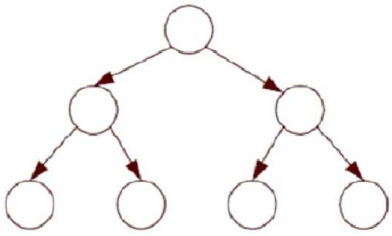
La programmation dynamique

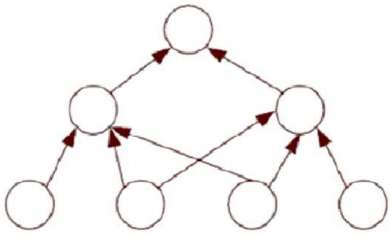
L'idée de départ : Les algorithmes récursifs sont souvent très simples à mettre en oeuvre, mais ont une complexité temporelle parfois élevée. Comment procéder pour préserver l'avantage du récursif tout en diminuant significativement la complexité ?

Exemple : L'algorithme récursif de calcul du nème terme de la suite de Fibonacci est très simple à programmer, mais a une complexité exponentielle...

Solution proposée : Partir des cas de base au lieu de descendre vers eux.



Programmation Récursive



Programmation Dynamique

I] Exemples simples

Comment faire ? Comme dans le principe de récurrence, les cas de bases vont nous permettre, grâce à la formule de récursivité, de déterminer les valeurs renvoyées par la fonction pour des arguments de plus en plus grands. L'idée est alors de stocker toutes les valeurs obtenues à partir des cas de base jusqu'à obtenir le résultat souhaité.

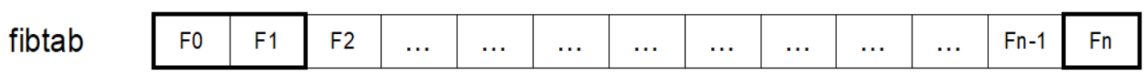
Bien que reposant sur une formule de récursivité, un tel algorithme est de nature itérative.

Exemple : Fibonacci

Le calcul de F_n passe par les calculs successifs de F_0, F_1, ..., F_{n-1}.

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_{n+2} = F_{n-1} + F_n \end{cases}$$

L'idée est alors de stocker au fur et à mesure ces valeurs dans un tableau ligne nommé ici `fibtab`.



Chaque itération k de la boucle calcule F_k et le stocke dans le tableau `fibtab`.
Le résultat de la fonction se trouve alors dans la dernière case du tableau.

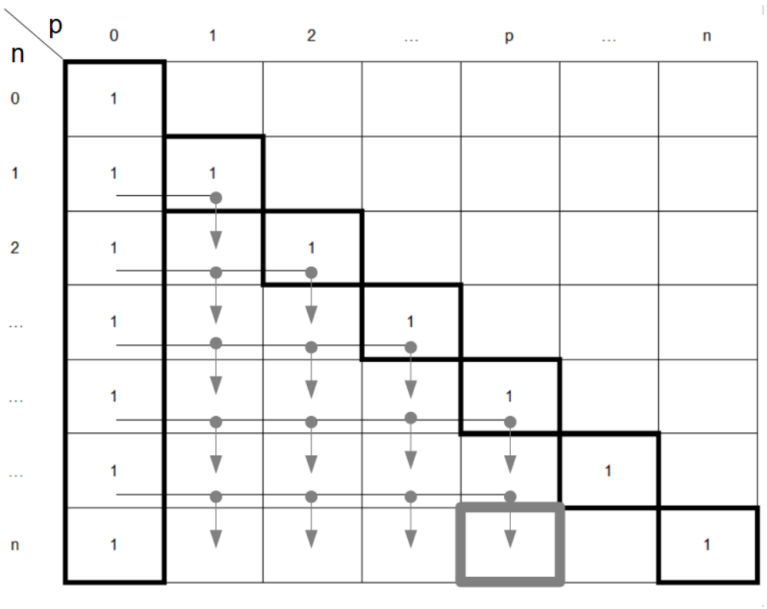
OCaml
```
let fib n = let fibtab = Array.make (n+1) 0 in      (* tableau de stockage *)
            fibtab.(0) <- 0;                        (* Initialisation du tableau *)
            fibtab.(1) <- 1;
            for k = 2 to n                          (* remplissage du tableau *)
                      do
                      fibtab.(k) <- fibtab.(k-1) + fibtab.(k-2)
                      done;
            fibtab.(n);;                            (* extraction de la valeur voulue *)

fib 5;;
```

Exemple 2 : Calcul de $\binom{n}{p}$

La formule d'addition permet le calcul de $\binom{n}{k}$ à partir valeurs $\binom{n'}{k'}$ pour tout $\begin{cases} 0 \leq n' < n \\ 0 \leq k' \leq k \end{cases}$

$$\boxed{\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{et} \quad \binom{n}{0} = 1 \quad \text{et} \quad \binom{n}{n} = 1}$$

L'idée est alors de stocker au fur et à mesure ces valeurs dans un tableau à deux entrée nommé ici `t`.



La composante `t.(n).(p)` contient alors le résulat à renvoyer.

OCaml
```
let binom k n = let t = Array.make_matrix (n+1) (n+1) 0 in   (* création du tableau *)
                for i = 0 to n do t.(i).(0) <- 1;            (* initialisation du tableau *)
                                  t.(i).(i) <- 1             (* initialisation du tableau *)
                               done;
                for i = 2 to n do                            (* remplissage du tableau *)
                    for j = 1 to (min k (i-1)) do
                        t.(i).(j) <- t.(i-1).(j-1) + t.(i-1).(j)
                                  done;
                               done;
    t.(n).(k);;                                              (* extraction de la valeur voulue *)

binom 7 12;;
```

Structure du programme :

Les 2 exemples précédents font apparaître la même structure :

- Définition du tableau de stockage des valeurs
- Initialisation du tableau avec les valeurs des cas de base
- Remplissage du tableau jusqu'à la valeur cherchée

Exercice : 1

Le nombre de surjections d'un ensemble de n éléments vers un ensemble à p éléments est donné par la formule de récurrence suivante :

$$S_n^p = p(S_{n-1}^{p-1} + S_{n-1}^p)$$

Programmer le calcul de S_n^p à l'aide d'un algorithme de programmation dynamique.

