Algorithmique avancée

Enssat LSI2

Olivier Pivert

Table des matières

Chapitre 1

Introduction

1.1 Introduction

1.1.1 *Divers aspects de l'algorithmique*

Definition 1.1. Un algorithme est une méthode de résolution d'un problème ; en informatique, c'est un procédé utilisable pour trouver la solution à un problème, donc une séquence finie d'instructions (à signification précise) s'exécutant en temps fini dans une mémoire finie.

Domaine de l'algorithmique.

- la conception des algorithmes : elle n'est pas automatisée, ie il n'existe pas de méthode générale de production des algorithmes dans le cas général (cas particulier : générateur de compilateur). Il existe des techniques de conception qui aident à exhiber de "bons" algorithmes ; bien que non universelles, elles facilitent la résolution de nombreux problèmes (diviser pour règner, essais successifs, programmation dynamique par exemple)
- l'expression des algorithmes : nécessité d'un support pour exprimer les algorithmes
- la validation des algorithmes : une fois exhibé, il faut prouver que l'algorithme fournit toujours le résultat voulu
- l'analyse des algorithmes : étudier le temps et la mémoire nécessaires à l'exécution d'un algoritme (deux situations principales : en moyenne et au pire) ; ceci fait souvent appel à des calculs mathématiques et on s'intéresse à la notion d'ordre de grandeur et non à des valeurs précises pour la mémoire et le temps requis.
- la définition de jeux d'essais : dans le cadre de l'étude expérimentale d'un algorithme, on le programme et on l'exécute sur des données d'entrée "représentatives" qu'il faut définir : ce sont les jeux d'essais. Ils servent à la fois à mettre au point le programme et à mesurer le temps effectif requis pour l'exécution.

Dans ce cours, on s'intéresse surtout à la conception, l'analyse et la validation des algorithmes.

Bibliographie disponible à l'ENSSAT :

- Horowitz, Shani : Fundamentals of computer algorithms
- Aho, Hopcroft, Ullman : Structures de données et algor.
- Knuth : Fundamental algorithms
- Gondran, Minoux : Graphes et algorithmes
- Beaudoin, Meyer : Méthodes de programmation
- Mohr, Pair, Schott : Construire les algorithmes
- Wirth : Algorithmes et structures de données

1.1.2 *A propos de complexité temporelle*

Rappels :

- la notation O. On note N les entiers, N^+ les entiers positifs, R les réels, R^+ les réels strictement positifs, R^* les réels positifs ou nuls. Soit $f : N \to R^*$ une fonction quelconque. On définit la classe de fonctions $O(f(n))$ par les fonctions $t(n)$ bornées supérieurement par un multiple de $f(n)$ au-delà d'un seuil donné, soit :

$$O(f(n)) = \{t : N \to R^+,\ \exists c \in R^+,\ \exists n_0 \in N,\ \forall n \geq n_0,\ [t(n) \leq c.f(n)]\}$$

- la notation θ. Il s'agit de l'ordre exact de f(n) défini par :

$$\theta(f(n)) = \{t : N \to R^*, \exists c, d \in R^+, \exists n_0 \in N, \forall n \geq n0, [c.f(n) \leq t(n) \leq d.f(n)]\}.$$

Complexité au pire, au mieux, en moyenne.

Classes de complexité : échelle.

$$log_k\, n,\ n,\ nlog\, n,\ n^2,\ n^3,\ \ldots,\ 2^n,\ n!$$

Jusqu'à n^k (avec k petit) on aura une réponse (faisable)

Pour les autres classes : faisable avec n faible seulement.

