mathbb{N}^*,\ T(n,p) = T(n-1,p) + T(n,p-1) + T(n-1,p-1) + \Theta(1)$$

donc en $\Theta(\mathscr{N}(n,p))$. On peut montrer que $\mathscr{N}(n,p) \geqslant \binom{n}{p}$ pour $n \geqslant p$ car d'après $(\mathscr{R})$:

$$\forall n > p \in \mathbb{N}^*,\ \mathscr{N}(n,p) \geqslant \mathscr{N}(n-1,p) + \mathscr{N}(n-1,p-1).$$

Remarque

Géométriquement, toujours pour $n \geqslant p$, cela s'interprète comme la dénombrement du sous-ensemble des chemins composés de déplacements verticaux et en diagonal. On voit que le nombre de tels chemins est $\binom{n}{p}$ car il faut choisir où l'on fait les p déplacements

diagonaux nécessaires pour traverser le quadrillage dans toute sa longueurs, les $n-p$ autres déplacements étant nécessairement verticaux.

Avec Stirling, on peut montrer que $\binom{2n}{n} = \frac{(2n)!}{(n!)^2} \underset{n\to\infty}{\sim} \frac{\sqrt{2\pi 2n}(2n/\mathrm{e})^{2n}}{(\sqrt{2\pi n}(n/\mathrm{e})^n)^2} = \Omega(3^n)$, donc on peut avoir une complexité plus qu'exponentielle.

En réalité, on réalise plusieurs fois les mêmes appels récursifs. Les sous-problèmes que l'on a exhibé pour résoudre celui du calcul de $\mathcal{N}(n,p)$ ne sont pas indépendants. Il faut donc procéder autrement.

On effectue les appels récursifs en se souvenant des résultats dans une matrice, pour n'avoir à les calculer qu'une seule fois : c'est la mémoïsation. On crée une matrice `resultats` qui devra contenir à la fin de l'exécution $(\mathcal{N}(i,j))_{0\leqslant i\leqslant n,\ 0\leqslant j\leqslant p}$.

Programme

```
let nb_chemins n p =
  let resultats = make_matrix (n + 1) (p + 1) 0 in
  for i = 0 to n do
    resultats.(i).(0) <- 1
  done ;                    (* tableau pour la memoisation *)
  for j = 0 to p do
    resultats.(0).(j) <- 1
  done ;
  let rec nb_chemins_aux x y =
    match (x, y) with
    | (_, 0) -> resultats.(x).(y) (* cas de base *)
    | (0, _) -> resultats.(x).(y)
    | (_, _) when resultats.(x).(y) > 0 ->
        resultats.(x).(y) (* quantite deja calculee avant *)
    | (_, _) ->
        let valeur =
          nb_chemins_aux (x - 1) y +
          nb_chemins_aux x (y - 1) +
          nb_chemins_aux (x - 1) (y - 1)
        in
        resultats.(x).(y) <- valeur ;
        valeur
  in
  nb_chemins_aux n p
;;

#nb_chemins 3 3 ;;
- : int = 63
#nb_chemins 5 3 ;;
- : int = 231
#nb_chemins 10 10 ;;
- : int = 8097453
```

La complexité temporelle et spatiale est en $\Theta(np)$ puisqu'il faut remplir la matrice pour avoir le résultat voulu.

Exercice 3

On note pour $(i,\ j) \in [\![0,\ n-1]\!] \times [\![0,\ i]\!]$, $S_{i,j}$ la somme maximale que l'on peut obtenir en

parcourant la figure à partir du sommet $(i,\ j)$ de la matrice représentant le triangle.

- $\forall j \in [\![0,\ n-1]\!],\ S_{n-1,\ j} =$ `triangle.(i).(j)`.
- $\forall (i,\ j) \in [\![0,\ n-2]\!] \times [\![0,\ j]\!],\ S_{i,\ j} =$ `triangle.(i).(j)` $+ \max(S_{i+1,\ j},\ S_{i+1,\ j+1})$ car l'on a que des termes positifs, donc pour obtenir le chemin correspondant à la somme maximale dans un triangle, il faut choisir de parcourir le sous-triangle adjacent dans lequel on a un chemin correspondant à la plus grande somme pour les deux sous-triangles.

Là encore, on a un problème récursif pour lequel les appels récursifs ne sont pas indépendants, examiner la figure suffit à s'en convaincre. On pourrait comme pour l'exercice 2 procéder par mémoïsation. Cette fois-ci, on va plutôt construire la famille S sous forme d'un tableau, de "bas" en "haut".

