Chapitre 9

Programmation dynamique

1. Introduction

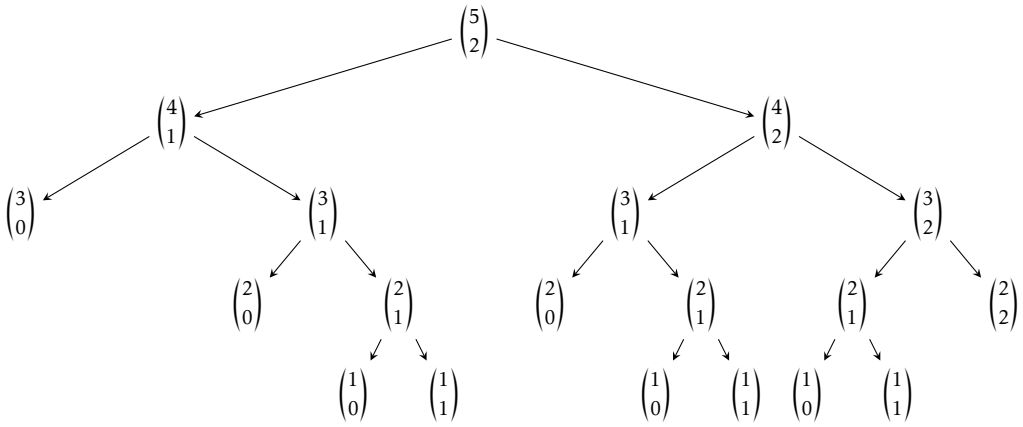
1.1 Un inconvénient de la programmation récursive

Nous allons nous intéresser au calcul du coefficient binomial $\binom{n}{p}$. Une solution consiste à utiliser la formule de Pascal, ce qui nous amène à écrire :

```
let rec binom = fun
  | n 0            -> 1
  | n p when n = p -> 1
  | n p            -> binom (n-1) (p-1) + binom (n-1) p ;;
```

Si on pose $E = \left\{(n,p) \in \mathbb{N}^2 \;\middle|\; p \leqslant n\right\}$, $A = \left\{(n,0),(n,n) \;\middle|\; n \in \mathbb{N}\right\}$ et que l'on munit E de l'ordre produit, les fonctions φ_1 et φ_2 définies sur $E \setminus A$ par : $\varphi_1(n,p) = (n-1,p-1)$ et $\varphi_2(n,p) = (n-1,p)$ vérifient : $\varphi_1(n,p) \prec (n,p)$ et $\varphi_2(n,p) \prec (n,p)$. Ceci suffit à assurer la terminaison de la fonction définie ci-dessus pour tout $(n,p) \in E$. Enfin, la formule de Pascal suffit à justifier par induction que cette fonction calcule effectivement le coefficient binomial.

Malheureusement, cette fonction s'avère très peu efficace, même pour de relativement faibles valeurs de n et p (à titre d'illustration, il faut 93 secondes à mon ordinateur pour calculer $\binom{30}{15}$). La raison en est facile à comprendre : lorsqu'on observe par exemple l'arbre de calcul de $\binom{5}{2}$ on constate que de nombreux appels récursifs sont identiques et donc superflus :



Nous pouvons par exemple constater que le calcul de $\binom{5}{2}$ nécessite de calculer deux fois $\binom{3}{1}$ et trois fois $\binom{2}{1}$.

Plus précisément, si on note $C(n,p)$ le nombre d'additions réalisées par cette fonction, on dispose des relations :