Etapes de programmation :

1. **Formule de récursivité :**
   Obtention de la formule de récursivité liant la solution d'un problème à celle de sous-problèmes.

2. **Définition et Initialisation de la table :**
   Selon le nombre d'arguments entiers, on utilise une table de dimension 1, 2 ou plus.
   L'initialisation de la table correspond aux valeurs des cas de base de la formule de récursivité.

3. **Remplissage de la table :**
   On remplit les valeurs du tableau à l'aide de boucles imbriquées en partant des cas de base et en se servant de la formule de récursivité.

4. **Lecture de la solution :**
   Contrairement aux exemples traités précédemment, la solution du problème de départ n'est pas toujours stockée dans l'une des cases de la table. Pour obtenir cette solution il est souvent nécessaire de déchiffrer la table obtenue. Pour cela, on est amené (voir les algorithmes de la fin du cours) à lire la table en suivant un chemin inverse à celui suivi lors de la remplissage du tableau.

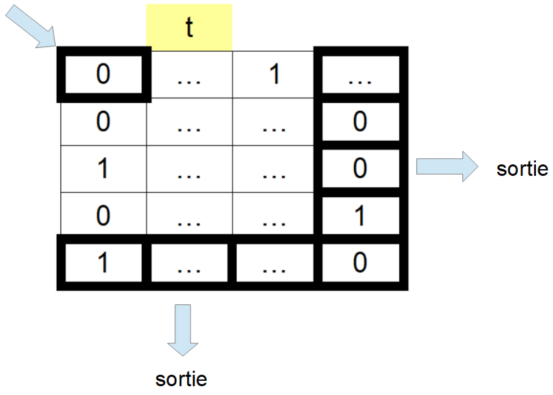
II] L'exemple d'un labyrinthe

Problèmes d'optimisation :

La programmation dynamique est surtout utilisée pour résoudre des problèmes d'optimisation pour lesquels la solution optimale s'obtient en considérant les solutions optimales à des sous-problèmes du même type. L'exemple qui suit en donne un premier exemple assez simple.

Formulation du problème :

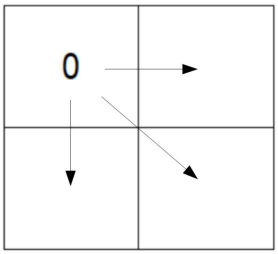
On modélise un "labyrinthe" par une matrice `t` de taille `nxp` ne contenant que des 0 et des 1.



Les 0 représentent les espaces vides et les 1 représentent les cases inaccessibles.

On entre par la case `t.(0).(0)` supposée donc vide et on souhaite ressortir par l'une des cases $\begin{cases} \texttt{t.(i).(p-1)} \\ \text{ou} \\ \texttt{t.(n-1).(j)} \end{cases}$.

On impose les règles de déplacement suivantes :

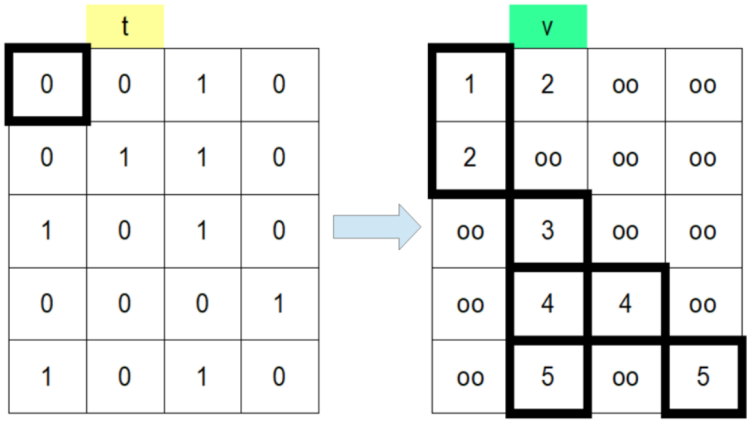


L'objectif :

Il s'agit pour nous de :

- savoir s'il existe un chemin possible permettant de ressortir
- si oui :
  - → connaître les sorties `t.(i).(p-1)` et `t.(n-1).(j)` possibles
  - → connaître la sortie correspondant au chemin le plus court
  - → connaître le chemin le plus court permettant de sortir

On décide de s'intéresser pour chacune des cases du tableau, à la longueur du plus court chemin permettant de s'y rendre. On note `v` le nouveau tableau obtenu en posant `v.(i).(j) = oo` lorsque cette case est inaccessible.



Un tel tableau permet de répondre aux questions posées :

- Il existe deux sorties possibles : `t.(4).(3)` et `t.(4).(1)`
- Ces sorties correspondent à des chemins de même longueur 5
- On retrouve les chemins parcourus en "lisant" le tableau à l'envers en partant de la case de sortie.

Programmation dynamique :

- Formule de récursivité : On veut connaître la valeur de `v.(i).(j)`

  On ne peut arriver en `t.(i).(j)` qu'en effectuant 1 déplacement à partir des cases $\begin{cases} \texttt{t.(i-1).(j)} \\ \texttt{t.(i-1).(j-1)} \\ \texttt{t.(i-1).(j-1)} \end{cases}$.

  → Lorsque `t.(i).(j) = 1` ou que $\begin{cases} \texttt{v.(i-1).(j) = infini} \\ \texttt{v.(i).(j-1) = infini} \\ \texttt{v.(i-1).(j-1) = infini} \end{cases}$, la case `t.(i).(j)` ne peut être atteinte.

  On a alors : `v.(i).(j) = infini`.

  → Sinon, la longueur du plus court chemin pour arriver en `t.(i).(j)` est :

  ```
  v.(i).(j) = 1 + min (v.(i-1).(j), v.(i).(j-1), v.(i-1).(j-1))
  ```

- Cas de base :

  Pour pouvoir appliquer la formule précédente, il faut connaître les valeurs de $\begin{cases} \text{la première ligne} \\ \text{la première colonne} \end{cases}$.