Programme

```
let somme_max triangle =
  let n = vect_length triangle in
  let s = make_matrix n n 0 in
  for j = 0 to n - 1 do
    s.(n - 1).(j) <- triangle.(n - 1).(j)
  done ;
  for i = n - 2 downto 0 do
    for j = 0 to i do
      s.(i).(j) <- triangle.(i).(j) +
                   (max s.(i + 1).(j) s.(i + 1).(j + 1))
    done
  done ;
  s.(0).(0)
;;
```

La complexité temporelle et spatiale est en $\Theta(n^2)$.

Exercice 4

Le problème posé est équivalent à la recherche du maximum dans la famille $(c_{i,\ j})_{1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant p}$, où $c_{i,\ j}$ est le côté du plus grand carré de 1 de la matrice dont le coefficient $(i,\ j)$ est le sommet en bas à droite. Le tout est de savoir comment l'on peut construire cette famille algorithmiquement.

- $\forall i \in [\![0,\ n-1]\!],\ c_{i,\ 0} = M_{i,\ 0}$.
- $\forall j \in [\![0,\ p-1]\!],\ c_{0,\ j} = M_{0,\ j}$.
- $\forall (i,\ j) \in [\![1,\ n-1]\!] \times [\![1,\ p-1]\!],\ c_{i,\ j} = \begin{cases} 0 & \text{si } M_{i,\ j} = 0 \\ 1 + \min(c_{i-1,\ j},\ c_{i,\ j-1},\ c_{i-1,\ j-1}) & \text{sinon} \end{cases}$.

Le problème est récursif, et on voit que les sous-calculs que l'on doit faire pour calculer $c_{i,\ j}$ ne sont pas indépendant. On va, pour éviter des calculs trop coûteux, procéder itérativement en construisant $(c_{i,\ j})_{1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant p}$ de bas en haut, de la gauche vers la droite. Intelligemment, on va aussi rechercher le maximum de la famille en même temps qu'on la construit, grâce à une variable référencée.

On réalise l'algorithme en langage CAML.

Programme

```
let plus_grand_carre matrice =
  let n = vect_length matrice
  and p = vect_length matrice.(0) in
  let detect_1 = make_matrix n p 0 in
  let cote_max = ref 0 in
  for i = 0 to n - 1 do
    if matrice.(i).(0) = 1 then begin
      detect_1.(i).(0) <- 1 ;
      cote_max := 1
    end
  done ;
  for j = 0 to p - 1 do
    if matrice.(0).(j) = 1 then begin
      detect_1.(0).(j) <- 1 ;
      cote_max := 1
    end
  done ;
  for i = 1 to n - 1 do
    for j = 1 to p - 1 do
      if matrice.(i).(j) = 1 then begin
        let cote_carre =
          1 +
          min (min detect_1.(i - 1).(j) detect_1.(i).(j - 1))
              detect_1.(i - 1).(j - 1))
        in
        detect_1.(i).(j) <- cote_carre ;
        cote_max := max !cote_max cote_carre
      end
    done
  done ;
  !cote_max
;;

let matrice =
  [| [| 1 ; 0 ; 0 ; 1 ; 1 ; 1 |] ;
     [| 0 ; 1 ; 1 ; 1 ; 1 ; 0 |] ;
     [| 0 ; 0 ; 1 ; 1 ; 1 ; 0 |] ;
     [| 1 ; 0 ; 1 ; 1 ; 1 ; 1 |] ;
     [| 1 ; 1 ; 1 ; 0 ; 0 ; 1 |] ;
     [| 0 ; 1 ; 1 ; 1 ; 1 ; 1 |] |]
;;

#plus_grand_carre matrice ;;
- : int = 3
```

Comme demandé, la complexité temporelle (et spatiale) est en $\Theta(np)$.

Exercice 5

Soient $E = (n_1, \ldots, n_k) \in \mathbb{N}^{*k}$ et $M \in \mathbb{N}^*$. On cherche la somme maximale des éléments d'une sous-famille de E ne dépassant pas M :

$$\max\left\{\sum_{i\in I} n_i \ : \ I \subseteq [\![1,\ k]\!],\ \sum_{i\in I} n_i \leqslant M\right\}.$$

Naïvement, on peut décider d'énumérer toutes les sous-familles possibles, de calculer la somme de leurs éléments pour chercher le maximum : on aura une complexité au moins exponentielle à cause de l'énumération, c'est donc exclu.