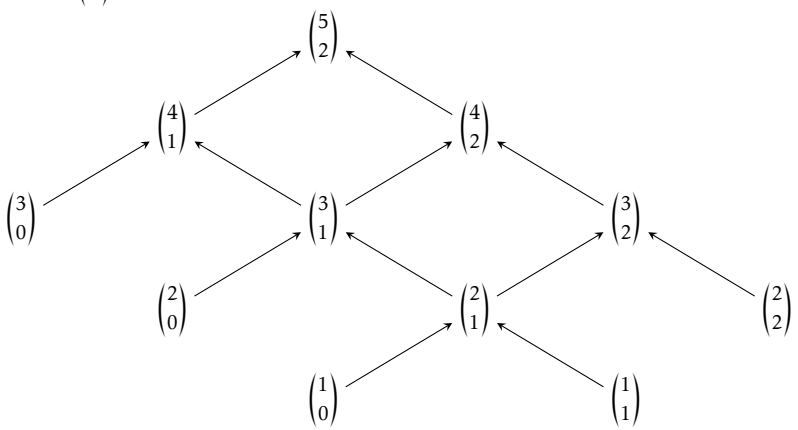
$$C(n,0) = C(n,p) = 0 \quad \text{et} \quad \forall p \in [\![1, n-1]\!], \quad C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$$

qui permettent de prouver par induction que $C(n,p) = \binom{n}{p} - 1$. Or la formule de Stirling permet d'établir l'équivalent : $\binom{2n}{n} \sim \dfrac{4^n}{\sqrt{\pi n}}$; le calcul de $\binom{2n}{n}$ par cette fonction est donc de complexité exponentielle.

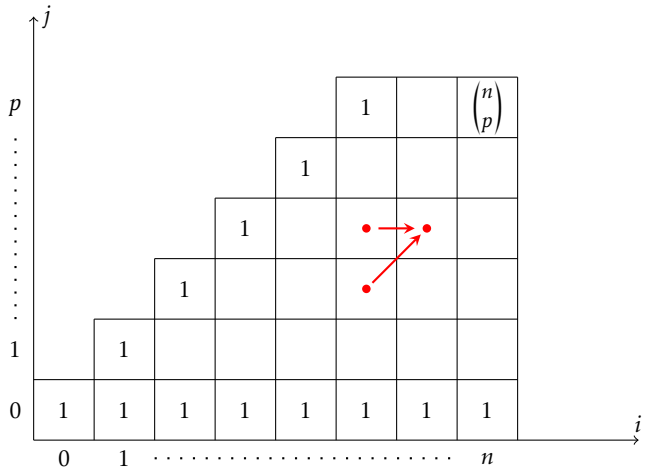
Autrement dit, lorsque le problème à résoudre (ici le calcul de $\binom{n}{p}$) se ramène à la résolution de deux sous-problèmes (le calcul de $\binom{n-1}{p-1}$ et de $\binom{n-1}{p}$) *qui sont en interaction*, la complexité peut rapidement croitre de manière rédhibitoire.

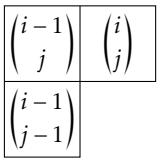
1.2 La solution proposée par la programmation dynamique

La solution proposée par la programmation dynamique consiste à commencer par résoudre les plus petits des sous-problèmes, puis de combiner leurs solutions pour résoudre des sous-problèmes de plus en plus grands. Concrètement, le calcul de $\binom{5}{2}$ se réalise en suivant le schéma suivant :



Pour réaliser ce type de solution on utilise souvent un tableau, ici un tableau bi-dimensionnel $(n+1)\times(p+1)$ (dont seule la partie pour laquelle $j \leqslant i$ sera utilisée). Ce tableau sera progressivement rempli par les valeurs des coefficients binomiaux, en commençant par les plus petits :





Il faut faire attention à bien respecter la relation de dépendance (modélisée par les flèches sur le schéma ci-dessus) pour remplir les cases de ce tableau : la case destinée à recevoir la valeur de $\binom{i}{j}$ ne peut être remplie qu'après les cases destinées à recevoir $\binom{i-1}{j-1}$ et $\binom{i-1}{j}$.

```
let binom n p =
  let c = make_matrix (n+1) (p+1) 1 in
  for i = 2 to n do
    for j = 1 to min (i-1) p do
      c.(i).(j) <- c.(i-1).(j) + c.(i-1).(j-1)
    done ;
  done ;
  c.(n).(p) ;;
```

Au prix d'un coût spatial (la création du tableau) cet algorithme est bien plus efficace que l'algorithme récursif initial puisque sa complexité temporelle et spatiale est maintenant en $\Theta(np)$.

