Programmation dynamique
(Résumé du chapitre 3 du manuel)

Table des matières

3 Programmation dynamique

– On a vu au chapitre précédent une approche *descendante* et récursive à la résolution d'un problème :

- On *décompose* le problème à résoudre en sous-problèmes *plus simples* (similaires au problème initial).
- On trouve, récursivement, la solution des sous-problèmes (sauf si le problème est trivial, auquel cas on calcule directement la solution).
- On *combine* les solutions des sous-problèmes pour obtenir la solution du problème initial.

Cette stratégie récursive produit généralement des algorithmes clairs et facilement compréhensibles. Toutefois, pour que l'algorithme résultant soit efficace, une condition majeure doit être satisfaite, à savoir que les divers sous-problèmes générés soient *indépendants* les uns des autres. En d'autres mots, une approche diviser-pour-régner avec récursivité peut conduire à des algorithmes inefficaces lorsque la décomposition récursive conduit à résoudre *plusieurs fois* un même sous-problème — on dit aussi lorsqu'il y a présence de *sous-problèmes superposés*.

– Un exemple classique est le calcul des nombres de Fibonacci, que nous examinerons plus en détail à la prochaine section. Nous verrons alors que, à l'aide de structures de données appropriées, il est possible d'obtenir un algorithme plus efficace en *évitant* de recalculer des solutions à des sous-problèmes déjà rencontrés. Dans la présentation classique de la programmation dynamique, la structure de données généralement utilisée est un tableau, lequel est construit/rempli de façon *ascendante*. Par contre, nous verrons qu'une approche récursive et *descendante* peut aussi être utilisée pour éviter de résoudre plusieurs fois un même sous-problème.

– Notons que cette "approche" de la programmation dynamique n'est pas présentée dans tous les manuels qui traitent de conception et d'analyse d'algorithmes ... parce que ces références reposent généralement sur le paradigme impératif et procédural de programmation. Par contre, dans le cadre du paradigme de *programmation fonctionnelle*, plus précisément dans les langages fonctionnels non stricts, cette façon différente d'aborder la programmation dynamique est fondamentale, sinon *nécessaire*. Soulignons par contre que Cormen, Leiserson et Rivest présentent cette approche, qu'ils appellent plutôt la stratégie récursive avec *recensement*.

3.0 Nombres de Fibonacci

Note : Cette section n'est pas présente dans le manuel.

3.0.1 Solution récursive simple

```
PROCEDURE fib( n: Nat ): Nat
DEBUT
   SI n == 0 || n == 1 ALORS
     RETOURNER 1
   SINON
     RETOURNER fib(n-1) + fib(n-2)
   FIN
FIN
```

Algorithme 1: Solution purement récursive du calcul du $n$ième nombre de Fibonacci

L'algorithme 1 présente une solution purement récursive du calcul du $n$ième nombre de Fibonacci.

Le temps d'exécution est décrit par les équations de récurrence suivantes :

- $T(n) = T(n-1) + T(n-2) + \Theta(1), \text{si } n > 1$
- $T(1) = \Theta(1)$
- $T(0) = \Theta(1)$

La solution de ces équations permet de conclure que le temps d'exécution est au moins exponentiel. Plus précisément, ceci peut être montré comme suit. Tout d'abord, notons que la fonction $T(n)$ donnant le temps d'exécution est monotonique, c'est-à-dire $n_1 \leq n_2 \Rightarrow T(n_1) \leq T(n_2)$ (c'est généralement le cas de telles fonctions). Soit alors c et d les constantes associées, respectivement, au travail de complexité $\Theta(1)$ de $T(n)$ et $T(0)$. Supposons aussi, pour simplifier l'analyse, que n est un nombre pair ($n = 2 * k$ pour un certain k). On a alors les relations suivantes :

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c \\
&\geq T(n-2) + T(n-2) + c \\
&= 2T(n-2) + c \\
&\geq 2[2T(n-4) + c] + c \\
&= 2^2 T(n - 2*2) + 2*c \\
&\geq 2^2[2T(n - 2*2 - 2) + c] + 2*c \\
&= 2^3 T(n - 2*3) + 3*c \\
&\dots \\
&\geq 2^k T(n - 2*k) + k*c \\
&= 2^k T(0) + k*c \\
&= 2^k * d + k*c \\
&= 2^{n/2} * d + n/2 * c \\
&\geq 2^{n/2} * d
\end{aligned}$$

On en conclut donc que $T(n) \in \Omega(2^{n/2}) = \Omega((\sqrt{2})^n)$. [1]

[1] En fait, il est possible de montrer, mais nous ne le ferons pas ici, que $T(n) \in \Theta(r^n)$ où $r = \frac{1+\sqrt{5}}{2}$ est appelé le nombre d'or.

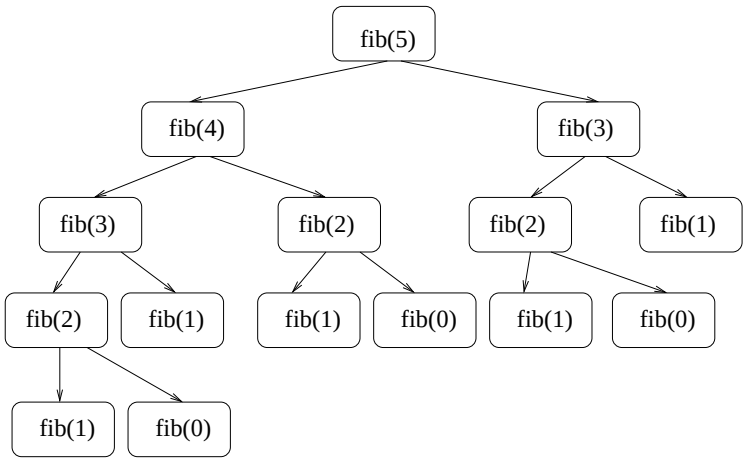

Figure 1: Présence d'appels superposés pour le calcul de `fib(5)`

Cet algorithme exponentiel, donc, est inefficace, et ce à cause de la présence de *sous-problèmes superposés*, c'est-à-dire qu'un appel pour une valeur de `n` donnée peut être effectué plusieurs fois, tel que cela est illustré à la Figure 1, qui représente l'arbre des appels pour `fib(5)`.

3.0.2 Solution avec "*mémorisation*" (programmation dynamique descendante)

L'algorithme 2 présente une solution récursive avec mémorisation du calcul du $n$ième nombre de Fibonacci. Un tableau auxiliaire `A` est utilisé pour *mémoriser* les appels récursifs ayant déjà été effectués, c'est-à-dire, pour prendre en note les arguments pour lesquels un appel a déjà été effectué et, donc, pour lesquels on a déjà calculé le résultat correspondant. En d'autres mots, le tableau `A` joue un rôle semblable à celui d'une *mémoire cache*.

Pour analyser le temps d'exécution de cette solution avec mémorisation, il faut tout d'abord réaliser que l'arbre des appels récursifs *ne sera pas du tout le même que celui présenté à la figure 1.* La raison en est qu'un appel `fib'(n, A)` ne va générer d'autres appels récursifs *que si c'est la première fois que l'argument* `n` *est rencontré.* En d'autres mots, certains des sous-arbres de la figure 1 *seront élagués* (tronqués).

L'arbre des appels récursif pour `fib'(n, A)` sera tel qu'illustré à la figure 2. Dans cette figure, on suppose tout d'abord que les appels récursifs se font dans l'ordre indiqué dans le code de l'algorithme 2, c'est-à-dire, l'appel définissant `r1` suivi de l'appel définissant `r2`. Ensuite, les appels `fib'(n, A)` qui sont à l'intérieur de rectangles aux coins arrondis représentent des appels qui correspondent à l'évaluation d'un argument `n` qui n'a pas encore été rencontré, donc dont l'évaluation va générer d'autres appels à `fib'` *et* qui va mettre à jour `A[n]` avec la valeur appropriée (deuxième partie de la procédure `fib'`). Par contre, les appels `fib'(n, A)` qui ne sont pas à l'intérieur de tels rectangles correspondent à des cas où la valeur `n` a déjà été évaluée (`A[n] != PAS_DEFINI`) et où le résultat peut être retourné de façon immédiate : dans tous les cas, le deuxième appel à `fib'` (celui définissant `r2`) va toujours correspondre à un appel de ce type, sauf pour le cas de base `fib'(0, A)`.