1.1.3 *Exemples de raffinement d'algorithmes*

1.1.3.1 *Le calcul de C_n^p*

Méthode 1. On utilise la définition usuelle, soit :

$$C_n^p = \frac{n!}{p!\,(n-p)!}$$

d'où en calculant les trois factorielles on a un nombre de multiplications/divisions égal à :

$$(n-1) + (p-1) + (n-p-1) + 2 = 2n - 1$$

Méthode 2. Idée : en calculant $n!$, on a déjà calculé $p!$ et $(n-p)!$ — on suppose $(n-p) \geq p$. On fait le calcul en trois "tranches" : $p!$, $(n-p)!$, et $n!$. En réalité, on calcule $n!$ avec deux arrêts. On calcule donc les trois factorielles en même temps en

une boucle, d'où $(n-1)+2 = n+1$ opérations.

Méthode 3. On simplifie l'expression qui devient :

$$\frac{(n-p+1)\times\ldots\times(n)}{1\times 2\times\ldots\times p}.$$

Supposons que $p \leq n-p$ (ce qui ne gêne pas car sinon on calcule C_n^{n-p}). On a alors

$$(n-(n-p+1))+(p-1)+1 = 2p-1$$

opérations à effectuer. Ce nombre est inférieur à $n+1$. En effet :

$$\begin{aligned} p \leq n-p &\Rightarrow 2p \leq n \\ &\Rightarrow 2p-1 \leq n-1 \leq n+1. \end{aligned}$$

1.1.3.2 *Le calcul du n^e terme de la suite de Fibonacci*

Rappel :

$fib(1) = fib(2) = 1$
$fib(n) = fib(n-1) + fib(n-2)$ pour $n > 2$.

Méthode 1. On peut démontrer par récurrence que :

$$fib(n) = \frac{(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n}{\sqrt{5}}.$$

On peut par conséquent utiliser l'algorithme suivant (en langage de description) :

fonction *fibo1* (ent n) **résultat** réel ;
début
    résult $\leftarrow$ ((((1 + $sqrt(5)$)/2) ** n − ((1 − $rac(5)$)/2) ** n)/$sqrt(5)$)
fin.

Il donnera une valeur approchée (un réel) de la valeur exacte (qui est un entier) et ce de façon très rapide (instantanément). Neanmoins, la précision de calcul de la racine carrée génère des erreurs d'arrondis pour des valeurs assez grandes de n.

Méthode 2. Calcul naïf par une fonction récursive (calquée sur la formule de récurrence).

fonction *fibo2* (ent n) **résultat** ent ;
début
    si $n = 1$ **ou** $n = 2$ **alors résult** $\leftarrow 1$
    sinon résult $\leftarrow fibo2(n-1) + fibo2(n-2)$
    fsi
fin.

Tracer l'arbre des appels pour $fibo2(6)$. On constate une importante redondance de calcul.

Cette fonction conduit au bon résultat mais son exécution est longue en raison de cette redondance.

Exercice. Calculer le nombre d'appels récursifs engendrés par l'appel $fibo2(k)$.

Solution.

$$nbap(k) = \begin{cases} 0 \text{ si } k = 1 \text{ ou } k = 2 \\ 2 + nbap(k-1) + nbap(k-2) \text{ si } k > 2 \end{cases}$$

ce qui se réécrit en posant $NB(i) = nbap(i) + 2$:

$$NB(k) = \begin{cases} 2 \text{ si } k = 1 \text{ ou } k = 2 \\ NB(k-1) + NB(k-2) \text{ si } k > 2 \end{cases}.$$

On retombe sur une suite de Fibonacci avec valeurs initiales différentes (2 au lieu de 1). Il est aisé de montrer que la valeur du terme général de cette suite est le double de celle de la suite d'origine, d'où :

$$NB(k) = 2 \times \left(\frac{(\frac{1+\sqrt{5}}{2})^k - (\frac{1-\sqrt{5}}{2})^k}{\sqrt{5}} \right)$$

et donc

$$nbap(k) = 2 \times \left(\frac{(\frac{1+\sqrt{5}}{2})^k - (\frac{1-\sqrt{5}}{2})^k}{\sqrt{5}} \right) - 2.$$

On peut vérifier que c'est correct sur l'exemple $fibo2(6)$ dans lequel on compte 14 appels :

$$NB(1) = NB(2) = 2,\ NB(3) = 4,\ NB(4) = 6,\ NB(5) = 10,\ NB(6) = 16$$

d'où $nbap(6) = 14$.