On peut même encore améliorer la complexité spatiale de cet algorithme en observant qu'on peut se contenter d'un tableau uni-dimensionnel puisque le remplissage d'une colonne de ce tableau ne dépend que de la colonne précédente :

```
let binom n p =
  let c = make_vect (p+1) 1 in
  for i = 2 to n do
    for j = min (i-1) p downto 1 do
      c.(j) <- c.(j) + c.(j-1)
    done ;
  done ;
  c.(p) ;;
```

Cette fonction a maintenant une complexité spatiale en $\Theta(p)$ (mais toujours une complexité temporelle en $\Theta(np)$).

Mais attention à toujours bien respecter l'ordre de dépendance des cases de ce tableau : la valeur $\binom{i-1}{j}$ sert à calculer les valeurs de $\binom{i}{j+1}$ et de $\binom{i}{j}$. Lorsque la case contenant $\binom{i-1}{j}$ reçoit la valeur $\binom{i}{j}$ il faut donc que le calcul de $\binom{i}{j+1}$ ait déjà été réalisé, ce qui impose d'effectuer les calculs par ordre *décroissant* de j.

Remarque. Pour calculer ces coefficients binomiaux, il existe une autre technique qui combine l'élégance de la réponse récursive avec l'efficacité de la programmation dynamique, et qui consiste à associer à une fonction une table de valeurs qui est remplie au fur et à mesure des calculs. Ainsi, à chaque étape, le programme va voir dans la table si la valeur dont il a besoin a déjà été calculée et en fait le calcul que si ce n'est pas le cas. Cette technique porte le nom de *mémoïsation*, nous en reparlerons dans le chapitre suivant consacré aux tables d'association.

2. La programmation dynamique

2.1 Programmation dynamique et gloutonne

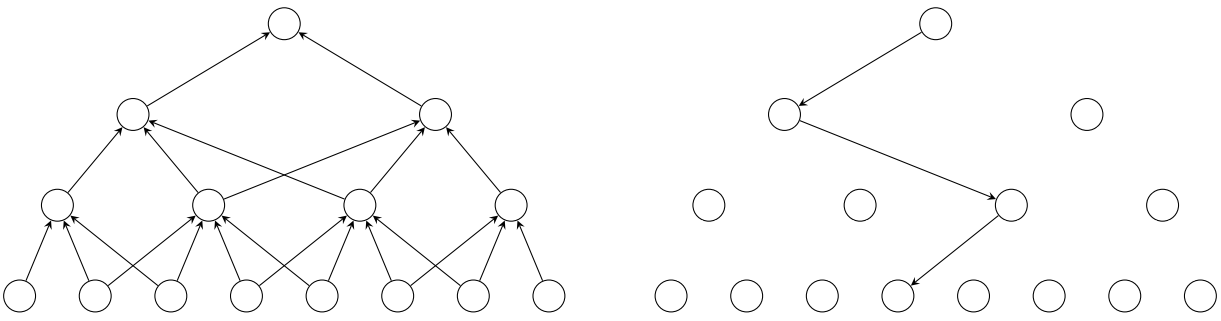
Tout comme les problèmes que l'on résout par une méthode « diviser pour régner », les problèmes que l'on résout par la programmation dynamique se ramènent à la résolution de sous-problèmes de tailles inférieures. Mais à la différence de la méthode « diviser pour régner », ces sous-problèmes *ne sont pas indépendants*, ce qui invalide le plus souvent la programmation récursive.

Concrètement, résoudre un problème par la programmation dynamique consiste à suivre les étapes indiquées ci-dessous :

(1) obtenir une relation de récurrence liant la solution du problème global à celles de problèmes locaux *non indépendants* ;

(2) initialiser d'un tableau à l'aide des conditions initiales de la relation obtenue au point précédent ;

(3) remplir ce tableau en résolvant les problèmes locaux par taille croissante, à l'aide de la relation obtenue au premier point.



programmation dynamique *programmation gloutonne*

FIGURE 1 – Une illustration des programmations dynamique et gloutonne.