Dans le but de faciliter l'analyser de l'algorithme, chacun des arcs dénotant un appel a été annoté par un nombre indiquant le *numéro* de l'appel auquel il correspond. Ainsi, une valeur de `i` sur un arc indique qu'il s'agit du `i`ème appel du type indiqué (générant d'autres appels ou retournant un résultat de façon immédiate). On constate ainsi que le nombre total d'appels à `fib'` sera d'ordre $\Theta(n)$ — `n` appels du premier type, `n-2` appels du deuxième type.

```
PROCEDURE fib( n: Nat ): Nat
DEBUT
   A <- new Nat[0:n]
   POUR i <- 0 A n FAIRE
     A[i] <- PAS_DEFINI // Valeur spéciale (par ex., -1) ne pouvant pas être un résultat.
   FIN
   RETOURNER fib'( n, A )
FIN

PROCEDURE fib'( n: Nat, A: ARRAY[*] OF Nat ): Nat
DEBUT
   SI A[n] != PAS_DEFINI ALORS
     // L'appel fib(n) a déjà été calculé.
     RETOURNER A[n]
   FIN

   // Premier appel à fib(n).
   SI n == 0 || n == 1 ALORS
     r <- 1
   SINON
     r1 <- fib'( n-1, A )
     r2 <- fib'( n-2, A )
     r  <- r1 + r2
   FIN
   A[n] <- r
   RETOURNER r
FIN
```

Algorithme 2: Solution récursive avec mémorisation (utilisant un tableau) pour le calcul du $n$ième nombre de Fibonacci

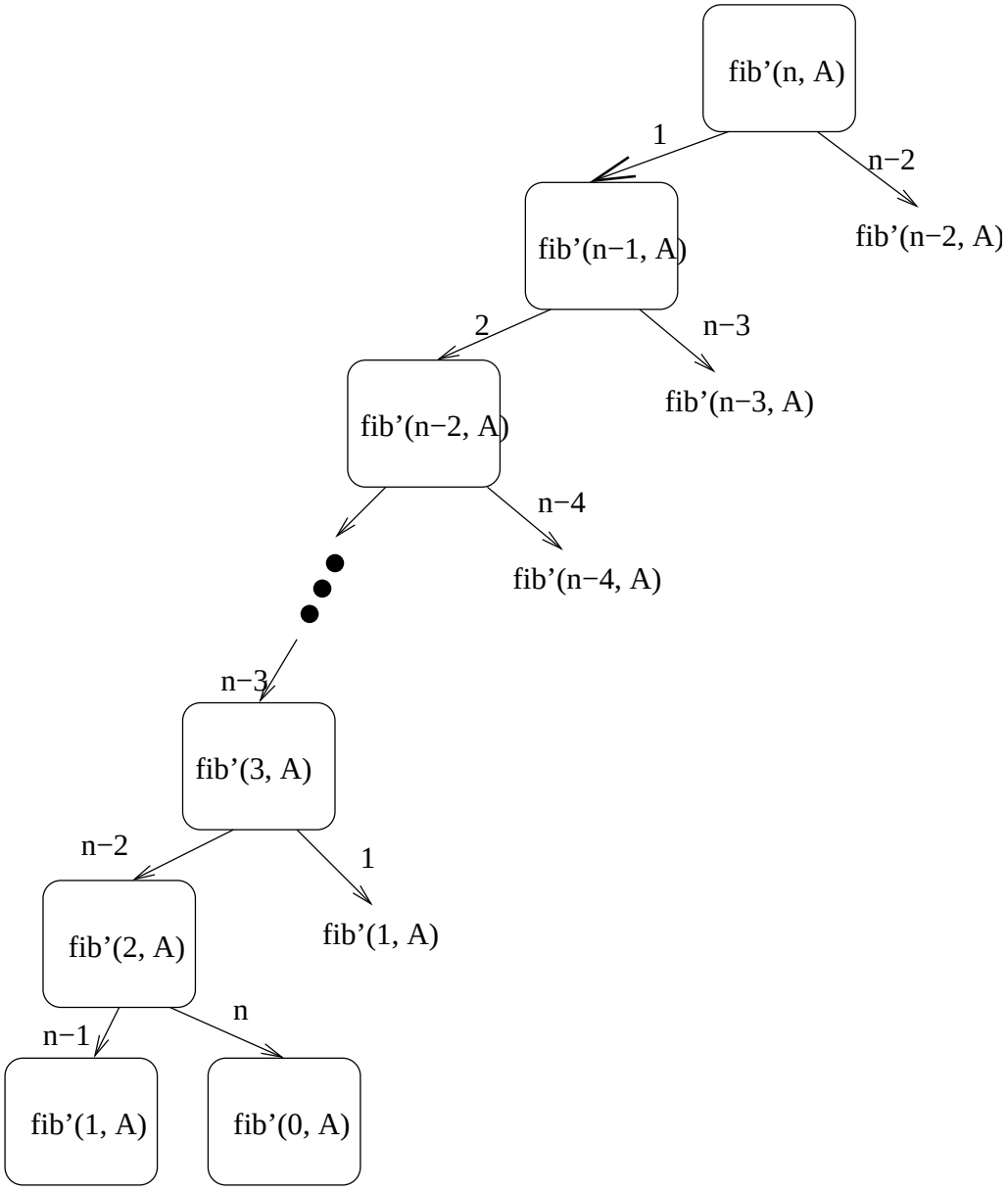

Figure 2: Arbre des appels récursifs en présence de mémorisation pour `fib'(n, A)`

On a donc l'analyse suivante pour l'algorithme `fib` :

- La boucle `POUR` pour l'initialisation de `A` dans la procédure `fib` est $\Theta(n)$.
- Le travail effectué dans chacune des instances de `fib'`, *si on ignore le travail effectué dans les appels récursifs* (puisqu'il sera compté dans l'ensembles des appels), est $\Theta(1)$. Or, comme on l'a vu précédemment, on aura $\Theta(n)$ appels de la procédure `fib'`. Le temps total pour exécuter l'ensemble des appels à la procédure `fib'` sera donc $\Theta(n)$.

De ces deux points, on peut donc conclure que l'algorithme `fib` est de complexité linéaire $\Theta(n)$.

3.0.3 Méthode de programmation dynamique plus classique

```
>> >> >> >> >> A[1] = 1
>> >> >> >> >> A[0] = 1
>> >> >> >> A[2] = 2
>> >> >> A[3] = 3
>> >> A[4] = 5
>> A[5] = 8
```

Figure 3: Ordre de définition des éléments de `A` pour un appel à `fib(5)` dans l'algorithme 2

La prochaine étape pour en arriver à la version classique de la programmation dynamique consiste à construire le tableau `A` *sans utiliser* de récursivité, c'est-à-dire, à remplir ce tableau de façon *ascendante* (plutôt que *descendante*, comme on le fait avec la récursivité), et ce dans un ordre qui assure que les positions du tableau qui sont requises pour calculer la valeur à une position donnée *ont déjà été calculées.* La Figure 3 présente une trace de l'ordre dans lequel les éléments de `A` sont définis pour un appel à `fib(5)`, de même que le niveau de récursion atteint pour définir cet élément ; tous les autres appels récursifs n'apparaissant pas dans cette trace correspondent donc à des appels pour lesquels `A[n] != PAS_DEFINI` était vrai, c'est-à-dire, pour lesquels le résultat correspondant pour `n` était déjà calculé.