Méthode 3. On est donc tenté de rechercher une autre méthode de résolution moins coûteuse. Une solution passe par l'utilisation d'un tableau dans lequel on enregistre les valeurs déjà calculées de la suite, de façon à les utiliser lors du calcul d'un nouvel élément. On va avoir un algorithme itératif et un tableau :

```
fonction fibo3 (ent n) résultat ent ;
var ent i ;
début
    T[1] ← 1 ; T[2] ← 1 ;
    pour i de 3 à n faire
        T[i] ← T[i − 1] + T[i − 2]
    fait ;
    résult ← T[n]
fin.
```

Cette solution donne le bon résultat et ce de façon beaucoup plus rapide que la

précédente. Cependant, en observant cet algorithme, on constate que l'utilisation du tableau n'est pas très appropriée dans la mesure où on n'utilise que les deux dernières valeurs enregistrées, d'où une nouvelle version qui ne va utiliser que deux variables auxiliaires :

fonction *fibo4* (ent n) **résultat** ent ;
var ent x, y, z ;
début
  $x \leftarrow 1$; $y \leftarrow 1$;
  pour i **de** 3 **à** n **faire**
    $z \leftarrow x + y$; co calcul de $f(n)$ à partir de $f(n-1) = y$ et $f(n-2) = x$ fco ;
    $x \leftarrow y$;
    co x contiendra la valeur actuelle de y i.e. vaudra $f(n-2)$ fco ;
    $y \leftarrow z$;
    co y contiendra la valeur actuelle de z i.e. vaudra $fib(n-1)$ fco
  fait ;
  résult $\leftarrow z$
fin.

Méthode 4. On pose : $u(n) = fib(n)$ et $v(n) = fib(n+1)$.

(1) Montrer que le problème posé se ramène alors à celui de la résolution d'une équation matricielle récurrente (où $V(i)$ est un vecteur colonne à deux éléments et F une matrice carrée 2×2) du type :

$V(0) =$ constante
$V(n) = F \times V(n-1)$

En particulier, on exhibera le vecteur V(0) et la matrice F.

Le vecteur initial s'écrit :

$$V(0) = \begin{bmatrix} u(0) \\ v(0) \end{bmatrix} = \begin{bmatrix} fib(0) \\ fib(1) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

et on a la matrice :

$$F = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

ce qui conduit bien à :

$$V(n) = \begin{bmatrix} u(n) \\ v(n) \end{bmatrix} = F \times V(n-1) = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} u(n-1) \\ v(n-1) \end{bmatrix}$$

soit

$$u(n) = v(n-1) = fib(n)$$
$$v(n) = u(n-1) + v(n-1) = fib(n-1) + fib(n) = fib(n+1).$$

(2) Montrer que la solution de cette équation de récurrence s'écrit

$$V(n) = F^n \times V(0).$$

On a donc :

$$\begin{aligned} V(n) &= F \times V(n-1) \\ &= F \times (F \times V(n-2)) \\ &= \dots \\ &= F^n \times V(0). \end{aligned}$$

(3) En déduire un algorithme calculant $fib(n)$ de type diviser pour règner de complexité $log_2(n)$ en termes de nombre de multiplications/additions d'entiers (ou de nombre de multiplications de matrices 2×2).