Cette technique est fréquemment employée pour résoudre des problèmes d'optimisation : elle s'applique dès lors que la solution optimale peut être déduite des solutions optimales des sous-problèmes (c'est le *principe*

d'optimalité de Bellman, du nom de son concepteur). Cette méthode garantit d'obtenir la meilleure solution au problème étudié, mais dans un certain nombre de cas sa complexité temporelle reste trop importante pour pouvoir être utilisée dans la pratique.
Dans ce type de situation, on se résout à utiliser un autre paradigme de programmation, la *programmation gloutonne*. Alors que la programmation dynamique se caractérise par la résolution par taille croissante de *tous* les problèmes locaux, la stratégie gloutonne consiste à choisir à partir du problème global un problème local *et un seul* en suivant une heuristique (c'est à dire une stratégie permettant de faire un choix rapide mais pas nécessairement optimal). On ne peut en général garantir que la stratégie gloutonne détermine la solution optimale, mais lorsque l'heuristique est bien choisie on peut espérer obtenir une solution proche de celle-ci.

• Rendu de monnaie

Le problème du rendu de monnaie est un exemple typique qui oppose programmation dynamique et programmation gloutonne : comment rendre la monnaie à l'aide d'un nombre minimal de pièces et de billets ?

Une heuristique gloutonne élémentaire consiste à rendre la pièce ou de billet de valeur maximale possible, et ce tant qu'il reste quelque chose à rendre. Par exemple, la somme de 48€ sera décomposée comme suit : 20€+20€+5€+2€+1€. Mais en l'absence de preuve, nous ne pouvons garantir que cette solution soit optimale.
Nous pouvons aussi chercher à adopter une stratégie récursive. Pour cela, on note $f(n,\mathrm{S})$ le nombre minimal de pièces nécessaire pour décomposer la somme n à partir de pièces contenues dans la liste S. Si $c \in \mathrm{S}$ et si $c \geqslant n$ on peut choisir d'utiliser la pièce c, auquel cas le nombre minimal de pièces à utiliser sera égal à $1 + f(n-c,\mathrm{S})$, ou on peut choisir de ne pas utiliser la pièce c, auquel cas le nombre minimal de pièces à utiliser sera égal à $f(n,\mathrm{S}\setminus\{c\})$. Autrement dit, on dispose de la relation :

$$f(n,\mathrm{S}) = \begin{cases} \min\Big(1+f(n-c,\mathrm{S}), f(n,\mathrm{S}\setminus\{c\})\Big) & \text{si } c \leqslant n \\ f(n,\mathrm{S}\setminus\{c\}) & \text{sinon} \end{cases}$$

Les deux sous-problèmes à résoudre : $f(n-c,\mathrm{S})$ et $f(n,\mathrm{S}\setminus\{c\})$ ne sont pas indépendants ; il faudra donc appliquer une méthode de programmation dynamique plutôt qu'un algorithme récursif pour résoudre ce problème (ceci vous sera demandé dans l'exercice 3).

Remarque. Il est possible de prouver que dans le cadre du système monétaire européen, l'algorithme glouton fournit toujours la solution optimale, mais ce n'était pas le cas du système britannique d'avant 1971 qui utilisait les multiples suivant du *penny* : 1, 3, 6, 12, 24, 30. Avec ce système, l'algorithme glouton décompose $48p$ en $30p+12p+6p$ alors que la décomposition optimale est $24p+24p$.

Nous allons maintenant passer en revue quelques problèmes classiques qui font appel à la programmation dynamique.

2.2 Le problème du sac à dos

Ce problème se pose dans les termes suivants :

> *étant donnés n objets de valeurs $c_1,\dots,c_n$ et de poids respectifs $w_1,\dots,w_n$, comment remplir un sac à dos maximisant la valeur emportée $\sum_{i\in\mathrm{I}} c_i$ tout en respectant la contrainte $\sum_{i\in\mathrm{I}} w_i \leqslant \mathrm{W}_{max}$?*