  → Si `v.(i-1).(0) = infini` ou `t.(i).(0) = 1` alors : `v.(i).(0) = infini`

  Sinon : `v.(i).(0) = v.(i-1).(0) + 1`

  → Si `v.(0).(j-1) = infini` ou `t.(0).(j) = 1` alors : `v.(0).(j) = infini`

  Sinon : `v.(0).(j) = v.(0).(j-1) + 1`

- Programme OCaml :

  L'entier "infini" n'existant pas en OCaml, on prendra une valeur ne pouvant être jamais atteinte pour la longueur d'un chemin. La longueur maximale d'un chemin étant $l = n + p - 1$, on pourra prendre par exemple : `t = 10*(n+p)`

OCaml
```
let tablab t = let n = Array.length t and
                   p = Array.length t.(0) in
               let infini = 10*(n + p) and
                   v = Array.make_matrix n p 1 in
               for i = 1 to (n-1) do if v.(i-1).(0) = infini ||    (* Init° colonne 1 *)
                                        t.(i).(0) = 1
                                     then v.(i).(0) <- infini
                                     else v.(i).(0) <- v.(i-1).(0) + 1
                                   done ;
               for j = 1 to (p-1) do if v.(0).(j-1) = infini ||    (* Init° ligne 1 *)
                                        t.(0).(j) = 1
                                     then v.(0).(j) <- infini
                                     else v.(0).(j) <- v.(0).(j-1) + 1
                                   done ;
               for i = 1 to (n-1) do                               (* Rempl. du tableau *)
                   for j = 1 to (p-1) do if t.(i).(j) = 1 ||
                                            (v.(i-1).(j) = infini &&
                                             v.(i).(j-1) = infini &&
                                             v.(i-1).(j-1) = infini)
                                         then v.(i).(j) <- infini
                                         else let a = min v.(i-1).(j) v.(i).(j-1) in
                                              v.(i).(j) <- 1 + min a v.(i-1).(j-1)
                                        done;
                                   done;
               v;;
```

OCaml
```
tablab [|[|0 ; 0 ; 1 ; 0|];
         [|0 ; 1 ; 1 ; 0|] ;
         [|1 ; 0 ; 1 ; 0|] ;
         [|0 ; 0 ; 0 ; 1|] ;
         [|1 ; 0 ; 1 ; 0|]|];;

- : int array array = [|[|1  ; 2  ; 90 ; 90|];
                        [|2  ; 90 ; 90 ; 90|];
                        [|90 ; 3  ; 90 ; 90|];
                        [|90 ; 4  ; 4  ; 90|];
                        [|90 ; 5  ; 90 ; 5|]|]
```

Exercice : 2

Finir l'exercice sur le labyrinthe en :

1. Proposant une fonction permettant de déterminer la ou les cases de sortie possibles
2. Proposant une fonction permettant de déterminer la sortie correspondant à un plus court chemin
3. Proposant une fonction permettant de déterminer les cases par lesquelles passe le plus court chemin.

III] Mémoïzation

Idée : Pour optimiser le temps de réalisation d'un programme, on peut envisager de garder en mémoire les résultats obtenus lors des précédentes utilisations (expl : fonction *remember* de Maple pour les fonction récursives). C'est ce que fait la *mémoire cache* pour la navigation sur le web.

Dans le cas de la programmation dynamique d'une fonction f, nous pouvons envisager de stocker systématiquement toutes les valeurs calculées dans un tableau mémoire T défini antérieurement de façon globale, initialisé avec les cas de base et dont la taille a été fixée pour pouvoir contenir toutes les combinaisons d'arguments envisageables en pratique.

L'amélioration de la fonction f peut alors se programmer de la façon suivante :

OCaml
```
let T = .... ;;                    (* Déf° du tableau GLOBAL de stockage des valeurs de la f° *)
                                   (* Ce tableau est initialisé avec les cas de base          *)

let f x1 ... xn = match tab[x1,...,xn] <> valeur d'initialisation du tableau T with

                        | true -> renvoyer T[x1,...,xn]

                        | -    -> let s = f(x1,...,xn) in   (* avec la formule de récursivité *)
                                     T[x1,...,xn] <- s:
                                     s;;
```

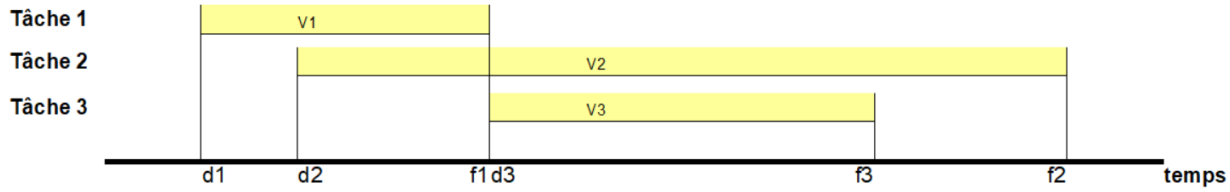
Ce choix de mémoïzation engendre la mobilisation d'un espace mémoire supplémentaire, mais permet d'éviter de refaire des calculs d'ores et déjà effectués lors d'une utilisation antérieure de la fonction. Si on y perd en complexité spatiale, on y gagne cependant beaucoup en complexité temporelle.

Bien entendu, ce principe de *mémoïzation* s'applique à toutes les formes de programme : itératifs et récursif.

IV] Ordonnancement de tâches

Ce problème est aussi connu sous sa formulation anglosaxonne : "weighted interval scheduling".

Le problème : L'objectif est d'obtenir à partir d'un ensemble de tâches, chacune définie par un intervalle de temps et une pondération, un sous-ensemble de tâches compatibles (pas de chevauchement des intervalles) dont la somme des pondérations est maximale.



Mise en forme mathématique du problème :

Notations :

- $[\![0, n-1]\!]$ l'ensemble des n tâches à effectuées
- $(d_i,\ f_i)$ l'intervalle de temps associé à la tâche i
- v_i la pondération associée à la tâche t_i.

On peut stocker ces n tâches dans un tableau `t` sous la forme

```
t = |[(d0,f0,v0) ; .... |]
```

On supposera ce tableau est initialement ordonné selon les f_i croissants.

Objectif :

On cherche : $J \subset [\![0, n-1]\!]$ tel que : $\begin{cases} \sum_{i \in J} v_i \quad \text{soit maximale} \\ \forall i_k \in J = \{i_1,\ i_2,\ \ldots,\ i_p\}, \quad f_{i_k} \leq d_{i_{k+1}} \end{cases}$.