L'idée est de calculer une puissance quelconque d'une matrice carrée (ici 2×2) en $log_2(n)$. Le principe repose sur le fait que :

$$F^n = F^{\frac{n}{2}} \times F^{\frac{n}{2}} \times G$$

avec

$$G = \begin{cases} I \text{ (matrice identité) si } n \text{ est pair} \\ F \text{ si } n \text{ est impair.} \end{cases}$$

D'où l'algorithme ci-dessous pour calculer $fib(n)$:

```
fonction fibo (ent n) résultat ent ;
var mat22 T ;
   fonction pmat22enlog (ent n) résultat mat22 ;
   var mat22 T1, T2 ;
   début
      si n = 1 alors T2 ← ((0, 1), (1, 1))
      sinon T1 ← pmat22enlog (n div 2) ;
         si n pair alors T2 ← prodmat(T1, T1)
         sinon T2 ← prodmat(prodmat(T1, T1), F)
         fsi
      fsi ;
   résult ← T2
fin ;
début
   T ← pmat22enlog(n) ;
   résult ← T[1, 2]
fin.
```

Remarque : Compte tenu du fait que $V(n) = \begin{bmatrix} fib(n) \\ fib(n+1) \end{bmatrix}$, on pourrait calculer $V(n-1)$ et non pas $V(n)$ en rendant comme résultat sa seconde composante au lieu de sa première.

n	$nbprodmat$
1	0
$2 = 2^1$	$1 = log_2(2)$
$3 = 2^2 - 1$	$2 = 2(2-1)$
$4 = 2^2$	$2 = log_2(4)$
5	3
6	3
$7 = 2^3 - 1$	$4 = 2(3-1)$
$8 = 2^3$	$3 = log_2(8)$

(4) Donner un minorant et un majorant (en explicitant les cas où ils sont atteints) de la complexité exacte de cette procédure en terme de nombre de multiplications de matrices 2×2.

Si $n = 2^k$, on effectue donc exactement k produits de matrices. Par contre, si $n = 2k-1$, il faut à chaque étape faire deux produits de matrices, d'où $2(k-1)$ produits de matrices sont réalisés.

On a donc $nbprodmat \geq sup(log(n))$ et $nbprodmat \leq 2(sup(log(n)) - 1)$. Les premières valeurs du nombre de produits de matrices 2 x 2 sont données dans le tableau ci-dessus.

Cet exemple assez détaillé a pour but de mettre en évidence différents modes de résolution d'un même problème. On verra par la suite que les algorithmes $fibo2$ et $fibo3$ relèvent de méthodes de conception "générales" (diviser pour règner et programmation dynamique).

Chapitre 2

Diviser pour régner

2.1 Principe, usage de la récursivité

2.1.1 *Idée*

Pour résoudre un problème de taille n, on le divise en problèmes identiques (de même nature) mais de taille strictement inférieure et on exhibe un (ou des) cas élémentaires pour lesquelles on sait résoudre directement le problème. En toute rigueur, il faut s'assurer que l'on aboutira uniquement sur des cas élémentaires.

2.1.2 *Mise en œuvre*

Le mécanisme précédent trouve directement un modèle de mise en œuvre : la procédure récursive. La transformation est immédiate : le cas général de division engendre un ou plusieurs appels récursifs avec un paramètre de taille en décroissance ; les cas élémentaires correspondent à l'arrêt de la récursivité. On adoptera donc une écriture systématiquement fondée sur un test d'arrêt. Si on a bien spécifié la division du problème initial — en particulier les cas élémentaires —, la procédure récursive produite ne peut qu'être correcte.

2.1.3 *Modèle de décomposition et exemples*

$$Pb(n) \rightarrow a\ Pb(n_i) + f'(n) \text{ avec } a \geq 1,\ n_i < n$$

où $f'(n)$ correspond au traitement nécessaire pour diviser le problème et le cas échéant rassembler les résultats rendus par les sous-problèmes engendrés ; il fait souvent appel à des opérations fondamentales au sens de l'évaluation de la complexité.