Dans la méthode classique de programmation dynamique, on construit donc une table qui contient les solutions aux sous-problèmes intermédiaires. De plus, cette table est construite de façon à ce qu'on soit assuré que la solution d'un sous-problème soit déjà calculée et présente dans la table au moment où elle est doit être utilisée. En d'autres mots, si on se réfère à l'arbre des appels récursifs, on construit la table de façon *ascendante* — c'est-à-dire, des problèmes simples *vers* les problèmes complexes.

```
PROCEDURE fib(  n: Nat ): Nat         DEBUT
   A <- new Nat[n+1]           // Table pour les solutions intermédiaires.
   A[0] <- 1                   // Initialisation pour les problèmes de base.
   A[1] <- 1
   POUR i <- 2 A n FAIRE
       // Construction (ascendante) de la table.
       A[i] <- A[i-1] + A[i-2]
   FIN
   RETOURNER A[n]
FIN
```

Algorithme 3: Solution non récursive avec tableau construit de façon ascendante (programmation dynamique classique) du calcul du $n$ième nombre de Fibonacci

L'algorithme 3 présente une solution classique de calcul du $n$ième nombre de Fibonacci à l'aide d'une approche de programmation dynamique. Il est alors facile de voir que le temps d'exécution sera de complexité $O(n)$.

3.1 Coefficients binomiaux

Voir la dernière section sur la programmation fonctionnelle et la programmation dynamique.

3.2 Algorithme de Floyd pour le calcul du plus court chemin

Section omise.

3.3 Programmation dynamique et problèmes d'optimisation

Les algorithmes basés sur la programmation dynamique sont souvent utilisés pour résoudre des problèmes *d'optimisation*, c'est-à-dire des problèmes dans lesquels l'objectif est de d'identifier une ou plusieurs solutions qui satisfont certains critères *d'optimalité*, donc minimisant (ou maximisant, selon le cas) une certaine fonction de coût.

La stratégie pour utiliser la programmation dynamique *ascendante* repose alors sur les principales étapes suivantes :

1. Identifier les caractéristiques d'une solution optimale (à l'aide d'une fonction de coût appropriée).
2. Définir récursivement la solution optimale à une instance du problème en identifiant les divers sous-problèmes à résoudre (par exemple, à l'aide d'équations récursives appropriées ou à l'aide d'une approche récursive diviser-pour-régner).
3. Identifier une structure de données auxiliaire qui permettra de mémoriser les solutions aux sous-problèmes et déterminer de quelle façon cette structure de données devra être remplie pour permettre de calculer la valeur d'une solution optimale de façon *ascendante*, c'est-à-dire en remontant progressivement des sous-problèmes simples jusqu'au problème global initial.

   Ici, on parle aussi de déterminer le *schéma de remplissage* de la structure de données. Dans le cas d'un algorithme de programmation dynamique itératif, c'est essentiellement ce schéma de remplissage qui va déterminer la structure de l'algorithme. Le principal travail de l'algorithme consiste alors à remplir et parcourir la structure de données de sorte que l'on ne demande jamais un résultat qui n'a pas encore été calculé ; en d'autres mots, il faut donc examiner les *dépendances de données* entre les éléments de la structure de données pour déterminer le schéma de remplissage.
4. Si désiré, construire *une*[2] solution optimale à partir des informations calculées.[3]

Toutefois, ce ne sont pas tous les problèmes d'optimisation qui peuvent être résolus à l'aide de la programmation dynamique. Le problème à résoudre doit satisfaire le *principe d'optimalité* pour que la solution obtenue par la programmation dynamique soit optimale.

Définition 1 (Principe d'optimalite) *Le* principe d'optimalité *s'applique à un problème si une solution optimale à une instance du problème peut toujours être obtenue à partir de solutions optimales aux sous-problèmes.*

[2] Il arrive fréquemment qu'il y ait plusieurs solutions optimales possibles. Il s'agit donc généralement d'identifier *une* de ces solutions, pas nécessairement *la* solution.

[3] Notons que pour simplifier la présentation des algorithmes, dans plusieurs des exemples que nous présentons, nous nous contentons de déterminer le coût de cette solution, sans nécessairement identifier la solution elle-même.

Si le principe d'optimalité s'applique, alors on peut utiliser la programmation dynamique pour construire une solution optimale parce que, récursivement et de façon ascendante, chaque sous-solution sera optimale et qu'il suffit d'identifier une solution arbitraire.

Cormen, Leiserson et Rivest expriment le principe d'optimalité en disant qu'un problème *fait apparaître une* **sous-structure optimale** si une solution optimale au problème comporte des solutions optimales aux sous-problèmes.

Un problème est donc un bon candidat pour une solution basée sur la programmation dynamique si les deux conditions suivantes sont satisfaites :

1. Le problème fait apparaître **une sous-structure optimale** ;
2. Le problème contient **des sous-problèmes superposés** (c'est-à-dire que l'algorithme récursif génère plusieurs fois les mêmes sous-problèmes).

Si la première condition n'est pas satisfaite, la solution obtenue par un algorithme de programmation dynamique pourrait alors *ne pas être optimale*. Par contre, si la deuxième condition n'est pas satisfaite, l'utilisation d'un algorithme de programmation dynamique pourrait alors être considérée comme de l'*overkill*, en ce sens qu'on construirait un tableau des solutions intermédiaires, mais que chacune de ces solutions ne serait utilisée qu'une seule et unique fois.

Parenthèse 1: Applicabilité de la programmation dynamique selon Cormen *et al.*

3.4 Multiplications de chaînes de matrices

– Problème = multiplier une chaîne de matrices en minimisant le nombre total d'opérations. Plus précisément, étant donné une suite de matrices $A_1, \ldots, A_n$, il faut trouver *l'ordre optimal* dans lequel les produits de matrices devraient s'effectuer pour minimiser le nombre total d'opérations. Il faut évidemment respecter l'ordre des matrices dans la suite, mais on peut varier l'ordre des opérations en jouant avec l'*associativité*, c'est-à-dire, en parenthésant de différentes façons les divers produits.

– Exemple : Soit les quatre matrices suivantes (la ligne du bas indique la taille de la matrice) :

$$\begin{array}{ccccccc} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{array}$$

– De façon générale, le produit d'une matrice de taille $i \times j$ par une matrice $j \times k$ nécessite $i \times j \times k$ multiplications de base. Différentes façons de parenthéser les multiplications demanderont donc des nombres totaux de multiplications différents :

$$\begin{array}{lcll} A(B(CD)) & : & 30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 & = 3680 \\ ((AB)C)D & : & 20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 & = 10320 \\ (A(BC))D & : & 2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 & = 3120 \end{array}$$

– Note : L'objectif de l'algorithme à développer n'est *pas* de multiplier les matrices mais simplement de déterminer l'ordre optimal de multiplications. Les arguments de l'algorithme sont donc uniquement les tailles des matrices.

```
PROCEDURE MinMult( d: sequence{Nat} ): Nat
ARGUMENTS
  Les matrices à multiplier sont A_1, ..., A_n (non requises).
  La taille de la matrice A_i est donnée par d_{i-1} × d_i.
PRECONDITION
  d = [d_0, d_1, ..., d_{n-1}, d_n]
DEBUT
  n <- length(d)-1
  RETOURNER MinMult'( d, 1, n )
FIN

PROCEDURE MinMult'( d: sequence{Nat}, i, j: Nat ): Nat
PRECONDITION
   i <= j  &  j < length(d)
DEBUT
  SI i == j ALORS
    // Cas trivial.
    RETOURNER 0
  FIN

  // Cas complexe.
  minMulij <-  +∞  // On va chercher le nombre minimum requis d'ops.
  POUR k <- i A j-1 FAIRE
    // Diviser : on décompose en deux (2) sous-problèmes qu'on résout récursivement.
    nbMulik <- MinMult'( d, i, k )     // Premier sous-problème récursif.
    nbMulkj <- MinMult'( d, k+1, j )  // Deuxième sous-problème récursif.

    // Combiner : on connaît le nombre de multiplications requises pour les
    // deux sous-problèmes. On les combine (avec +) en ajoutant
    // aussi le nombre d'opérations requises pour multiplier
    // les deux matrices qui résultent de cette décomposition.
    nbMulikj <- nbMulik + nbMulkj + (d[i-1] * d[k] * d[j])

    // On vérifie si ce nouveau résultat est inférieur au minimum.
    minMulij <- MIN{ minMulij, nbMulikj }
  FIN

  // On a fait la division-combinaison pour tous les k possibles
  // et on a trouvé le minimum : on retourne ce minimum comme résultat global.
  RETOURNER minMulij
FIN
```

Algorithme 4: Algorithme récursif pour minimiser le nombre de multiplications d'une série de matrices

3.4.1 Solution diviser-pour-régner pure (récursive)

– L'algorithme 4 utilise la méthode diviser-pour-régner récursive pour résoudre le problème du minimisation du nombre d'opérations pour la multiplication d'une chaîne de matrices.