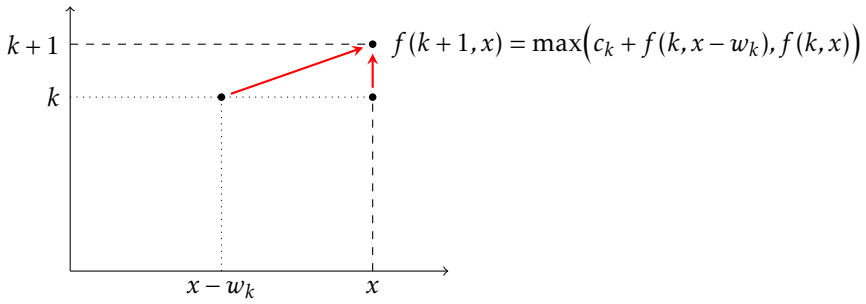
Pour le résoudre, nous allons noter $f(n,\mathrm{W}_{\max})$ la valeur maximale qu'il est possible d'atteindre avec les n objets. Si l'objet d'indice n est dans le sac à dos optimal, alors $w_n \leqslant \mathrm{W}_{\max}$ et $f(n,\mathrm{W}_{\max}) = c_n + f(n-1,\mathrm{W}_{\max}-w_n)$; s'il n'y est pas alors $f(n,\mathrm{W}_{\max}) = f(n-1,\mathrm{W}_{\max})$. On en déduit :

$$f(n,\mathrm{W}_{\max}) = \begin{cases} \max\Big(c_n + f(n-1,\mathrm{W}_{\max}-w_n), f(n-1,\mathrm{W}_{\max})\Big) & \text{si } w_n \leqslant \mathrm{W}_{\max} \\ f(n-1,\mathrm{W}_{\max}) & \text{sinon} \end{cases}$$

Cette formule se prête à merveille à une programmation dynamique : nous allons utiliser un tableau bi-dimensionnel de taille $(n+1)\times(\mathrm{W}_{\max}+1)$ destiné à contenir les valeurs de $f(k,w)$ pour $k \in [\![0,n]\!]$ et $w \in [\![0,\mathrm{W}_{\max}]\!]$. Nous prendrons bien entendu comme valeurs initiales $f(0,w) = 0$.
En considérant que les valeurs c_k et w_k sont données sous forme de vecteurs, on en déduit l'algorithme suivant :

```
let sacados c w wmax =
  let n = vect_length c in
  let f = make_matrix (n+1) (wmax+1) 0 in
  for k = 0 to n-1 do
    for x = 0 to wmax do
      if w.(k) <= x
        then f.(k+1).(x) <- max (c.(k)+f.(k).(x-w.(k))) f.(k).(x)
        else f.(k+1).(x) <- f.(k).(x)
    done
  done ;
  f.(n).(wmax) ;;
```

On peut constater que chaque ligne du tableau ne nécessite, pour être remplie, que de connaître la ligne précédente, ce qui permet de n'utiliser qu'un tableau uni-dimensionnel de taille $(W_{max}+1)$, *à condition de bien respecter l'ordre de dépendance des calculs* :



Pour qu'une ligne remplace la précédente, il est nécessaire de la remplir de la droite vers la gauche pour que la case indexée par $x - w_k$ n'ait pas encore été modifiée au moment où on remplit la case d'indice x.

```
let sacados c w wmax =
  let n = vect_length c in
  let f = make_vect (wmax+1) 0 in
  for k = 0 to n-1 do
    for x = wmax downto 0 do
      if w.(k) <= x && c.(k)+f.(x-w.(k)) > f.(x)
        then f.(x) <- c.(k)+f.(x-w.(k))
    done
  done ;
  f.(wmax) ;;
```

Il apparaît clairement que la complexité temporelle de ce dernier algorithme est proportionnel au produit nW_{max} ; il s'agit d'un $\Theta(nW_{max})$. Quant au coût spatial, il s'agit d'un $\Theta(W_{max})$.