Formatisation :

Pour $i \in [\![0, n-1]\!]$, notons :

- V_i la valeur maximale que l'on peut obtenir en sélectionnant des tâches compatibles parmi les tâches de 0 à i.
- $p(i)$ la plus grande tâche $j < i$ tel que $f_j \leq d_i$, c'est à dire la dernière tâche parmi $[\![0, i-1]\!]$ compatible avec i.

Si aucune de ces tâches est compatible avec i alors on pose $p(i) = -1$.

Formule de récursivité :

Pour $i \geq 1$, la valeur de V_i s'obtient de deux façons possibles selon que la tâche i est retenue ou non.
Comme a priori, on ne sait pas si c'est le cas, on envisage les deux possibilités :

- Si la tâche i n'est pas retenue : $V_i = V_{i-1}$.
- Si la tâche i est retenue :

  Les autres tâches retenues doivent être compatibles avec i.
  Il faut donc les choisir parmi les tâches d'index 1 à $p(i)$.

  On obtient alors la valeur maximale V_i de la façon suivante :

  → Si $p(i) \neq -1$: il existe des tâches précédentes compatibles avec i

  $$V_i = v_i + V_{p(i)}$$

  → Si $p(i) = -1$: aucune tâche précédente est compatible avec i

  $$V_i = v_i$$

  On peut noter V_i' cette valeur.

La formule de récursivité est alors : $\boxed{V_i = \max(V_{i-1},\ V_i')}$

Tableau de stockage :

- La valeur V_i ne dépendant que d'un seul indice entier i, nous utiliserons un tableau ligne `v` à n composantes.
- La première composante de ce tableau est initialisée à `v.(0) =` v_0.
- Nous pourrons stocker les valeurs $p(i)$ obtenues dans un tableau `p`.
  Les valeurs $p(i)$ pourront s'obtenir à l'aide d'une fonction locale f.

Programmes :

On a supposé ici que les n tâches sont ordonnées selon la croissance des temps de fin.
On considère un tableau ligne `t` de n triplets $(d_i,\ f_i,\ v_i)$ pour coder l'ensemble des tâches à traiter.

- `indcomp ('a * 'a * 'b) array -> int array` : détermine le tableau contenant les valeurs $p(i)$

OCaml
```
let indcomp t = let n   = Array.length t in
                let p = Array.make n 0 in                (* Stockage des indices p(i) *)
                let f t i = let (di,fi,vi) = t.(i) and   (* f° de calcul de p(i)      *)
                                k = ref (-1) in          (* derniere tâche compatible *)
                     for j = 0 to (i-1)
                         do let (dj,fj,vj) = t.(j) in
                            if fj <= di then k := j;     (* mise à jour de k si j est *)
                         done;                           (* compatible avec i         *)
                     !k in
                for i = 0 to (n-1) do p.(i) <- f t i     (* remplissage du tableau *)
                                   done;
                p;;

indcomp [|(1,2,0);(1,4,3);(3,6,5);(4,7,2);(7,10,1)|];;
- : int array = [|-1; -1; 0; 1; 3|]
```

- `schedule : ('a * 'a * int) array -> int array * int array` : renvoie le tableau des sommes maximales de valeurs de tâches compatibles

OCaml
```
let schedule t = let n = Array.length t in
                 let v = Array.make n 0 and      (* Stockage des valeurs Vi   *)
                     p = indcomp t in            (* Stockage des indices p(i) *)
                 let (d0,f0,v0) = t.(0) in
                 v.(0) <- v0 ;                   (* Initialisation de v       *)
                 for i = 1 to (n-1) do let (di,fi,vi) = t.(i) in
                                       let ti = ref vi in
                                           if p.(i) <> -1 then ti := v.(p.(i)) + vi;
                                           v.(i) <- max v.(i-1) !ti
                                   done;
                 v;;   (* Tableaux des valeurs MAX *)

schedule [|(1,2,0);(1,4,3);(3,6,5);(4,7,2);(7,10,1)|];;
- : int array = [|0; 3; 5; 5; 6|]
```

Décodage du tableau :

Le tableau obtenu ne donne que les valeurs maximales que l'on peut emporter en ne considérant que les i premiers objets. Mais à quels taches correspondent ces valeurs ?

Exemple :

OCaml
```
let taches = [|(0,0,0);(2,5,3);(3,6,2);(5,7,2);(2,8,1);
               (6,9,3);(5,10,5);(5,12,4)|];;

let ind = indcomp taches;;
# let ind = indcomp taches;;
val ind : int array = [|-1; 0; 0; 1; 0; 2; 1; 1|]

let val = schedule taches;;
# let valmax = schedule taches;;
val valmax : int array = [|0; 3; 3; 5; 5; 6; 8; 8|]
```

Pour le savoir, on parcourt le tableau à l'envers en se demandant comment la valeur étudiée a été déterminée.

On rappelle la formule de récursivité : $\boxed{V_i = \max(V_{i-1},\ v_i + V_{p(i)} \text{ ou } v_i)}$

Avec :

- $V_i = V_{i-1}$ lorsque la tache t_i n'a pas été sélectionnée.
- $V_i = v_i + V_{p(i)}$ lorsque t_i a été sélectionnée et $p(i) \neq -1$.
  Dans ce cas, les autres taches font parties de $\{t_0, \ldots, t_{p(i)}\}$
- $V_i = v_i$ lorsque t_i a été sélectionnée et $p(i) = -1$.
  Dans ce cas, seule la tache t_i a été sélectionnée.

Dans certains cas, il est possible d'avoir à la fois $V_i = V_{i-1}$ et $V_i = v_i + V_{p(i)}$.
Cela signifie que plusieurs combinaisons de tâches donnent une valeur totale maximale.

Reprenons l'exemple précédent :

Etape 1 :

indices	0	1	2	3	5	5	6	7
taches compatibles	-1	0	0	1	0	2	1	1
valeurs maximales	0	3	3	5	5	6	8	8

Nous avons $V_7 = V_6$ par conséquent la tâche t_7 n'a pas été sélectionnée.
On regarde alors parmi les tâches $t_0, \ldots, t_6$.