– Pour résoudre le problème de trouver le nombre optimal de multiplications requises pour multiplier les matrices A_1, ..., A_n, on va calculer, pour tous les k possibles entre 1 et n, le nombre optimal de multiplications pour faire le produit avec le parenthésage $(A_1..A_k) \times (A_{k+1}..A_n)$. On choisira ensuite le k qui minimise le nombre total de multiplications (par applications répétées de la stratégie diviser-pour-régner avec différents k, puis en recherchant la solution minimale parmi les diverses solutions obtenues).

```
MinMult'(1, 5)
>  k = 1
   MinMult'(1, 1)
   MinMult'(2, 5)
   >>  k = 2
       MinMult'(2, 2)
       MinMult'(3, 5)
       >>>  k = 3
            MinMult'(3, 3)
            MinMult'(4, 5)
            >>>> k = 4
                 MinMult'(4, 4)
                 MinMult'(5, 5)
       >>> k = 4
           MinMult'(3, 4)
           >>>> k = 3
                MinMult'(3, 3)
                MinMult'(4, 4)
           MinMult'(5, 5)
   >> k = 3
      MinMult'(2, 3)
      >>> k = 2
          MinMult'(2, 2)
          MinMult'(3, 3)
      MinMult'(4, 5)
      >>> k = 4
          MinMult'(4, 4)
          MinMult'(5, 5)
   >> k = 4
      MinMult'(2, 4)
         ...
      MinMult'(4, 5)
         ...
>  k = 2
    MinMult'(1, 2) ...
    MinMult'(3, 5) ...
>  k = 3
    MinMult'(1, 3) ...
    MinMult'(4, 5) ...
>  k = 4
    MinMult'(1, 4) ...
    MinMult'(5, 5) ...
```

Figure 4: Exemple d'exécution pour `MinMult'(1, 5)` récursif

– Par exemple, supposons qu'on désire multiplier les matrices `A1`, `A2`, `A3`, `A4`, `A5`. Avec la méthode diviser-pour-régner naïve indiquée plus haut, les appels récursifs à `MinMult'`

seraient alors ceux indiqués à la Figure 4. On voit clairement que certains de ces appels vont inévitablement se répéter (comme Fibonacci), rendant l'algorithme inefficace.

Note: Le nombre de "`>`" devant une expression "`k = ?`" indique le niveau de la récursion. Les deux appels `MinMult'` sous une telle expression sont ceux faits pour cette valeur de `k` donnée.

Il est possible de prouver (mais nous ne le ferons pas ici) que la complexité de l'algorithme résultant est exponentielle. Plus précisément, le nombre total de parenthésages pour n matrices peut être décrit par les équations de récurrence suivantes (Cormen et al., 1994), k représentant les différents points de coupure de la séquence de matrices :

- $P(1) = 1$
- $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$ si $n \geq 2$

La solution de cette équation de récurrence est $P(n) = C(n-1)$, où $C(n)$ est le $n$ième élément de la séquence des *nombres de Catalan* définie comme suit :

$$\begin{aligned} C(n) &= \frac{1}{n+1} C_{2n}^{n} \\ &\in \Omega(4^n / n^{\frac{3}{2}}) \end{aligned}$$

3.4.2 Solution récursive avec mémorisation

– L'algorithme 5 est une solution récursive descendante *avec mémorisation* pour la solution au problème du minimisation du nombre d'opérations pour la multiplication d'une chaîne de matrices.

Comme dans l'exemple du calcul du $n$ième nombre de Fibonacci (Section 3.0.2), la complexité de cet algorithme avec mémorisation est évidemment inférieure à celle de l'algorithme purement récursif, puisque pour un couple donné d'arguments `i` et `j`, le travail associé n'est effectué *qu'une seule fois*.

Comme dans l'exemple des nombres de Fibonacci, la complexité de l'algorithme peut être déterminé en analysant le nombre total d'appels distincts à `MinMult'` dans lesquels le calcul et l'affectation à `M[i][j]` sont effectués : la mémorisation des arguments déja vus va alors assurer que chaque combinaison d'arguments ne générera ce travail de calcul du minimum et d'affectation qu'au plus une fois.

Étant donné le problème initial avec les bornes `1` et `n`, le nombre total d'instances *distinctes* de la fonction `MinMult'` dans lesquelles le calcul de `minMulij` sera effectué est $\Theta(n^2)$ (`MinMult'(i, j)` pour `i` et `j` variant de 1 à `n`). Or, à chaque fois que le traitement pour une paire d'arguments `i` et `j` doit être effectué parce que son résultat n'a pas encore été calculé, on effectue (dans le pire cas) un travail de complexité $\Theta(n)$, à savoir, les diverses itérations de la boucle `POUR`. On peut donc en conclure que la complexité du temps d'exécution de `MinMult'` est $\Theta(n^3)$. La boucle d'initialistion dans la procédure `MinMult` étant $\Theta(n^2)$, on a donc bien que cet algorithme est de complexité cubique $\Theta(n^3)$.

```
PROCEDURE MinMult( d: sequence{Nat} ): Nat
ARGUMENTS
  Les matrices à multiplier sont A_1, ..., A_n.
  La taille de la matrice A_i est d_{i-1} × d_i.
PRECONDITION
  d = [d_0, d_1, ..., d_{n-1}, d_n]
DEBUT
   n <- length(d)-1
   M <- new Nat[n][n]
   POUR i <- 1 A n, j <- 1 A n FAIRE
     M[i][j] <- PAS_DEFINI
   FIN
   RETOURNER MinMult'( d, 1, n, M )
FIN

PROCEDURE MinMult'(
  d: sequence{Nat},
  i, j: Nat,
  M: ARRAY[n][n] OF Nat
  ): Nat
DEBUT
   SI M[i][j] != PAS_DEFINI ALORS
     RETOURNER M[i][j]
   FIN
   // Premier appel avec les arguments i et j.
   SI i == j ALORS
     minMulij <- 0
   SINON
     minMulij <-  +∞
     POUR k <- i A j-1 FAIRE
       nbMulik <- MinMult'( d, i, k, M )
       nbMulkj <- MinMult'( d, k+1, j, M )
       nbMulikj <- nbMulik + nbMulkj + (d[i-1] * d[k] * d[j])
       minMulij <- MIN{ minMulij, nbMulikj }
     FIN
   FIN
   M[i][j] <- minMulij
   RETOURNER minMulij
FIN
```

Algorithme 5: Algorithme récursif avec *mémorisation* pour minimiser le nombre de multiplications d'une série matrices

```
>> M[1][1] = 0
>> >> M[2][2] = 0
>> >> >> M[3][3] = 0
>> >> >> >> M[4][4] = 0
>> >> >> >> M[5][5] = 0
>> >> >> M[4][5] = 16
>> >> >> M[3][4] = 16
>> >> M[3][5] = 32
>> >> M[2][3] = 32
>> >> M[2][4] = 24
>> M[2][5] = 32
>> M[1][2] = 16
>> M[1][3] = 48
>> M[1][4] = 28
M[1][5] = 36
```

Figure 5: Ordre de définition des éléments de `M` pour `MinMult'(d, 1, 5, M)`

Ordre de définition et utilisation des éléments de `M`

La Figure 5 illustre l'ordre dans lequel les différents éléments de `M` sont définis pour un appel à `MinMult'(d, 1, 5, M)`. On constate évidemment que l'on ne doit calculer que les positions de la matrice `M` telles que `i <= j`. On constate aussi que si on calcule les éléments de `M` dans l'ordre suivant, on respectera les dépendances de données appropriées :

1. On calcule tous les `M[i][j]` tels que `j-i = 0`
2. On fait de même pour les `M[i][j]` tels que `j-i = 1`
3. Idem pour les `M[i][j]` tels que `j-i = 2`
4. etc.