Enfin, on peut observer que cet algorithme calcule la valeur maximale qui peut être emportée mais non pas la façon d'y parvenir. Pour la connaître, il faudrait ajouter à chaque case du tableau la liste des indices des objets à sélectionner. Ceci conduit à la version suivante :

```
let sacados c w wmax =
  let n = vect_length c in
  let f = make_vect (wmax+1) (0, []) in
  for k = 0 to n-1 do
    for x = wmax downto 0 do
      if w.(k) <= x && c.(k)+fst f.(x-w.(k)) > fst f.(x)
        then f.(x) <- c.(k)+fst f.(x-w.(k)), k::snd f.(x-w.(k))
    done
  done ;
  f.(wmax) ;;
```

2.3 Distance d'édition

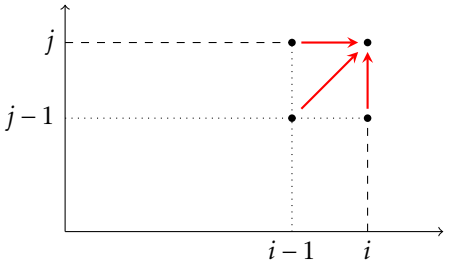
La distance d'édition, ou distance de Levenshtein, est une mesure de la similarité de deux chaînes de caractères : elle est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne de caractères à une autre.
Par exemple, on peut passer du mot **polynomial** au mot **polygonal** en suivant les étapes suivantes :

- suppression de la lettre **'i'** : **polynomial** → **polynomal** ;
- remplacement du **'n'** par un **'g'** : **polynomal** → **polygomal** ;
- remplacement du **'m'** par un **'n'** : **polygomal** → **polygonal** ;

donc la distance d'édition entre ces deux mots est égale au plus à 3, et on se convaincra aisément qu'il n'est pas possible de faire mieux.
Nous allons calculer la distance d'édition entre deux mots $a = a_1 a_2 \cdots a_m$ et $b = b_1 b_2 \cdots b_n$ en généralisant le problème, c'est à dire en définissant la distance d'édition $\mathrm{d}(i,j)$ entre les mots $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$.
Dans le chemin reliant de manière optimale $a_1 a_2 \cdots a_i$ et $b_1 b_2 \cdots b_j$, plusieurs cas de figure peuvent se rencontrer :

- a_i a été supprimé, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j)+1$;
- b_j a été ajouté, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i,j-1)+1$;
- a_i a été remplacé par b_j, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j-1)+1$;
- $a_i = b_j$, auquel cas $\mathrm{d}(i,j) = \mathrm{d}(i-1,j-1)$.

On en déduit que $\mathrm{d}(i,j) = \begin{cases} \min\big(\mathrm{d}(i-1,j), \mathrm{d}(i,j-1), \mathrm{d}(1-1,j-1)\big)+1 & \text{si } a_i \neq b_j \\ \min\big(\mathrm{d}(i-1,j)+1, \mathrm{d}(i,j-1)+1, \mathrm{d}(i-1,j-1)\big) & \text{si } a_i = b_j \end{cases}$.
Les conditions initiales sont clairement : $\mathrm{d}(i,0) = i$ et $\mathrm{d}(0,j) = j$, ce qui conduit au schema de dépendance suivant :



puis à l'algorithme :

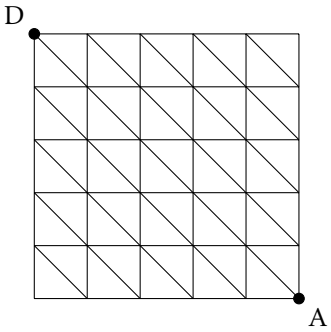
```
let dist a b =
  let m = string_length a and n = string_length b in
  let d = make_matrix (m + 1) (n + 1) 0 in
  for i = 1 to m do
    d.(i).(0) <- i
  done ;
  for j = 1 to n do
    d.(0).(j) <- j
  done ;
  for i = 1 to m do
    for j = 1 to n do
      d.(i).(j) <- min d.(i).(j-1) d.(i-1).(j) ;
      match a.[i-1] = b.[j-1] with
        | false -> d.(i).(j) <- min d.(i).(j) d.(i-1).(j-1) + 1
        | true -> d.(i).(j) <- min (d.(i).(j) + 1) d.(i-1).(j-1)
    done
  done ;
  d.(m).(n) ;;
```

Cette fonction a une complexité temporelle et spatiale en $\Theta(mn)$. Compte tenu du schema de dépendance, il est possible de faire passer le coût spatial en $\Theta(n)$ en ne mémorisant qu'une ligne (la rédaction de l'algorithme correspondant est laissé au lecteur).