Cherchons à voir ce qu'il se passe si on isole le dernier terme de la famille. On voit deux possibilités :

- soit $n_k > M$ et dans ce cas la somme maximale est à chercher dans $(n_1, \ldots, n_{k-1})$;
- soit $n_k \leqslant M$ et on là encore deux possibilités :
  - soit n_k fait partie de la somme cherchée et dans ce cas on doit don calculer

$$n_k + \max\left\{\sum_{i\in I} n_i \ : \ I \subseteq [\![1,\ k-1]\!],\ \sum_{i\in I} n_i + n_k \leqslant M\right\}$$

$$= n_k + \max\left\{\sum_{i\in I} n_i \ : \ I \subseteq [\![1,\ k-1]\!],\ \sum_{i\in I} n_i \leqslant \underbrace{M - n_k}_{0\leqslant\cdot\leqslant M}\right\}$$

  - soit n_k n'en fait pas partie et la somme maximale est à chercher dans $(n_1, \ldots, n_{k-1})$;

    donc il faut prendre le maximum des deux quantités évoquées ci-dessus.

Notons pour $i \in [\![0,\ k]\!]$ et $j \in [\![0,\ M]\!]$: $s_{i,\,j} = \max\left\{\sum_{\ell\in I} n_\ell \ : \ I \subseteq [\![1,\ i]\!],\ \sum_{\ell\in I} n_\ell \leqslant j\right\}$.

- $\forall j \in [\![0,\ M]\!],\ s_{0,\,j} = 0$ (sommes vide) ;
- $\forall i \in [\![0,\ k]\!],\ s_{i,\,0} = 0$ (sommes majorée par 0) ;
- en généralisant ce que l'on vient de voir, on a que

$$\forall (i,\ j) \in [\![1,\ k]\!] \times [\![1,\ M]\!],\ s_{i,\,j} = \begin{cases} s_{i-1,\,j} & \text{si } n_i > j \\ \max(s_{i-1,\,j-n_i} + n_i,\ s_{i-1,\,j}) & \text{sinon} \end{cases}.$$

Les sous-problèmes ne sont pas indépendants donc on va procéder récursivement avec mémoïsation.

On donne l'algorithme en langage CAML. On suppose que la famille E est donnée sous la forme d'un vecteur de taille k. Une matrice `s` va contenir les valeurs $(s_{i,\,j})_{i,\,j}$. En parallèle de la construction de la matrice `s`, on va construire une matrice `p` qui va contenir, sous forme de listes d'entiers, les indices de la sous-famille de E dont la somme correspond à la somme $s_{i,\,j}$.

La fonction `partie_somme_max : int vect -> int -> int * int list` renverra la somme maximale et l'indexation de la sous-famille réalisant cette somme.

Programme

```
let partie_somme_max n m =
  (* On prend garde au fait que n est indexe a partir de 0 !
  n_i devient n.(i - 1) dans le code. *)
  let k = vect_length n in
  let s = make_matrix (k + 1) (m + 1) (-1) in
  for i = 0 to k do
    s.(i).(0) <- 0  (* cas de base de la matrice s *)
  done ;
  for j = 0 to m do
    s.(0).(j) <- 0
  done ;
  let p = make_matrix (k + 1) (m + 1) [] in
  let rec partie_somme_max_aux i j =
    match (i, j) with
    | (0, _) -> (0, []) (* cas de base *)
    | (_, 0) -> (0, [])
    (* appel deja effectue avant *)
    | _ when s.(i).(j) >= 0 -> (s.(i).(j), p.(i).(j))
    (* appel effectue pour la premiere fois *)
    | _ -> begin
        let (val_s, val_p) =  partie_somme_max_aux (i - 1) j
        in
        if n.(i - 1) > j then begin
          s.(i).(j) <- val_s ;
          p.(i).(j) <- val_p ;
          (val_s, val_p)
        end
        else begin
          let (val_s1, val_p1) =
            partie_somme_max_aux (i - 1) (j - n.(i - 1))
          in
          if val_s1 + n.(i - 1) > val_s then begin
            s.(i).(j) <- val_s1 + n.(i - 1) ;
            p.(i).(j) <- (i - 1) :: val_p1 ;
            (s.(i).(j), p.(i).(j))
          end
          else begin
            s.(i).(j) <- val_s ;
            p.(i).(j) <- val_p ;
            (s.(i).(j), p.(i).(j))
          end
        end
      end
  in
  let val_s, val_p = partie_somme_max_aux k m in
  (val_s, rev val_p)
;;

#partie_somme_max [| 1 ; 2 ;  1 ; 5  ; 2 ; 3 ; 4|] 10 ;;
- : int * int list = 10, [0; 1; 3; 4]
```

```
#partie_somme_max [| 1 ; 2 ;  1 ; 5  ; 2 ; 3 ; 4|] 11 ;;
- : int * int list = 11, [0; 1; 2; 3; 4]
#partie_somme_max [| 3 ; 5 ; 6|] 4 ;;
- : int * int list = 3, [0]
```

Les complexités temporelles et spatiales sont en $O(k \cdot M)$ car l'on doit parcourir la matrice `s` pour résoudre le problème.