Cette caractéristique sera utilisée ultérieurement dans l'algorithme de programmation dynamique classique (algorithme 6).

3.4.3 Solution (du manuel) avec programmation dynamique classique

– Notons par $M[i][j]$ le nombre minimum de multiplications requis pour multiplier les matrices A_i jusqu'à A_j, lorsque $i < j$. La propriété récursive devant être satisfaite par la solution optimale sera donc la suivante :

$$\begin{array}{lcll} M[i][i] & = & 0, & \text{pour } 1 \leq i \leq n \\ M[i][j] & = & \min\limits_{i \leq k \leq j-1} (M[i][k] + M[k+1][j] + d_{i-1} d_k d_j), & \text{pour } i < j \end{array}$$

– Principe d'optimalité : Étant donnée une solution optimale (par ex., $A_1((((A_2A_3)A_4)A_5)A_6)$) pour une suite de six matrices), alors n'importe quelle sous-solution (par ex., $(A_2A_3)A_4$) devra elle aussi être optimale. Ceci peut être prouvé par *contradiction* : s'il existait un parenthésage plus économique pour une sous-expression (par ex., $A_2(A_3A_4)$), alors il suffirait de remplacer cette sous-expression dans la parenthésage global pour obtenir une meilleure solution (par ex., $A_1(((A_2(A_3A_4))A_5)A_6)$).

– Le principe d'optimalité étant valable et ayant définit une propriété caractérisant la solution optimale, il reste ensuite à définir un algorithme qui permettra de construire de façon ascendante la solution. Par la façon dont les équations récursives sont définies, on peut constater

Principe d'optimalité plus en détails

Supposons que l'on ait une solution optimale, c'est-à-dire, un parenthésage optimal pour les n matrices $A_1, \ldots, A_n$. On désire alors montrer que n'importe quelle sous-solution utilisée dans cette solution optimale sera elle aussi optimale. En d'autres mots, il nous faut montrer que les solutions aux sous-problèmes sont nécessairement optimales elles aussi.

Pour être plus concret, supposons donc qu'on ait une série de six matrices $A_1, \ldots, A_6$ et que le parenthésage optimal soit le suivant:

$$(A_1((((A_2A_3)A_4)A_5)A_6))$$

Soit alors le sous-problème de multiplier les matrices A_2, A_3 et A_4. La sous-solution utilisée dans la solution optimale est $((A_2A_3)A_4)$. Supposons que cette solution ne soit pas optimale pour A_2, A_3 et A_4. (C'est ici la clé de la preuve par contradiction : on veut montrer que cette solution doit être optimale, en supposant la solution globale optimale. On va donc supposer qu'elle ne l'est pas, puis montrer que cela est impossible parce que cela conduit à une contradiction.) Si la solution $((A_2A_3)A_4)$ n'était pas optimale, cela signifierait alors qu'il existe *un autre* parenthésage qui soit meilleur, donc qui demande moins d'opérations, par ex., $(A_2(A_3A_4))$. Or, si on utilisait cet autre parenthésage dans le parenthésage initial (supposé optimal), la solution globale résultante utilisant cet autre parenthésage demanderait, *elle aussi*, moins d'opérations (de par la façon dont on compte le nombre total d'opérations, les tailles des autres sous-matrices n'ayant pas changé). En d'autres mots, on obtiendrait une meilleure solution que la solution initiale ... solution que nous avions supposé optimale. *Contradiction!*

Conclusion : il n'est pas possible qu'il existe un meilleur parenthésage pour $(A_2 \ldots A_4)$ si ce parenthésage fait partie du parenthésage optimal global.

Parenthèse 2: Détails additionnels justifiant le principe d'optimalité pour la multiplication d'une série de matrices

Preuve par contradiction

De façon plus explicite, on a utilisé le raisonnement qui suit dans la preuve plus haut (preuve par contradiction).

- Soit P et Q les propositions suivantes :
  - P = la solution globale est optimale
  - Q = la solution au sous-problème est optimale
- On veut montrer : $P \Rightarrow Q$
- Pour ce faire (preuve par contradiction), on suppose $\neg Q$, ce qui va nous permettre de conclure $\neg P$:
$$P \wedge \neg Q \Rightarrow \neg P$$
- Or, on a nécessairement :
$$P \wedge \neg Q \Rightarrow P$$
- Donc, des deux implications précédentes, on peut conclure :
$$P \wedge \neg Q \Rightarrow P \wedge \neg P$$
C'est-à-dire :
$$P \wedge \neg Q \Rightarrow FAUX$$

Contradiction! Il n'est donc pas possible que $\neg Q$ soit vrai.

Parenthèse 3: Petit rappel sur les preuves par contradiction

que seuls les éléments *au-dessus* de la diagonale ($i < j$) ont besoin d'être calculés. On note aussi que pour calculer $M[i][j]$, on doit avoir calculé $M[i][k]$ et $M[k+1][j]$ pour $k \geq i$ et $k \leq j-1$.

– L'algorithme 6 est alors celui résultant de l'application de cette stratégie. À cause de la façon dont sont utilisées les solutions des sous-problèmes, la table peut être remplie "diagonale par diagonale" ($M[i][i]$), à partir de la plus grande, tel que décrit précédemment. Par exemple, pour une matrice à six lignes et colonnes, on aura un remplissage comme suit, la `i`ème itération de la boucle externe remplissant les éléments indiqués par `i` :

0	1	2	3	4	5
	0	1	2	3	4
		0	1	2	3
			0	1	2
				0	1
					0

On verra, lorsqu'on abordera les algorithmes parallèles, que cela conduit à un calcul de type *wavefront*, les éléments de chaque diagonale pouvant se calculer de façon indépendante (donc en parallèle).

– L'analyse de l'algorithme est relativement direct : on a $\Theta(n^2)$ itérations des deux boucles externes (`dist` et `i`), correspondant au nombre d'éléments de la matrice. Chacune de ces itérations demande un travail d'au plus $\Theta(n)$ (recherche du minimum parmi au plus n éléments à l'aide de la boucle interne avec variable `k`). On conclut donc, comme dans la solution précédente, que l'algorithme est $\Theta(n^3)$.

```
PROCEDURE MinMult( d: sequence{Nat} ): Nat
ARGUMENTS
  Les matrices à multiplier sont A_1, ..., A_n.
  La taille de la matrice A_i est d_{i-1} × d_i.
PRECONDITION
  d = [d_0, d_1, ..., d_{n-1}, d_n]
DEBUT
   n <- length(d)-1
   M <- new Nat[n][n]
   POUR i <- 1 A n FAIRE
     // On définit les cas de base de la table.
     M[i][i] <- 0
   FIN
   POUR dist <- 1 A n-1 FAIRE
     POUR i <- 1 A n-dist
       j <- i + dist
       minMulij <-  +∞
       POUR k <- i A j-1 FAIRE
         nbMulikj <- M[i][k] + M[k+1][j] + (d[i-1] * d[k] * d[j])
         minMulij <- MIN{ minMulij, nbMulikj }
       FIN
       M[i][j] <- minMulij
     FIN
   FIN
   RETOURNER M[1][n]
FIN
```

Algorithme 6: Algorithme de programmation dynamique sans récursivité pour minimiser le nombre de multiplications d'une série de matrices

Programme MPD pour l'algorithme de programmation dynamique classique (calcul ascendant)

L'algorithme 7 présente une solution compilable et exécutable d'un programme, écrit dans la notation MPD, pour déterminer le nombre minimum de multiplications requises pour une chaîne de matrices (ici, pour simplifier, on utilise des tailles générées aléatoirement).

3.5 Arbres binaires de recherche optimaux

Section omise.

3.6 Problème du commis voyageur

À venir.

Note : Les sections suivantes ne sont pas présentes dans le manuel.

3.7 Le problème de sac à dos

– Le problème du sac à dos consiste à maximiser le "bénéfice" pouvant être obtenu en remplissant un sac avec divers items. À chaque item est associé un *bénéfice* (un entier positif) et un *poids* (un entier positif). L'objectif est de remplir le sac à dos de façon à maximiser le bénéfice, mais sans dépasser le poids maximum pouvant être contenu dans le sac.