Example 2.1. $Pb(n) \rightarrow Pb(n-1) + f'(n)$

$$\begin{cases} T(n) = T(n-1) + f(n) \\ T(1) = c \end{cases}$$

Solution :

$$T(n) = c + \sum_{i=2}^{n} f(i).$$

si $f(i) =$ constante (d) (recherche séquentielle)

$$T(n) = c + (n-1)d \in \theta(n).$$

si $f(i) = i$,

$$T(n) = c + ((n-1)(n+2))/2 \in \theta(n^2).\diamond$$

Example 2.2. $Pb(n) \rightarrow Pb(n/2) + g'(n)$

$$\begin{cases} T(n) = T(\frac{n}{2}) + g(n) \\ T(1) = c \end{cases}$$

Solution :

$$T(n) = c + \sum_{i=0}^{log_2(n)-1} g(\frac{n}{2^i}).$$

si $g(i) =$ constante (d) (recherche dichotomique)

$$T(n) = c + d.log_2(n) \in \theta(log_2(n)) \quad \text{(complexité logarithmique)}.$$

si $g(i) = i$,

$$T(n) \in \theta(n) \quad \text{(complexité linéaire)}.\diamond$$

Example 2.3. Les tours de Hanoi.

Ici il faut bien vérifier que jamais on ne met un anneau sur un plus petit.

$$Pb(n) \rightarrow 2Pb(n-1) + 1$$
$$Pb(1) \text{ fait appel à un seul déplacement}$$

donc $T(n) = 2^n - 1$ (complexité exponentielle).$\diamond$

On voit sur ces exemples que toutes les stratégies de division ne conduisent pas à la même classe de complexité.

2.2 Complexité pour la réduction logarithmique

L'exemple 2.2 précédent en est une illustration dans un cas particulier.

2.2.1 *Modèle général*

$$Pb(n) \rightarrow a\, Pb(\frac{n}{b}) + f(n).$$

$$\begin{cases} T(n) = a\, T(\frac{n}{b}) + f(n) \\ T(1) = d \end{cases}$$

Divers cas peuvent se produire selon le comportement de la fonction $f(n)$ qui ne fait que traduire le mécanisme de division propre au problème traité.

Cas 1. $f(n) = c$

On pose $n = b^k$ donc $Log(n) = k\, Log(b)$, $k = \frac{Log(n)}{Log(b)} = log_b(n)$ et

$$\begin{aligned} Log(a^k) &= k\, Log(a) \\ &= \frac{Log(n)}{Log(b)} * Log(a) \\ &= \frac{Log(a)}{Log(b)} * Log(n) \\ &= log_b(a)\, Log(n) \\ &= Log(n^{log_b(a)}) \end{aligned}$$

Donc $a^k = n^{log_b(a)}$.

On démontre facilement que

$$T(n) = a^k\, d + c \sum_{i=0}^{k-1} a^i.$$

La valeur de la somme dépend de a.

Si $a = 1$, $T(n) \in \theta(log_b(n))$.

Si $a > 1$, $T(n) \in \theta(n^{log_b(a)})$.

Cas 2. $f(n) = c\, n$

On peut démontrer que :

$$T(n) = a^k\, d + c \sum_{i=1}^{k} a^{k-i}\, b^i.$$

La valeur de la somme dépend de celles de a et b.

Si $a = b$, $T(n) \in \theta(n\, log_b(n))$.

Si $a < b$, $T(n) \in \theta(n)$.

Si $a > b$, $T(n) \in \theta(n^{log_b(a)})$.

Cas 3. $f(n) = c\, n^{\alpha}$

On peut démontrer que :

$$T(n) = a^k\, d + c \sum_{i=1}^{k} a^{k-i}\, b^{i\alpha}.$$

La valeur de la somme dépend de celles de a, b et α. On se retrouve dans la même situation que précédemment avec b^{α} jouant le rôle de b.

Si $a = b^{\alpha}$, $T(n) \in \theta(n^{\alpha}\, log_b(n))$.

Si $a < b^{\alpha}$, $T(n) \in \theta(n^{\alpha})$.

Si $a > b^{\alpha}$, $T(n) \in \theta(n^{log_b(a)})$.

2.2.2 *Illustrations simples*

2.2.2.1 *Recherche dichotomique et généralisation*

On s'intéresse à la recherche d'un élément dans un tableau trié, on emploie donc une méthode dichotomique dont le modèle de division est :

$$Pb(n) \rightarrow Pb(\frac{n}{2}) + f(n)$$

où $f(n)$ correspond à la recherche du demi-tableau où se poursuit la recherche, et $Pb(1)$ correspond au problème élémentaire.