Etape 2 :

indices	0	1	2	3	5	5	6	7
taches compatibles	-1	0	0	1	0	2	1	1
valeurs maximales	0	3	3	5	5	6	8	8

Nous avons $V_6 \neq V_5$ par conséquent la tâche t_6 a été sélectionnée.
Pour connaître l'ensemble des autres tâches potentielles, on regarde la valeur de $p_6 = 1$.
Cela signifie que les autres tâches sont parmi $\{t_0, t_1\}$.

Etape 3 :

indices	0	1	2	3	5	5	6	7
taches compatibles	-1	0	0	1	0	2	1	1
valeurs maximales	0	3	3	5	5	6	8	8

Nous avons $V_1 \neq V_0$ par conséquent la tâche t_1 a été sélectionnée.
Pour connaître l'ensemble des autres tâche potentielles, on regarde la valeur de $p_1 = 0$.
Cela signifie que les autres tâches sont parmi $\{t_0\}$.

Etape 4 :

indices	0	1	2	3	5	5	6	7
taches compatibles	-1	0	0	1	0	2	1	1
valeurs maximales	0	3	3	5	5	6	8	8

La tâche t_0 étant la dernière, elle a automatiquement été sélectionnée.

Les taches sélectionnées sont donc : t_0, t_1, t_6

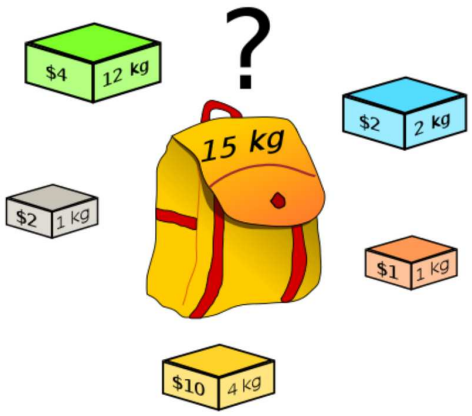
Programme de décodage :

OCaml
```
let decode valmax ind =
           let l = ref [] and                    (* Pour stocker les tâches sélectionnées *)
               n = Array.length ind in
           let k = ref (n-1) in                  (* Indice de la tâche a étudier *)
           while !k <> (-1) && !k <> 0 do        (* Tant que la tache est compatible et <> t0 *)
                 if valmax.(!k) <> valmax.(!k-1)
                 then begin l := (!k)::(!l);     (* tk est sélectionnée *)
                            k := ind.(!k)        (* nouvelle tache a étudier *)
                      end
                 else k := !k - 1                (* on s'intéresse à la tache précédente *)
                                    done;
            if !k = 0 then l := 0::(!l);         (* ajout de t0 si compatible *)
           !l;;
```

Application :

OCaml
```
let taches = [|(0,0,0);(2,5,3);(3,6,2);(5,7,2);(2,8,1);(6,9,3);(5,10,5);(5,12,4)|];;
let ind = indcomp taches;;
let valmax = schedule taches;;
decode valmax ind;;

# decode valmax ind;;
- : int list = [0; 1; 6]
```

V] Problème du sac à dos



La situation :

Soit :

- un ensemble de n objets $E_n = [\![1, n]\!]$,
- un sac à dos pouvant contenir une masse maximale `M`.

Chaque objet i a une masse $m_i \in \mathbb{N}^*$ et une valeur $v_i \in \mathbb{R}^{+*}$.

Le problème :

Le problème consiste à choisir un ensemble d'objets parmi les n objets (un objet ne pouvant être choisi qu'une seule fois) de telle manière que la somme totale des valeurs soit maximisée, sans dépasser la capacité M du sac.

La modélisation :

En termes mathématiques, le problème se formule ainsi :

$$\text{On cherche } (x_1,\ x_2,\ \ldots,\ x_n) \in \{0,\ 1\}^n \text{ tel que : } \begin{cases} \sum_{k=1}^{n} x_i v_i \text{ soit maximal} \\ \sum_{k=1}^{n} x_i m_i \leq M \end{cases}$$

Bien entendu, on recherche un algorithme de complexité inférieure à la complexité de la force brute.

Recherche d'une formule de récursivité :

Valeur Maximale :

Notons V_i^j la valeur maximum généré par le choix d'objets dont la somme des poids ne dépasse pas j parmi les i premiers objets.
Résoudre le problème revient alors à trouver la valeur de V_n^M.

Pour calculer V_i^j la séquence d'objets peut être divisée en deux sous-ensembles : $\begin{cases} \text{les } (i-1) \text{ premiers objets} \\ \text{l'objet } i \end{cases}$.

L'objet i est alors $\begin{cases} \text{soit choisi} \\ \text{soit ignoré} \end{cases}$ dans le calcul de V_i^j.

- On commence par tester si le poids de l'objet i ne dépasse pas la capacité j que l'on s'est imposée. Si c'est le cas, alors nous avons :

$$\boxed{V_i^j = V_{i-1}^j}$$

- Si la masse de l'objet i est inférieure à j, alors nous devons considérer deux possibilités :
  - → Soit on choisit l'objet i et il contribue alors à la solution optimale et nous avons :

$$V_i^j = V_{i-1}^{j-m_i} + v_i$$

  - → Soit l'objet i n'est pas choisi dans la solution optimale et la capacité du sac est alors inchangée. La solution optimale est donc celle obtenue en ne considérant que les $i-1$ premiers objets, soit :

$$V_i^j = V_{i-1}^j$$

Pour trouver V_i^j il suffit alors de prendre le maximum entre le cas où l'objet est choisi ou ignoré :

$$\boxed{V_i^j = \max(V_{i-1}^{j-m_i} + v_i,\ V_{i-1}^j)}$$

La formule de récursivité est alors : $V_i^j = \begin{cases} 0 & \text{si } i = 0 \text{ ou } j = 0 \\ V_{i-1}^j & \text{si } w_i > j \text{ et } i > 0 \\ \max(V_{i-1}^{j-m_i} + v_i,\ V_{i-1}^j) & \text{sinon} \end{cases}$.

Définition et Initialisation du tableau :

La valeur V_i^j dépendant des deux paramètres entiers i et j, on utilise une matrice de taille $(n+1) \times (M+1)$ pour stocker ces valeurs V_i^j.

Les cas de base sont : $V_i^j = 0$ pour $i = 0$ ou $j = 0$.