– Dans la version *fractionnaire* de ce problème, il est permis de prendre une partie d'un item (par ex., on a trois kilos de sucre et on met seulement un kilo dans le sac). Dans ce cas, comme on le verra au prochain chapitre, il est possible d'utiliser un algorithme *vorace* pour obtenir une solution optimale (et ce en temps linéaire si on ignore le tri préalable des éléments, donc en temps $\Theta(n \lg n)$ si on compte le temps du tri).

– Dans le cas du problème du sac à dos 0-1 ("*0-1 knapsack problem*"), le problème est différent dans la mesure où il n'est pas possible de prendre une fraction d'un item. Un item donné peut soit être inclus dans le sac (1), soit ne pas être inclus (0). On verra au chapitre suivant qu'une solution vorace à ce problème ne conduit pas nécessairement à une solution optimale.

Dans ce qui suit, nous allons résoudre ce problème, tout d'abord à l'aide de la méthode diviser-pour-régner récursive (pour mieux comprendre le problème et l'idée générale de la solution), puis à l'aide d'une approche de programmation dynamique.

3.7.1 Solution diviser-pour-régner récursive

De facon récursive descendante, diviser-pour-régner (donc pas du tout efficace), on peut résoudre ce problème tel qu'indiqué à l'algorithme 8. Notons que, pour simplifier la présentation et aller à l'essentiel de l'algorithme, celui-ci retourne simplement *le bénéfice total résultant* ; il ne retourne donc pas la série d'items à inclure pour obtenir ce résultat optimal. Les explications sur la stratégie utilisée sont présentées dans les commentaires.

Très souvent, il est utile et intéressant de produire une telle solution récursive simple pour bien comprendre le problème et obtenir une solution qui pourra ensuite nous guider vers une solution plus efficace (une fois le problème et les caractéristiques de la solution ayant été mieux compris).

```
resource ParenthesageMatrices()
  # On lit le nombre de matrices a generer/traiter
  int n; getarg(1, n);
  if (n < 0) { write( "*** Erreur: n < 0" ); stop(1); }

  # Pour simplifier, on genere de facon aleatoire les tailles des n matrices.
  int d[0:n];
  for [i= 0 to n] {
    d[i] = int(random(1, n));
  }

  # La procedure principale.
  procedure MinMult( int d[0:*], int n ) returns int minMult
  {
    # On initialise la matrice des couts minimums sur la diagonale.
    int m[n][n];
    for [i = 1 to n] {
      m[i][i] = 0;
    }

    # On remplit la matrice.
    for [dist = 1 to n-1] {
      for [i = 1 to n-dist] {
        int j = i + dist;
        int minMulij = high(int);
        for [k = i to j-1] {
          int nbMulikj = m[i][k] + m[k+1][j] + (d[i-1]*d[k]*d[j]);
          minMulij = min( minMulij, nbMulikj );
        }
        m[i][j] = minMulij;
      }
    }
    minMult = m[1][n];
  }

  # On appelle la procedure.
  int minMult = MinMult(d, n);

  # On ecrit les entrees et le resultat (nombre total de multiplications).
  for [i = 1 to n-1] {
    writes( d[i-1], "x", d[i], " X " );
  }
  writes( d[n-1], "x", d[n], " => \n" );
  write( "", minMult, "multiplications" );
end
```

Algorithme 7: Programme MPD pour le calcul du nombre minimum de multiplications pour une série de matrices

```
PROCEDURE BeneficeSac( W: Nat, poids: sequence{Nat},
                                benefs: sequence{Nat} ): Nat
PRECONDITION
  length(poids) = length(benefs) = n
DEBUT
   // Pour simplifier, on retourne simplement le bénéfice résultant, pas la sélection d'items.
   RETOURNER BS'( n, W, poids, benefs )
FIN

PROCEDURE BS'(
  k: nat,                  // Prochain item à tenter d'inclure.
  w: nat,                  // Poids restant à combler.
  poids: sequence{Nat},    // On suppose que les séquences sont indexées à partir de 1.
  benefs: sequence{Nat}
  ): Nat
DEBUT
   SI w == 0 || k == 0 ALORS
     RETOURNER 0
   FIN

   SI poids[k] > w ALORS
     // Pour trouver la solution optimale pour un poids de w avec les
     // items 1 à k, on ne doit pas inclure l'item k pour un poids
     // restant w car son poids dépasse w.
     RETOURNER BS'( k-1, w, poids, benefs )
   SINON
     ASSERT( poids[k] <= w )
     // On peut inclure l'item k ... si le bénéfice en vaut la peine.
     // On examine deux solutions possibles (inclure ou non l'item k) et on choisit celle qui
     // apporte le meilleur bénéfice.

     // Première solution possible : on n'inclut pas l'item k: w restant ne change donc pas.
     b1 <- BS'( k-1, w, poids, benefs )

     // Deuxième solution possible : on inclut l'item k,
     // donc le poids restant est diminué en conséquence.
     b2 <- BS'( k-1, w - poids[k], poids, benefs ) + benefs[k]

     // On retourne la meilleure solution parmi les deux alternatives possibles.
     RETOURNER MAX{ b1, b2 }
   FIN
FIN
```

Algorithme 8: Algorithme diviser-pour-régner récursif pour le problème du sac à dos 0–1

3.7.2 Solution avec programmation dynamique

– Soit W le poids maximum pouvant être contenu par le sac. Soit n le nombre total d'items (on suppose que les items, de même que les poids et bénéfices associés, sont numérotés de 1 à n). La stratégie pour obtenir la solution maximale est de résoudre les divers sous-problèmes suivants :

- $B[k, w]$: trouver la façon de remplir le sac en utilisant uniquement les items 1, 2, ..., k ($k \leq n$) sans jamais dépasser un poids de w ($w \leq W$)

Si on peut résoudre tous ces sous-problèmes, alors la solution optimale au problème initial sera $B[n, W]$. Notons que ceci signifie qu'il faut, pour un poids W donné, calculer possiblement tous les bénéfices possibles pour $w = 0, 1, 2, 3, \ldots, W$.

– La formule récursive décrivant la solution optimale (uniquement le bénéfice) est la suivante :

$$
\begin{aligned}
B[k,w] &= \begin{cases} \max\{B[k-1,w],\ \text{benefs}[k] + B[k-1, w-\text{poids}[k]]\} & \text{si poids}[k] \leq w \\ B[k-1,w] & \text{si poids}[k] > w \end{cases} \\
B[i,0] &= 0, \text{ pour } i = 1, \ldots, n \\
B[0,w] &= 0, \text{ pour } w = 0, \ldots, W
\end{aligned}
$$

La deuxième clause de la formule pour $B[k, w]$ représente le cas où le poids de l'item k dépasse le poids restant, donc on ne doit pas inclure l'item. La première clause correspond au cas où le poids de l'item k ne dépasse pas le poids restant, auquel cas on doit déterminer si l'item doit être inclus ou non, en fonction du bénéfice associé.

– Un algorithme réalisant ces équations est présenté à l'algorithme 9 — notons que le calcul de $B[k, w]$ nécessite au plus l'utilisation des éléments $B[k-1, 0]$ à $B[k-1, w]$, donc les éléments de la ligne précédente à droite ou immédiatement au-dessus.

– Complexité de l'algorithme : $T(n) \in \Theta(nW)$:

- Le temps est clairement dominé par le travail fait dans les deux boucles `POUR` imbriquées (les deux premières boucles ne servent que pour initialiser la première ligne et première colonne et sont respectivement $\Theta(n)$ et $\Theta(W)$).
- Le nombre total d'itérations est $n \times (W+1) \in \Theta(nW)$. Le travail effectué dans chacune des itérations est $\Theta(1)$.

Remarque : On dit d'un tel algorithme qu'il est en temps *pseudo-polynomial* parce que son temps d'exécution dépend de la *magnitude* du nombre reçu en argument, et non de la taille de l'encodage de cet argument. Bien qu'un tel algorithme puisse être plus efficace qu'un algorithme utilisant une approche brute et naïve, le temps d'exécution peut devenir très important pour des grandes valeurs de W.

