Résumé du chapitre 3 : Programmation dynamique

23–30 sept. 2002

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.

– L'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 *un peu* différente" de la programmation dynamique n'est pas souvent présentée dans les références classiques sur la conception et l'analyse d'algorithmes ... parce que ces références reposent essentiellement sur le paradigme impératif et procédural. Par contre, dans le cadre du paradigme de *programmation fonctionnelle*, plus précisément dans les langages fonctionnels stricts, cette façon d'approcher la programmation dynamique est fondamentale, sinon nécessaire.

3.0 Nombre de Fibonacci

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

3.0.1 Version 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
```

Complexité : $O(2^n)$

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

```
PROCEDURE fib( n: Nat ): Nat
DEBUT
   dict <- new Dictionnaire()
   dict.insererCleDefn( 0, 1 )
   dict.insererCleDefn( 1, 1 )
   RETOURNER fib'( n, dict )
FIN

PROCEDURE fib'( n: Nat, dict: Dictionnaire ): Nat
DEBUT
   r <- dict.obtenirDefn( n )
   SI r != CLE_ABSENTE ALORS
      // L'appel fib(n) a déjà été calculé
      RETOURNER r
   FIN

   // Premier appel à fib(n)
   r1 <- fib'( n-1, dict ) // dict peut être modifié
   r2 <- fib'( n-2, dict ) // idem
   r <- r1 + r2
   dict.insererCleDefn( n, r )
   RETOURNER r
FIN
```

Supposons que le temps pour insérer une clé et sa définition de même que le temps pour retrouver la définition associée à une clé soient $O(1)$ — par exemple, en utilisant une table de hachage, ou encore un tableau, si on sait *a priori* que les clés seront nécessairement dans un intervalle `0` à `n`.

Complexité : ?

3.0.3 Méthode de programmation dynamique plus classique

Dans la méthode classique de programmation dynamique, on construit une table qui contiendra 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          // Idem.
   POUR i <- 2 A n FAIRE
      // Construction (ascendante) de la table
      a[i] <- a[i-1] + a[i-2]
   FIN
   RETOURNER a[n]
FIN
```

Complexité : ?

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

Un algorithme basé sur la programmation dynamique peut être utilisé pour résoudre un problème d'optimisation (c'est-à-dire, identifier *une* solution optimale à un problème) en utilisant une stratégie basée sur les étapes suivantes :

1. Établir (définir) une propriété récursive qui donne la solution optimale à une instance du problème.

2. Calculer la valeur de la solution optimale de façon ascendante.

3. Utiliser la table construite par la programmation dynamique pour construire, de façon ascendante, une solution.

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 :** 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.

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.

Applicabilité de la programmation dynamique : Cormen, Leiserson et Rivest expriment le principe d'optimalité en disant plutôt 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 alors un bon candidat pour une solution basée sur la programmation dynamique si *le problème fait apparaître une sous-structure optimale* et si le problème contient des **sous-problèmes superposés** (c'est-à-dire que l'algorithme récursif génère souvent les mêmes sous-problèmes).

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 en jouant avec l'*associativité*

– 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 :

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

– 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.

3.4.1 Version diviser-pour-régner (pure)

– L'algorithme présenté à la Figure 1 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 $A_1, \ldots, A_n$, 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).

– 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 2. 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 appels `MinMult'` sous une telle expression sont ceux (les deux appels) faits pour cette valeur de `k` donnée.

En fait, il est possible de prouver que la complexité de l'algorithme résultant est exponentiel, c'est-à-dire $O(2^n)$.

3.4.2 Version récursive avec mémorisation

– L'algorithme présenté à la Figure 3 est une version récursive descendante avec mémorisation pour la solution au problème du minimisation du nombre d'opérations pour la multiplication d'une chaîne de matrices.

– À première vue, la complexité de l'algorithme peut être décrite par l'équation de récurrence suivante, en supposant, comme précédemment, que chacune des opérations de manipulation du dictionnaire est $\Theta(1)$:

- $T(i,j) = \sum_{k=1}^{j-1} [T(i,k) + T(k+1,j)] + \Theta(1)$
- $T(i,i) = \Theta(1)$

– Question : La solution de cette équation conduirait-elle à un résultat optimiste ou pessimiste par rapport à la véritable complexité de l'algorithme?