Programmation OCaml :

L'implémentation OCaml de construction de la matrice est la suivante :

Fiche signalétique :

OCaml
```
Nom : sac_a_dos
Arguments :  o : le tableau contenant les objets (v,m) susceptibles
                 d'être emportés
             m_max : la masse maximale admissible par le sac à dos
Sortie : le tableau contenant les valeurs V_i^j
Algorithme : Dynamique
```

OCaml
```
let sac_a_dos o m_max = let n = Array.length o in
                        let valmax = Array.make_matrix (n+1) (m_max+1) 0 in
     for i = 1 to n do
         for j = 1 to m_max do
             let (vi,mi) = o.(i-1) in
             if mi > j then valmax.(i).(j) <- valmax.(i-1).(j)
                       else valmax.(i).(j) <- max (valmax.(i-1).(j)) (valmax.(i-1).(j-mi)+vi)
                        done;
                    done;
    valmax;;

sac_a_dos [|(1,1);(6,2);(18,5);(22,6);(28,7)|] 11;;
```

Application :

OCaml
```
sac_a_dos [|(1,1);(6,2);(18,5);(22,6);(28,7)|] 11;;

- : int array array =
[|[|0; 0; 0; 0; 0;  0;  0;  0;  0;  0;  0;  0|];
  [|0; 1; 1; 1; 1;  1;  1;  1;  1;  1;  1;  1|];
  [|0; 1; 6; 7; 7;  7;  7;  7;  7;  7;  7;  7|];
  [|0; 1; 6; 7; 7; 18; 19; 24; 25; 25; 25; 25|];
  [|0; 1; 6; 7; 7; 18; 22; 24; 28; 29; 29; 40|];
  [|0; 1; 6; 7; 7; 18; 22; 28; 29; 34; 35; 40|]|]
```

Nous avons ainsi rempli une table de valeurs dont la dernière case d'indice (n, M) (n étant le nombre d'objets que nous souhaitons emporter et M la masse totale admissible) donne la somme des valeurs des objets emportés. Mais nous ne savons toujours pas quels sont les objets à emporter...

Décodage sur un exemple :

Partons de l'exemple ci-dessus : `sac_a_dos [|(1,1);(6,2);(18,5);(22,6);(28,7)|] 11;;`

La solution optimale est donc donnée par `V.(5).(11) = 40`.
Nous obtenons donc la valeur maximale que peut contenir le sac, mais le résultat ne nous dit pas quels sont les objets qui ont été sélectionnés. Pour retrouver les objets faisant partie de la solution optimale, on parcourt le tableau V obtenu en partant de la dernière valeur :

Etape 1 :

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Nous remarquons que $V[5,11] = \max(V[4,11],\ V[4,11-m_5]+v_5) = V[4,11]$.
Cela signifie que l'objet 5 ne fait pas partie des objets sélectionnés.

Etape 2 :

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Nous remarquons que $V[4,11] = V[3, 11 - m_4] + v_4 = V[3,5] + 22$ car $(v_4, m_4) = (22, 6)$.
L'objet 4 est donc choisi et les autres objets choisis sont parmi les 3 premiers objets.

Etape 3 :

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5	0	1	6	7	7	18	22	28	29	34	35	40

Nous remarquons que $V[3,5] = V[2, 5 - m_3] + v_3 = V[2,0] + 18$ car $(v_3, m_3) = (18, 5)$.
L'objet 3 est donc choisi et les autres objets choisis sont parmi les 2 premiers objets.
Comme nous avons alors $j = 0$ cela signifie qu'il n'est plus possible d'ajouter d'objets dans le sac.

Les objets faisant partie de la solution optimale sont donc les objets 4 et 3

Programme de décodage :

Fiche signalétique :

OCaml
```
Nom : decode
Arguments :  v : le tableau de valeurs obtenu à l'étape précédente
             o : la liste des objets susceptibles d'être emportés dans le sac
Sortie : la liste contenant les objets à emporter
Algorithme : on utilise une fonction récursive "aux valmax o (i,j)" qui traite
             la partie du tableau jusqu'à l'élément (i,j)
```

OCaml
```
let decode valmax o = let n = (Array.length valmax) - 1
                      and masmax = (Array.length valmax.(0)) - 1
                  in let rec aux valmax o = function
                      | (0,j) -> []
                      | (i,0) -> []
                      | (i,j) -> let (v,m) = o.(i-1) in
                                 match valmax.(i).(j) = valmax.(i-1).(j)
                                 with
                                 | true -> aux valmax o (i-1,j)
                                 | false -> i::(aux valmax o (i-1,j-m))
                in aux valmax o (n,masmax);;
```

Application à l'exemple :

OCaml
```
let valmax = sac_a_dos [|(1,1);(6,2);(18,5);(22,6);(28,7)|] 11;;
let o = [|(1,1);(6,2);(18,5);(22,6);(28,7)|];;
decode valmax o;;

- : int list = [4; 3]
```

On obtient bien les objets 3 et 4...

VI] Calcul de la distance d'édition

Le problème : On souhaite mesurer la similitude entre deux chaînes de caractères.

Exemple : Un exemple classique d'utilisation de la *distance d'édition* est lorsque Google renvoie le même résultat de recherche lorsqu'on lui entre "dynamic" et "dymanic" comme mots clés.

La modélisation : On définit sur les mots trois opérations élémentaires sur les chaînes de caractères :

- la substitution : on remplace une lettre par une autre,
- l'insertion : on ajoute une nouvelle lettre,
- la suppression : on supprime une lettre.

La **distance d'édition** entre deux mots U et V est alors le nombre minimal d'opérations nécessaires pour transformer U en V.

Exemples : Sur le mot `carie`, si on substitue c en d, a en u et si on insère t après le i, on obtient `durite`. On peut démontrer que ce nombre d'opérations est minimal et que la distance de `carie` à `durite` est donc 3 :

$$\begin{cases} \text{deux substitutions} \\ \text{une insertion} \end{cases}$$

De même, on montre que la distance de `aluminium` à `albumine` est 4 :

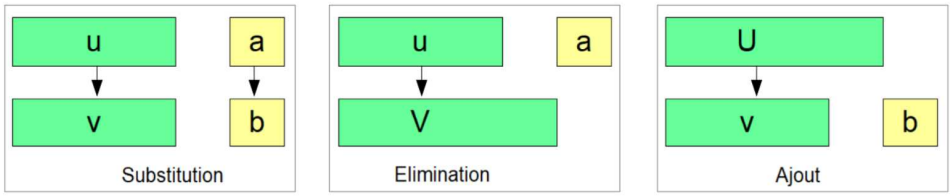
$$\begin{cases} \text{une insertion b} \\ \text{une substitution i en e} \\ \text{2 suppressions u et m} \end{cases}$$

Formule de récursivité :

Considérons deux mots $\begin{cases} U = ua \\ V = vb \end{cases}$ avec $\begin{cases} u \text{ et } v \text{ deux mots} \\ a \text{ et } b \text{ deux caractères} \end{cases}$.