3.8 Programmation dynamique et programmation fonctionnelle

Dans ce qui suit, nous allons montrer que, dans un langage fonctionnel *non strict*, la correspondance est quasi directe entre les équations définissant une solution de programmation dynamique et le programme lui-même. Mais tout d'abord, il faut comprendre ce qu'est un langage fonctionnel *non strict*.

```
PROCEDURE BeneficeSac( W: nat, poids: sequence{Nat}, benefs:
sequence{Nat} ): Nat
PRECONDITION
  length(poids) = length(benefs) = n
DEBUT
  // On utilise une 0ième ligne pour faciliter le calcul de la vraie 1ère ligne.
  B <- new Nat[n+1][w+1]
  POUR i <- 1 A n FAIRE
    B[i, 0] <- 0
  FIN
  POUR w <- 0 A W FAIRE
    B[0, w] <- 0
  FIN

  POUR k <- 1 A n FAIRE
    POUR w <- 0 A W FAIRE
      SI poids[k] > w ALORS
        B[k, w] <- B[k-1, w]
      SINON
        ASSERT(poids[k] <= w)
        B[k, w] <- MAX{ B[k-1, w], benefs[k]+B[k-1, w-poids[k]] }
      FIN
    FIN
  FIN
  RETOURNER B[n, W]
FIN
```

Algorithme 9: Algorithme de programmation dynamique pour le problème du sac à dos 0–1

3.8.1 Langage fonctionnel

Comme on l'a vu au chapitre 2, un langage de programmation *purement* fonctionnel — on dit aussi langage applicatif — est un langage basé uniquement sur l'utilisation et la manipulation de valeurs et de fonctions, donc basé strictement *sur l'évaluation d'expressions.*

La forme générale d'un programme fonctionnel est donc celle d'une série de déclarations (constantes et fonctions), plus une expression à évaluer :

```
ident_1 = ...
ident_2 = ...
...
ident_k = ...
expression à évaluer
```

3.8.2 Langage fonctionnel strict vs. langage non strict

On distingue deux grandes classes de langages fonctionnels, selon la façon dont les expressions utilisées comme arguments aux fonctions sont évaluées :

- Langage strict ⇒ les arguments sont toujours évalués, même s'ils ne sont pas utilisés.
- Langage non strict ⇒ les arguments peuvent ou non être évalués lorsque non utilisés, selon qu'une approche indulgente ou paresseuse est utilisée.

– **Langage strict** = si la valeur d'une expression utilisée comme argument effectif n'est pas définie (par ex., son évaluation génère une erreur d'exécution ou une boucle infinie), alors le résultat de la fonction n'est pas défini.

– Stratégie de mise en oeuvre = appel-par-valeur ⇒ on évalue les arguments *avant* l'appel, ensuite on appelle la fonction.

Exemple :

```
f x = x + f (x+1)
f' x = 1 / x
g x = 0

g (f 0)  -- ... => ne termine jamais!
g (f' 0) -- ... => erreur (division par 0)
```

Dans l'appel `g (f 0)`, une série infinie d'appels récursifs sera générée (pas de condition de terminaison à la récursion). Dans l'appel `g (f' 0)`, une erreur résultant de la division de 1 par 0 sera générée.

– **Langage non-strict** = il est possible pour une fonction de recevoir un argument qui n'est pas défini et de quand même produire un résultat.

– Deux stratégies possibles de mise en oeuvre :

1. Approche paresseuse : on appele immédiatement la fonction *sans évaluer* les arguments. On évalue un argument uniquement lorsqu'il devient requis dans la fonction (appel-par-nécessité).

   ⇒ Une expression utilisée comme argument doit être encapsulée dans une *suspension* (dans un *glaçon*).[4] Lorsqu'un argument est utilisé, on évalue la suspension (on fait *fondre* le glaçon).

[4] Notons que c'est le compilateur qui se charge de ce travail. Tout cela est donc transparent pour l'utilisateur du langage.

2. Approche indulgente : aucun ordre d'évaluation *a priori* n'est fixé, c'est-à-dire qu'il suffit de respecter les dépendances de données (si une expression utilise `x`, alors on doit attendre que `x` soit disponible).

   $\Rightarrow$ Les arguments et la fonction peuvent être évalués *en parallèle* et l'appelant et l'appelé se synchronisent pour assurer le respect des dépendances de données ($\approx$ producteur/consommateur).

3.8.3 Langage fonctionnel avec évaluation paresseuse

– Dans un langage fonctionnel *paresseux* (nonobstant les *optimisations* possibles) un argument n'est évalué que si il est *vraiment requis* = "appel par nécessité" (*call-by-need*). En d'autres mots, une expression utilisée comme argument est évaluée 1 ou 0 fois, selon que l'argument formel correspondant est utilisé ou non, jamais plus.

– Un exemple simple : l'expression `f' 0` n'est pas évaluée dans l'évaluation de l'expression "`g (f' 0) == 1`", car elle n'est pas nécessaire pour produire le résultat de l'appel à `g` (qui retourne toujours 0, donc l'expression est fausse) $\Rightarrow$ *pas* d'erreur de division par 0.

```
f' x  = 1 / x
g x  = 0

g (f' 0) == 1
```

– Un autre exemple simple est présenté dans l'extrait de code Haskell 1. Ce programme *ne génère pas* de récursivité infinie, parce que l'appel à `f` à droite de ":" ne s'effectue que si on utilise la partie `tail` (le constructeur ":" est lui aussi paresseux).

Code Haskell 1 Exemple illustrant la non évaluation d'un argument

```
-- Note : (x : xs) dénote une séquence dont le premier élément est x
--        suivi d'une séquence d'éléments dénotée par xs.
-- Note :  "_" dénote une valeur arbitraire ( don't care).

head (x : _)  = x     -- Retourne l'élément en tête d'une séquence.
tail (_ : xs) = xs    -- Retourne la séquence à l'exception du premier élément.

f x = x : f (x + 1)

head (tail (f 0)) == 1
```

Plus précisément, pour l'exécution de ce programme, on effectuerait les *réductions* suivantes :

```
head (tail (f 0)) == 1
--> head (tail (0 : f (0+1))) == 1
--> head (f (0+1)) == 1
--> head ((0+1) : (f (0+(0+1)))) == 1
--> (0+1) == 1
--> 1 == 1
--> true
```

– L'évaluation paresseuse est utile :

- Pour la manipulation de structures de données *potentiellement* infinies.
- Pour bien modulariser certains programmes en créant un couplage faible entre le producteur d'une structure de données et son consommateur.

– Exemple : Programme pour la recherche de nombres premiers à l'aide du crible d'Eratosthène, présenté dans l'extrait de code Haskell 2.

Code Haskell 2 Calcul de nombres premiers par la méthode du crible d'Ératosthène

```
-- Pour produire les 100 premiers nombres premiers.
take 100 premiers

-- Pour obtenir les nombres premiers strictement inferieurs a 100.
takewhile (< 100) premiers

-- Pour obtenir tous les nombres premiers compris entre 1000 et 9999.
takewhile (<= 9999) (dropwhile (< 1000) premiers)


-- Fonction principale.
premiers :: [Int]
premiers = cribleEratosthene [2..]
           where  cribleEratosthene (x : xs)
                      = x : (cribleEratosthene (filtrerMultiples x xs))

-- Supprimer de tous les multiples de p de la liste l.
filtrerMultiples :: Int -> [Int] -> [Int]
filtrerMultiples p l = [x | x <- l, not(x `mod` p == 0)]


-- Fonctions auxiliaires
takewhile :: (t -> Bool) -> [t] -> [t]
takewhile p [] = []
takewhile p (x : xs)
  | p x       = x : takewhile p xs
  | otherwise = []

dropwhile :: (t -> Bool) -> [t] -> [t]
dropwhile p [] = []
dropwhile p (x : xs)
  | p x       = dropwhile p xs
  | otherwise = x : xs
```

– Propriétés intéressantes :

- On peut travailler avec la liste (potentiellement) infinie de tous les nombres premiers.
- Permet une modularisation élégante du problème : on peut *découpler* le processus de génération des nombres premiers du processus de sélection ⇒ facile de changer le processus de sélection.