Réponse : ?

```
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
      RETOURNER 0
   FIN
   minMulij <- +∞ // On va chercher le nombre minimum requis d'ops
   POUR k <- i A j-1 FAIRE
      // Diviser: on décompose en 2 sous-problèmes qu'on résout récursivement
      nbMulik <- MinMult'( d, i, k )   // Sous-problème récursif
      nbMulkj <- MinMult'( d, k+1, j ) // Autre sous-problème récursif

      // Combiner: on connait 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 le minimum
      SI nbMulikj < minMulij ALORS
         minMulij <- nbMulikj
      FIN
   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
```

Figure 1: Algorithme récursif pour minimiser le nombre de multiplications de matrices

```
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 2: Exemple d'exécution pour `MinMult'` récursif

```
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
   dict <- new Dictionnaire()
   POUR i <- 1 A n FAIRE
      // On établit les cas de base de la récursion
      dict.insererCleDefn( (i, i), 0 )
   FIN
   RETOURNER MinMult'( d, 1, n, dict )
FIN

PROCEDURE MinMult'(
   d: sequence{Nat},
   i, j: Nat,
   dict: Dictionnaire
   ): Nat
DEBUT
   r <- dict.obtenirDefn( (i, j) )
   SI r != CLE_ABSENTE ALORS
      RETOURNER r
   FIN
   // ASSERT: i < j
   minMulij <- +∞
   POUR k <- i A j-1 FAIRE
      nbMulik <- MinMult'( d, i, k, dict )
      nbMulkj <- MinMult'( d, k+1, j, dict )
      nbMulikj <- nbMulik + nbMulkj + (d[i-1] * d[k] * d[j])
      SI nbMulikj < minMulij ALORS
         minMulij <- nbMulikj
      FIN
   FIN
   dict.insererCleDefn( (i, j), minMulij )
   RETOURNER minMulij
FIN
```

Figure 3: Algorithme récursif avec *mémorisation* pour minimiser le nombre de multiplications de matrices

3.4.3 Version (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 :

- $M[i][j] = \overset{\text{minimum}}{i \leq k \leq j\text{-}1} (M[i][k] + M[k+1][j] + d_{i\text{-}1} d_k d_j)$, pour $i < j$
- $M[i][i] = 0$, pour $1 \leq i \leq n$

– 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., $A_1(((A_2(A_3A_4))A_5)A_6)$).

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 ... et 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.

Petit rappel sur les preuves 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.

– Le principe d'optimalité étant valable et ayant défini une propriété pour la solution optimale, il reste ensuite à définir un algorithme qui permettra de construire de façon ascendante la solution, tel que cela est illustré à la Figure 3.8 (p. 108), où l'on remarque que seuls les éléments au-dessus de la diagonale ont besoin d'être calculés.

– L'algorithme résultant est présenté à la Figure 4. À cause de la façon dont sont utilisées les solutions des sous-problème, la table peut être remplie diagonale par diagonale, à parir de la plus grande ($M[i][i]$). Par exemple, pour une matrice à six lignes et colonnes, on aurait 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 : $\boxed{?}$

```
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
   POUR i <- 1 A n FAIRE
      // On définit les cas de base de la table
      M[i][i] <- 0
   FIN
   POUR diag <- 1 A n-1 FAIRE
      POUR i = 1 A n-diag
         j = i + diag
         minMulij <- +∞
         POUR k <- i A j-1 FAIRE
            nbMulikj <- M[i][k] + M[k+1][j] + (d[i-1] * d[k] * d[j])
            SI nbMulikj < minMulij ALORS
               minMulij <- nbMulikj
            FIN
         FIN
         M[i][j] <- minMulij
      FIN
   FIN
   RETOURNER M[1][n]
FIN
```

Figure 4: Algorithme de programmation dynamique sans récursivité pour minimiser le nombre de multiplications de matrices

3.5 Arbres binaires de recherche optimaux

Section omise.

3.6 Problème du commis voyageur

À venir (?).

<u>Note</u> : Les sections suivantes ne sont pas présentes dans le manuel.

3.7 Le problème de sac à dos

– Le problème du sac à dos est de 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).

– 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 (grosso modo) tel qu'indiqué à la Figure 5 (ici, on retourne simplement le bénéfice total résultant). Les explications sur la stratégie de solution 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).