Il y a plusieurs façons de transformer U en V :

- Lorsque a=b, on transforme naturellement u en v.
- Lorsque a≠b :



→ Avec une substitution : On commence par transformer u et v, puis on substitue a en b

→ Avec une élimination : On commence par transformer u en V et on élimine le a

→ Avec un ajout : On commence par transformer U en v et on ajoute le b

Comme nous souhaitons choisir parmi ces 3 possibilités, celle qui génère le moins d'opérations, nous obtenons alors la formule de récursivité suivante :

Formule de récursivité :

- Si $a = b$: $d(ua,\ va) = d(u,\ v)$
- Si $a \neq b$: $d(ua,\ vb) = 1 + \min(d(u,\ v),\ d(ua,\ v),\ d(u,\ vb))$

Les cas de base :

Au bout d'un nombre fini d'appels récursifs, l'un des deux mots devient "vide".
En notant ε le mot vide et $|u|$ le nombre de caractères d'un mot u, on obtient les cas de base suivant :

Cas de base :

- $d(\varepsilon,\ V) = |V|$ (on insère par exemple les lettres de V dans ε)
- $d(U,\ \varepsilon) = |U|$ (on élimine par exemple toutes les lettres de U)

Programmation dynamique :

Notons U et V les deux mots en argument avec n le nombre de lettres de U et m le nombre de lettres de V.

Valeur à optimiser :

$T(i, j)$ soit la distance d'édition du mot constitué des i premières lettres de U au mot constitué des j premières lettres de V.

On décide donc de stocker les résultats dans une matrice T de taille $(n + 1) \times (m + 1)$.

- La première colonne et la première ligne correspondant au cas où l'un des mots est vide.
- La valeur de $T(n, m)$ correspond alors à la distance d'édition de U à V.

Programme OCaml :

OCaml
```
let dist_ed mot1 mot2 =
    let n = String.length mot1 and
        m = String.length mot2 in
    let t = Array.make_matrix (n+1) (m+1) 0 in    (* création du tableau *)
    for i = 0 to n do t.(i).(0) <- i done;        (* initialisation du tableau *)
    for j = 0 to m do t.(0).(j) <- j done;
        for i = 1 to n do
            for j = 1 to m do if mot1.[i-1]=mot2.[j-1]
                              then t.(i).(j) <- t.(i-1).(j-1)
                              else let m1 = min t.(i-1).(j-1) t.(i).(j-1) in
                                   let m = min m1 t.(i-1).(j)
                                   in t.(i).(j) <- 1 + m
                          done;
                      done;
    t;;
```

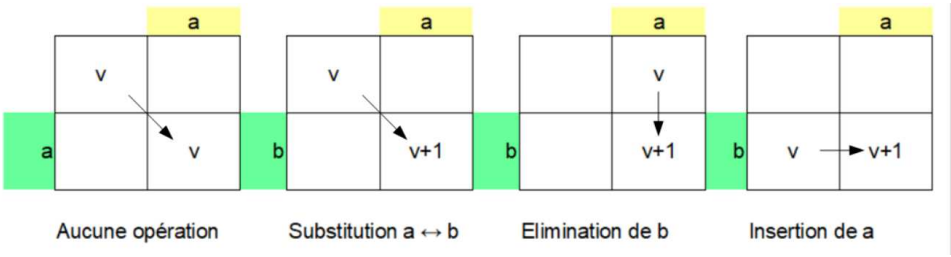
Application :

OCaml
```
dist_ed "durite" "carie";;

- : int array array = [|[|0; 1; 2; 3; 4; 5|];
                        [|1; 1; 2; 3; 4; 5|];
                        [|2; 2; 2; 3; 4; 5|];
                        [|3; 3; 3; 2; 3; 4|];
                        [|4; 4; 4; 3; 2; 3|];
                        [|5; 5; 5; 4; 3; 3|];
                        [|6; 6; 6; 5; 4; 3|]|]
```

La distance d'édition entre les mots "durite" et "carie" est donc 3.

Décodage de la matrice :

On aimerait maintenant déterminer les opérations élémentaires qui ont permis de transformer un mot en un autre mot. Il suffit pour cela, de décoder le tableau précédent.
En considérant qu'on trasforme le mot vertical en le mot horizontal, on remarque que :



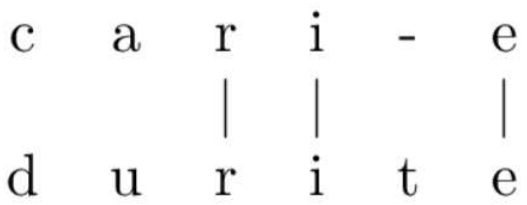
- Si $T(i,j) = T(i-1,j-1)$ avec `mot1.(i) = mot2.(j)` alors aucune opération a été effectuée.
- Si $T(i,j) = T(i-1,j-1)+1$ alors on a effectué une substitution
- Si $T(i,j) = T(i,j-1)+1$ alors il s'agit d'une insertion
- Si $T(i,j) = T(i-1,j)+1$ alors il s'agit d'une élimination

		vide	c	a	r	i	e
		0	1	2	3	4	5
vide	0	0	1	2	3	4	5
d	1	1	1	2	3	4	5
u	2	2	2	2	3	4	5
r	3	3	3	3	2	3	4
i	4	4	4	4	3	2	3
t	5	5	5	5	4	3	3
e	6	6	6	6	5	4	3

Interprétation :

Etape 1 : Aucune modification n'a été effectuée (lettres "e" identiques).

Etape 2 :	La lettre t a été éliminée.
Etape 3 :	Aucune modification n'a été effectuée.
Etape 4 :	Aucune modification n'a été effectuée.
Etape 5 :	Il y a eu substitution de u par a.
Etape 6 :	Il y a eu substitution de d par c.