On a donc une complexité

$$\begin{cases} T(n) = T(\frac{n}{2}) + 1 \\ T(1) = 1, \end{cases}$$

système dont la solution est $T(n) = 1 + log_2(n)$ (cas 1 avec $a = 1$, $b = 2$, $c = 1$ et $d = 1$, donc une complexité logarithmique. Or, on peut penser que bien que ne changeant pas de classe, une complexité $log_b(n)$ avec $b > 2$ améliorerait les performances de cette procédure. Que représente b ? En fait, le nombre de sous-tableaux dans lesquels peut se poursuivre la recherche avant que le "bon" sous-tableau ait été déterminé. Si on a b sous-tableaux, il faut effectuer $(b-1)$ tests sur la valeur recherchée pour trouver le sous-tableau où on poursuit la recherche, on a donc une complexité :

$$\begin{cases} T(n) = T(\frac{n}{b}) + (b-1) \\ T(1) = 1 \end{cases}$$

soit $T(n) = 1 + (b-1)\, log_b(n)$ (généralisation de la formule précédente où on avait $b = 2$).

On peut regarder ce qui se passe quand b devient très grand (tend vers n notamment). Alors $T(n)$ tend vers n, ce qui montre que l'on change de classe pour une moins bonne. La morale de l'affaire est donc que la dichotomie classique est la plus efficace et qu'il faut se méfier des apparences.

2.2.2.2 *Recherche simultanée du minimum et du maximum d'un tableau*

Une première solution consiste à balayer le tableau et remplacer *min* ou *max* par la valeur de l'élément courant s'il y a lieu.

```
procédure Mm1 (ent i, j ; var ent M, m) ;
début
   M ← T[i] ; m ← T[i] ;
   si j > i alors
      pour k de (i + 1) à j faire
         si T[k] < m alors m ← T[k]
         sinon si T[k] > M alors M ← T[k] fsi
         fsi
      fait
   fsi
fin.
```

Remarque : cette procédure n'utilise pas la stratégie DpR.

Etude de complexité : le baromètre est ici le nombre de comparaisons sur les éléments du tableau T ; ce nombre dépend de la configuration du tableau T ; au mieux on a $(j - i)$ comparaisons quand T est trié par ordre décroissant ; au pire on a $2 * (j - i)$ comparaisons quand T est trié en ordre croissant.

On peut remarquer que ces valeurs sont atteintes pour les cas de tri inverse si on fait les tests dans l'ordre inverse.

Donc, la complexité de *Mm1* est linéaire.

Une solution utilisant DpR peut être fondée sur le principe suivant : on découpe le tableau considéré en deux sous-tableaux dont on recherchera le minimum et le maximum ; le problème élémentaire est atteint ici quand on a un tableau de un élément qui est alors le minimum et le maximum du sous-tableau considéré. La fonction générique f' consiste ici à comparer au retour de ces deux appels les valeurs des deux minima et maxima retournés, d'où l'algorithme ci-dessous :

```
procédure Mm2 (ent i, j ; var ent M, m) ;
var ent mil, hmin, hmax, bmin, bmax ;
   co mil est l'indice du milieu du tableau
      hmin et hmax sont le min et le max du demi-tableau haut
      bmin et bmax sont le min et le max du demi-tableau bas
   fco ;
début
   si i = j alors
      M ← T[i] ; m ← T[i]
   sinon
```

```
        mil ← (i + j) div 2 ;
        Mm2(i, mil, hmax, hmin) ;
        Mm2(mil + 1, j, bmax, bmin) ;
        si bmin < hmin alors m ← bmin sinon m ← hmin fsi ;
        si hmax < bmax alors M ← bmax sinon M ← hmax fsi ;
    fsi
fin.
```