```
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 pas 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: le poids restant est donc diminué en conséquence.
      b2 <- BS'(k-1, w - poids[k], poids, benefs) + benefs[k]

      // On retourne la meilleure solution
      RETOURNER max{b1, b2}
   FIN
FIN
```

Figure 5: 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 initiale sera $B[n, W]$. Notons que ceci signifie qu'il faut, pour un poids W donné, calculer toutes les possibilités pour $w = 0, 1, 2, 3, \ldots, W$.

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

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

La deuxième clause de la formule $B[k, w]$ 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 pour ces équations aurait alors l'allure présentée à la Figure 6 — notons que le calcul de $B[k, w]$ nécessite au plus l'utilisation des éléments $B[k\text{-}1, 0]$ à $B[k\text{-}1, w]$, donc les éléments de la ligne précédente à droite ou immédiatement au-dessus.

– Complexité de l'algorithme : ?.
On dit d'un tel algorithme qu'il est en temps ?

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 fonctionnel 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
   POUR i <- 1 A n FAIRE
      B[i, 0] <- 0
   FIN
   // On ajoute une 0ième ligne, pour faciliter le calcul de la 1ère ligne.
   POUR w <- 0 A W FAIRE
      B[0, w] <- 0
   FIN

?

   RETOURNER B[n, w]
FIN
```

Figure 6: Algorithme de programmation dynamique pour le problème du sac à dos 0–1

3.8.1 Langage fonctionnel strict vs. langage non strict

– **Langage strict** = si un argument effectif n'est pas défini (par ex., erreur ou boucle), 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.

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

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

– **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*). Lorsqu'un argument est utilisé, on évalue la suspension (on fait *fondre* le glaçon).

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

   ⇒ Les arguments et la fonction peuvent être évalués *en parallèle* et l'appelant et l'appelé se synchronisent (≈ producteur/consommateur).

3.8.2 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.

– Un exemple simple : l'expression `f 0` n'est pas évaluée ⇒ *pas* d'erreur de division par 0

```
f x  = 1 / x
g y  = 1

g (f 0) == 1
```

– Autre exemple simple : le programme suivant *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) :

```
head (x : _)  = x
tail (_ : xs) = xs

f x = x : f (x + 1)

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

– 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 : Recherche de nombres premiers à l'aide du crible d'Eratosthène, présenté à la Figure 7.

– 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.

Dans un langage procédural traditionnel, les programmes permettant d'obtenir les différentes collections de nombres premiers (ceux plus petits que 100 vs. les 100 premiers vs. ceux compris entre deux bornes) seraient très différents les uns des autres . . . à moins de simuler la production d'un flot (*stream*) de valeurs comme le fait naturellement l'évaluation paresseuse.

– Autre exemple = calcul de la racine carrée d'un nombre par la méthode de Newton-Raphson :

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 $\sqrt{n}$:
>
> $$a_0, a_1, a_2, \ldots, a_i, a_{i+1}, \ldots$$

Solution `Haskell` : voir Figure 8.

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

```
-- 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
```

Figure 7: Calcul de nombres premiers par la méthode du crible d'Ératosthène en Haskell

```
-- Generation des 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 lorsque l'approximation est acceptee
-- si 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 lorsque l'approximation est acceptee
-- si 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)
```

Figure 8: Calcul de la racine carrée par la méthode de Newton-Raphson en Haskell

3.8.3 Liens avec la programmation dynamique

Les exemples qui suivent, qui illustrent comment certains problèmes de programmation dynamique s'expriment simplement dans un langage fonctionel non strict, seront aussi présentés en Haskell (langage *paresseux*). Toutefois, ces exemples ne nécessitent que l'évaluation non stricte, donc 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
```

Dans ce cas-ci, 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 calculé 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 $0 \leq k \leq n$:

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n\text{-}k)!}$$

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 très rapidement :

$$\begin{aligned} \begin{pmatrix} n \\ k \end{pmatrix} &= \begin{pmatrix} n\text{-}1 \\ k\text{-}1 \end{pmatrix} + \begin{pmatrix} n\text{-}1 \\ k \end{pmatrix} && \text{pour } 0 < k < n \\ &= 1 && \text{pour } k = 0 \text{ ou } k = n \end{aligned}$$

– 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 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 programmation dynamique 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 programmation dynamique 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 sera plutôt ? .