Nous pouvons donc passer de "durite" à "carie" en effectuant une élimination et 2 substitutions.

Exercice : 3

En vous inspirant du tableau suivant :

1. Déterminer la distance d'édition entre "aluminium" et "albumine".
2. Donner les opérations élémentaires minimales permettant de transformer l'un en l'autre. Vous proposerez 2 solutions possibles.

OCaml
```
# dist_ed "aluminium" "albumine";;
- : int array array =
[|[|0; 1; 2; 3; 4; 5; 6; 7; 8|];
  [|1; 0; 1; 2; 3; 4; 5; 6; 7|];
  [|2; 1; 0; 1; 2; 3; 4; 5; 6|];
  [|3; 2; 1; 1; 1; 2; 3; 4; 5|];
  [|4; 3; 2; 2; 2; 1; 2; 3; 4|];
  [|5; 4; 3; 3; 3; 2; 1; 2; 3|];
  [|6; 5; 4; 4; 4; 3; 2; 1; 2|];
  [|7; 6; 5; 5; 5; 4; 3; 2; 2|];
  [|8; 7; 6; 6; 5; 5; 4; 3; 3|];
  [|9; 8; 7; 7; 6; 5; 5; 4; 4|]|]
```

Exercice : 4

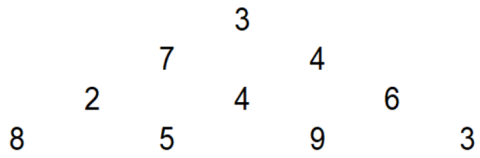
Ecrire une procédure qui prend pour arguments deux chaînes de caractères et affiche la suite des opérations élémentaires minimales permettant de transformer le premier mot en le deuxième.

VII] Exercices

Exercice : 5

Chemin à valeur maximale dans un triangle isocèle

Déterminer un algorithme dynamique permettant de déterminer le chemin allant du sommet à la base tel que la somme des valeurs par lequel il passe soit maximale.



Ce triangle pourra être codé dans un matrice dont chaque ligne correspond à une ligne du triangle.

Exercice : 6

Plus longue chaine commune

Notre objectif est ici de déterminer un algorithme de type dynamique permettant de déterminer la plus longue sous-chaînes de caractères communes à deux chaînes de caractères données. Il s'agit plus précisément de déterminer la plus longue chaîne de caractères que l'on retrouve à la fois dans deux chaînes u et v données avec le bon ordre, mais avec les caractères non nécessairement contigus.

Ainsi, la plus longue sous-chaîne commune à $\begin{cases} \texttt{programmation dynamique} \\ \texttt{exemplefacile} \end{cases}$ est de longueur 4 et est : `maie`.

On considère donc ici deux chaînes de caractères u et v de longueurs respectives m et n.
On introduit une matrice $L = (l_{i,j}) \in \mathfrak{M}_{mn}(\mathbb{R})$ telle que $l_{i,j}$ représente la longueur de la plus longue sous-chaîne commune aux préfixes de longueur i de la chaîne u et j de la chaîne v.

1. Formule de récursivité : soit $(i,\ j) \in [\![0, m]\!] \times [\![0, n]\!]$

   (a) Comment initialiser les valeurs $l_{i,0}$ et $l_{0,j}$?
   (b) Si $u_i \neq v_j$, exprimer $l_{i,j}$ en fonction de $l_{i-1,j}$ et de $l_{i,j-1}$.
   (c) Si $u_i = v_j$, exprimer $l_{i,j}$ en fonction de $l_{i-1,j}$, de $l_{i,j-1}$ et de $l_{i-1,j-1}$.

2. Programmation de la matrice L :

   (a) Proposer en pseudo-langage, un algorithme de construction de la matrice L et renvoyant la valeur $l_{m,n}$.
   (b) Traduire en langage CAML l'algorithme précédent.

Exercice : 7

Chemin de poid minimal dans une matrice

On considère ici une matrice $M \in \mathfrak{M}_{np}(\mathbb{R})$.
On décide de parcourir cette matrice à partir du terme $m_{1,1}$, soit en allant vers la droite, soit en descendant.
On appelle *poids du chemin parcouru*, la somme des coefficients de la matrice sur le chemin.
Par exemple, dans la matrice suivante, le poids du chemin indiqué en gras est 18.

$$M = \begin{pmatrix} \mathbf{1} & 4 & 2 & 5 \\ \mathbf{6} & \mathbf{1} & \mathbf{5} & 1 \\ 4 & 2 & \mathbf{3} & \mathbf{2} \end{pmatrix}$$

On cherche ici à construire un algorithme dynamique déterminant le chemin de poid minimal reliant $m_{1,1}$ à $m_{n,p}$.

On introduit pour cela la matrice $P = (p_{i,j}) \in \mathfrak{M}_{np}(\mathbb{R})$ telle que $p_{i,j}$ prend la valeur du poids minimal d'un chemin joignant $m_{1,1}$ à $m_{i,j}$.

1. Formules de récursivité :
   (a) Donner la valeur de $p_{1,1}$.
   (b) On considère ici que $i = 1$. Donner la formule liant $p_{1,j}$ à $p_{1,j-1}$ pour $j \in [\![2, p]\!]$.
   (c) On considère ici que $j = 1$. Donner la formule liant $p_{i,1}$ à $p_{i-1,1}$ pour $i \in [\![2, p]\!]$.
   (d) Pour $(i,\ j) \in [\![2, n]\!] \times [\![2, p]\!]$, exprimer $p_{i,j}$ en fonction de $p_{i-1,j}$ et de $p_{i,j-1}$.

2. Donner la matrice P associée à la matrice M donnée précédemment en exemple.
   Quel est son chemin de poids minimal ? Quel est son poids ?

3. Programmation de la matrice P :
   (a) Proposer en pseudo-langage, un algorithme de construction de la matrice P.
   (b) Traduire en langage OCaml l'algorithme précédent.

4. Recherche de la solution :
   (a) Expliquer comment à partir de la matrice P, on peut retrouver le chemin de poids minimal.
   (b) Rédiger un programme de déchiffrage de la matrice P permettant de donner le chemin de poids minimal de M ainsi que son poids.