Etude de complexité : dans cette version, les comparaisons sur des éléments de T ont disparu et leur équivalent devient des comparaisons entre minima et maxima des sous-tableaux. En conséquence, on a une réduction logarithmique puisque le modèle de division est :

$$\begin{cases} Pb(n) \to 2\,Pb(\frac{n}{2}) + \text{ calcul } min \text{ et } max \text{ global,} \\ Pb(1) \text{ élémentaire} \end{cases}$$

Soit $T(n) = 2\,T(\frac{n}{2}) + 2$ et $T(1) = 0$. Par identification ($a = 2$, $b = 2$, $c = 2$, $d = 0$), on se trouve dans le cas 1 avec $a > 1$, donc on a une complexité en $n^{log_b(a)}$, donc ici linéaire.

Conclusion : cette méthode plus sophistiquée ne permet pas d'atteindre une classe meilleure. De plus, on peut penser qu'une analyse plus poussée montrerait que la mécanique mise en œuvre étant plus complexe (appels récursifs notamment), la seconde méthode est moins efficace que la première.

2.2.2.3 *Multiplication de deux polynômes*

On veut multiplier deux polynômes. Les opérations élémentaires pour mesurer la complexité sont la multiplication et l'addition des coefficients.

La multiplication de deux polynômes $P = \sum_{k=0}^{n} p_k X^k$ et $Q = \sum_{k=0}^{m} q_k X^k$ est donnée par la formule :

$$PQ = \sum_{k=0}^{n+m} \sum_{i+j=k} p_i q_j X^k.$$

L'algorithme correspondant est :

```
pour i de 0 à n + m faire C[i] ← 0 ; fait
pour i de 0 à n faire
    pour j de 0 à m faire
        C[i + j] ← C[i + j] + A[i] ∗ B[j]
    fait
fait
```

Il apparaît que le coût global est donc en $\theta(n \cdot m)$.

On s'intéresse à la façon suivante de décomposer un polynôme. Pour tout entier k,

si on a $P = RX^k + S$ et $Q = TX^k + U$, alors le produit s'écrit :

$$PQ = X^{2k}RT + X^k(RU + ST) + SU. \tag{2.1}$$

Soit $P = \sum_{k=0}^{n} p_k X^k$. On a :

$$\begin{aligned} P &= \sum_{i=0}^{\lfloor n/2 \rfloor} p_i X^i + \sum_{i=1+\lfloor n/2 \rfloor}^{n} p_i X^i \\ &= \sum_{i=0}^{\lfloor n/2 \rfloor} p_i X^i + X^{1+\lfloor n/2 \rfloor} \sum_{i=0}^{n-1-\lfloor n/2 \rfloor} p_{i+1+\lfloor n/2 \rfloor} X^i. \end{aligned}$$

On pose alors $S = \sum_{i=0}^{\lfloor n/2 \rfloor} p_i X^i$ et $R = \sum_{i=0}^{n-1-\lfloor n/2 \rfloor} p_{i+1+\lfloor n/2 \rfloor} X^i$. R et S sont alors deux polynômes de degré au plus $\lfloor n/2 \rfloor$ et on a :

$$P = RX^{1+\lfloor n/2 \rfloor} + S.$$

On définit de même les polynômes T et U pour Q :

$$Q = TX^{1+\lfloor n/2 \rfloor} + U$$

en supposant $m \leq n$. On a :

$$PQ = RTX^{2+2\lfloor n/2 \rfloor} + (RU + ST)X^{1+\lfloor n/2 \rfloor} + SU.$$

En négligeant la multiplication par un X^j, cette méthode requiert quatre multiplications de polynômes plus petits : RT, RU, ST, et SU, ainsi que des additions de polynômes de degré au plus n — ce qui coûte $\theta(n)$. En prenant $n = m = 2^k$, on obtient une complexité :

$$\begin{cases} T(n) = 4T(n/2) + \theta(n) \\ T(0) = 1 \end{cases}$$

Donc on se trouve dans le second cas et par identification ($a = 4$, $b = 2$), on déduit que $T(n) \in \theta(n^{log_b(a)}) = \theta(n^2)$. On ne change pas de classe par rapport à la méthode usuelle.