3. Exercices

Exercice 1 Partant du coin supérieur gauche d'une grille 5×5, combien y-a-t'il de chemin menant au coin inférieur droit en ne suivant que les trois directions suivantes ?

les trois directions possibles



Et pour une grille 20×20 ?

Exercice 2 *Project Euler, problem 67.*

En partant du sommet du triangle ci-dessous et en se déplaçant vers les nombres adjacents de la ligne inférieure, le total maximum que l'on peut obtenir pour relier le sommet à la base est égal à 23 :

$$\begin{array}{ccccccc} & & & \mathbf{3} & & & \\ & & \mathbf{7} & & 4 & & \\ & 2 & & \mathbf{4} & & 6 & \\ 8 & & 5 & & \mathbf{9} & & 3 \end{array}$$

$$3 + 7 + 4 + 9 = 23$$

Rédiger une fonction calculant le total maximum d'un chemin reliant le sommet à la base d'un tel triangle de hauteur n. On pourra considérer que les valeurs de ce triangle sont stockées dans un tableau bi-dimensionnel $n \times n$ (autrement dit, `t.(i).(j)` contient la $(j+1)^{\text{e}}$ valeur de la $(i+1)^{\text{e}}$ ligne).

Exercice 3 on considère un système de monnaie utilisant p types de pièces différentes de valeurs respectives $S = \{c_1, c_2, \ldots, c_p\}$. pour tout $n \in \mathbb{N}$, on note $f(n, S)$ le nombre minimal de pièces nécessaires pour décomposer l'entier n dans ce système de monnaie.
Établir une relation de récurrence vérifiée par $f(n, S)$ et en déduire un algorithme dynamique permettant le calcul de cette quantité.
Modifier ensuite votre algorithme pour afficher une décomposition optimale.

Exercice 4 Si A et B sont deux matrices de tailles respectives $a \times b$ et $c \times d$, le produit AB n'est possible que si $b = c$, et dans ce cas la matrice produit AB est de taille $a \times d$, et peut être calculée à l'aide de acd multiplications.

Sachant que le produit matriciel est associatif mais pas commutatif, le produit ABC peut être calculé de deux manières : (AB)C ou A(BC), qui ne nécessitent pas *a priori* le même nombre de multiplications. Par exemple, si A est de taille 10×100, B de taille 100×5, et C de taille 5×50, le produit (AB)C nécessite $5000 + 2500 = 7500$ multiplications et le produit A(BC), $25000 + 50000 = 75000$ multiplications.

On considère une chaîne de matrices $M_0 M_1 M_2 \cdots M_{n-1}$ de tailles respectives $m_0 \times m_1$, $m_1 \times m_2$, …, $m_{n-1} \times m_n$. Rédiger une fonction calculant le nombre minimal de multiplications nécessaires pour effectuer ce produit matriciel. On pourra considérer que les valeurs de $m_0, m_1, \ldots, m_n$ sont rangées dans un tableau de taille $n+1$.

Quelle valeur obtient-on lorsque $n = 50$ et $m_k = k + 1$?

Exercice 5 On considère une matrice $A \in \mathcal{M}_{m,n}(\{0,1\})$ à m lignes et n colonnes dont les coefficients sont tous égaux à 0 ou 1 et on cherche à déterminer la taille maximale k d'un carré de 1 dans A. Par exemple, la matrice dessinée ci-dessous présente un carré de 1 de taille 4 :

1	0	0	0	0	0	0	1	0
1	0	1	1	1	1	1	1	0
1	0	1	1	1	1	0	0	0
0	1	1	1	1	1	0	1	0
0	0	1	1	1	1	0	0	1
0	0	1	1	1	0	0	1	1
0	1	0	0	0	0	0	1	1

Définir un algorithme résolvant ce problème.

♦ ♦

♦