Le dernier point est particulièrement intéressant. Dans un langage procédural traditionnel, les divers programmes qui permettraient d'obtenir les différentes collections de nombres

premiers produites dans le code Haskell 2 — ceux plus petits que 100, les 100 premiers, ceux compris entre deux bornes — seraient très différents les uns des autres.[5]

– Autre exemple = calcul de la racine carrée d'un nombre par la méthode de Newton-Raphson (extrait de code Haskell 3).

La méthode de Newton-Raphson pour le calcul d'une racine carrée est une méthode *itérative* de calcul par *approximations successives* :

> Soit n un nombre pour lequel on veut calculer $\sqrt{n}$
>
> Soit a_0, une approximation initiale (par ex., $(n+1)/2$).
>
> Soit $a_{i+1} = (a_i + N/a_i)/2$.
>
> Alors, la suite suivante *converge* vers $\sqrt{n}$:

$$a_0, a_1, a_2, \ldots, a_i, a_{i+1}, \ldots$$

Code Haskell 3 Calcul de la racine carrée par la méthode de Newton-Raphson en Haskell

```
-- Pour generer les approximations.
prochaineApprox n ai = (ai + (n / ai)) / 2

genererApproximations n = approximations
  where
    approximations = a0 : map (prochaineApprox n) approximations
    a0 = (n+1) / 2

-- Obtenir la racine carree pour le cas ou l'approximation est acceptee
-- lorsque l'erreur *absolue* est plus petite ou egale a eps.
racineCarree n eps =
  erreurInf eps (genererApproximations n)

erreurInf eps (a0 : a1 : as)
| abs (a0 - a1) <= eps = a1
| otherwise            = erreurInf eps (a1 : as)



-- Obtenir la racine carree pour le cas ou l'approximation est acceptee
-- lorsque l'erreur *relative* est plus petite ou egale a eps.
racineCarree n eps =
  erreurRelativeInf eps (genererApproximations n)

erreurRelativeInf eps (a0 : a1 : as)
| abs(a0 - a1) / (abs a1) <= eps = a1
| otherwise                      = erreurRelativeInf eps (a1 : as)
```

Dans un langage procédural traditionnel, la modification des conditions déterminant quand une approximation est acceptable ou non (erreur relative? erreur absolue?) entrainerait des modifications importantes au programme.

[5] À moins de construire le programme, ce qu'il est possible de faire, pour émuler la production d'un flot (un *stream*) de valeurs comme le fait naturellement l'évaluation paresseuse.

3.8.4 Liens avec la programmation dynamique

Les exemples qui suivent, encore présentés en Haskell, illustrent comment certains problèmes de programmation dynamique s'expriment simplement dans un langage fonctionel non strict. Comme on l'a vu précédemment, Haskell est un langage paresseux, une forme de langage non strict. Toutefois, ces exemples pour être valables ne requièrent qu'une forme d'évaluation non stricte, donc pas nécessairement paresseuse. Ces exemples pourraient être exécutés aussi bien en Haskell qu'en `pH`, une version indulgente, mais non paresseuse, de Haskell.

A. Calcul des nombres de Fibonacci

La fonction `fib` suivante permet de calculer le `n`ième nombre de Fibonacci :

```
-- Note: En Haskell, l'indexation d'un tableau a[i] est notee a!i.
fib n =
  let
    a = array (0, n) (
        -- fib(0) = 1
           (0, 1) :
        -- fib(1) = 1
           (1, 1) :
        -- fib(i) = fib(i-1) + fib(i-2),  pour i = 1 a n
           [(i, a!(i-1)+a!(i-2)) |  i <- [2..n]]
        )
  in
    a!n
```

La forme générale d'une déclaration et définition d'un tableau (à une dimension) en Haskell est la suivante :

```
nomTableau = array (borneInférieure , borneSupérieure)
             liste de paires définissnt le contenu du tableau
```

Chaque paire (2-tuple) de la liste de paires indique l'index de l'élément, suivi de la valeur de cet élément. Dans l'exemple qui précède, les deux premiers éléments de cette liste de paires sont (0, 1) et (1, 1), qui indiquent donc que `a[0] = 1` alors que `a[1] = 1`. Quant à la liste qui suit, définie en compréhension, elle indique que `a[i] = a[i-1]+a[i-2]`.

Notons que dans cet exemple, l'évaluation paresseuse assure que chacun des éléments du tableau ne sera calculé qu'au moment où il sera nécessaire pour produire le résultat. De plus, l'évaluation paresseuse assure aussi que chaque expression définissant un élément du tableau ne sera évaluée qu'une seule et unique fois (évaluation paresseuse $\Rightarrow$ on gèle l'expression puis, lorsqu'on doit l'évaluer, on remplace le *glaçon* par la valeur résultante).

B. Coefficient binomial

Le coefficient binomial est défini comme suit, pour $0 \leq k \leq n$:

$$\left(\begin{array}{c} n \\ k \end{array} \right) \quad = \quad \frac{n!}{k!(n-k)!}$$

Il est possible de montrer que le coefficient peut aussi s'exprimer de la façon suivante, ce qui évite de calculer $n!$ qui devient très grand rapidement :

$$\begin{array}{rcll} \left(\begin{array}{c} n \\ k \end{array} \right) & = & \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right) + \left(\begin{array}{c} n-1 \\ k \end{array} \right) & \text{pour } 0 < k < n \\ & = & 1 & \text{pour } k = 0 \text{ ou } k = n \end{array}$$

– Une mise en oeuvre directe de cette équation en Haskell a l'allure suivante :

```
bin n 0             = 1
bin n k | n == k    = 1
        | otherwise = bin (n-1) (k-1) + bin (n-1) k
```

Même pour de petites valeurs (par ex., $n = 100, k = 10$), le temps d'exécution est très élevé (aucune réponse sur `arabica`, la machine Sun du département, après plusieurs minutes d'exécution).

– Une solution de programmation dynamique, par contre, est plus efficace et presqu'aussi simple (et s'exécute de façon quasi immédiate sur `arabica`) :

```
bin' n k =
 let
  b = array ((0, 0), (n, k)) (
      [((i, 0), 1) | i <- [0..n]]
      ++
      [((i, i), 1) | i <- [1..k]]
      ++
      [((i, j), b!(i-1, j-1) + b!(i-1, j)) | i <- [1..n], j <- [1..k], not (i == j)]
  )
 in
   b!(n, k)
```

C. Multiplication d'une chaîne de matrices

La solution en Haskell s'exprime simplement comme suit :

```
-- Note: En Haskell, l'indexation d'une liste (sequence) est notee a!!i.
MinMult d =
  let
   n = length d - 1
   m = array ((1, 1), (n, n)) (
       [((i, i), 0) | i <- [1..n]]
       ++
       [((i, j), mulij i j) | i <- [1..n], j <- [1..n], i < j]
       )
       where
         mulij i j = minimum [m!(i,k) + m!(k+1,j) + d!!(i-1) * d!!k * d!!j | k <- [i..j-1]]
  in
   m!(1, n)
```

D. Problème du sac à dos 0–1

La solution en Haskell s'exprime simplement comme suit :

```
benefsSac poidsMax poids benefs =
  let
    n = length poids
    b = array ((0, 0), (n, poidsMax)) (
         [((i, 0), 0) | i <- [1..n]]
         ++
         [((0, w), 0) | w <- [0..poidsMax]]
         ++
         [((k, w), benef k w) | k <- [1..n], w <- [1..poidsMax]]
        )
        where
          benef k w = if poids!!(k-1) <= w then
                        maximum [b!(k-1, w), benefs!!(k-1) + b!(k-1, w-poids!!(k-1))]
                      else
                        b!(k-1, w)
  in
    b!(n, poidsMax)
```

Une caractéristique importante de cette solution, parce qu'elle est écrite dans un langage paresseux, est que *seuls les éléments* du tableau *requis pour produire le résultat final* `b!(n, poidsMax)` seront calculés. En d'autres mots, la complexité asymptotique résultante ne sera pas nécessairement $\Theta(nW)$ mais pourrait plutôt n'être que $\Theta(n^2)$.