Or Karatsuba (1960) a remarqué que, si on écrit le produit (2.1) sous la forme :

$$PQ = X^{2k}RT + X^k((R + S)(T + U) - (RT + SU)) + SU \tag{2.2}$$

alors le produit ne requiert que trois multiplications (au lieu de quatre naïvement) de polynômes plus petits : RT, SU, et $(R + S)(T + U)$. Cette remarque ouvre la voie à une implémentation en diviser pour régner plus efficace.

On a :

$$\begin{cases} T(n) = 3T(n/2) + \theta(n), \\ T(0) = 1 \end{cases}$$

D'où $T(n) \in \theta(n^{log_2(3)}) = \theta(n^{1.59})$.

2.2.2.4 *Produit de matrices carrées d'ordre n ($n = 2^k$)*

De façon classique, le produit de deux matrices carrées réclame 3 boucles imbriquées variant de 1 à n, puisque chacun des n^2 éléments de la matrice résultat est calculé par produit scalaire d'une ligne et d'une colonne de n éléments chacun. On a donc une complexité en $O(n^3)$ opérations élémentaires (additions et multiplications).

La méthode présentée ci-après connue sous le nom d'algorithme de Strassen illustre la troisième situation. Elle se fonde sur le calcul suivant :

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})\, B_{11} \\ R &= A_{11}\, (B_{11} - B_{22}) \\ S &= A_{22}\, (B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})\, B_{22} \\ U &= (A_{22} - A_{11})(B_{11} + B_{12}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

et alors, on a :

$$\begin{aligned} C_{11} &= P + S - T + V \\ C_{12} &= R + T \\ C_{21} &= Q + S \\ C_{22} &= P + R - Q + U \end{aligned}$$

Le modèle de division fait donc appel à 7 sous-problèmes de produits de matrices de taille $n/2$ assortis de 18 additions de matrices et de copies de matrices intermédiaires (P, Q, ...) ce qui a une complexité en n^2. On arrive à :

$$\begin{cases} T(n) = 7T(n/2) + 18\, n^2 \\ T(1) = 1 \end{cases}$$

qui illustre le cas 3 vu précédemment. On aboutit à une complexité de ce calcul en $\theta(n^{log_2(7)}) = \theta(n^{2.81})$.

Chapitre 3

Essais successifs

Cette technique est également appelée : essais et erreurs (trial and errors) et retour arrière (backtracking).

3.1 Principe et types de problèmes traités

Il existe de nombreux problèmes n'ayant pas de solution "directe", c'est-à-dire pour lesquels on ne sait pas construire les solutions autrement qu'en examinant les solutions candidates (recherche a priori exhaustive). La recherche des solutions au problème de placement de n reines sur un échiquier $n \times n$ sans qu'elles ne soient en prise (posé par Gauss vers 1850 et qui ne l'a pas totalement résolu) en est une illustration.

Dans la suite, on va s'intéresser à trois types de problèmes : trouver toutes les solutions à un problème ou une solution quelconque ou encore la meilleure solution ce qui suppose que l'on définisse clairement la fonction d'évaluation du problème concerné.

De façon générale, on représentera les solutions candidates par un vecteur $(x_1, \ldots, x_n)$ dépendant de la modélisation du problème. Si on admet que x_i prend ses valeurs dans un ensemble S_i fini de cardinalité m_i, on a :

$$\prod_{i=1}^{n} m_i = m_1 \times \ldots m_n$$

solutions candidates.

Un des principes de base à utiliser de sorte que cette technique soit viable est de ne pas construire et donc de ne pas examiner des solutions dont on peut dire a priori qu'elles ne conviennent pas. Le principe de construction des solutions va obéir aux règles suivantes :

- construction du vecteur pas à pas (x_1 puis x_2 etc),
- quand on choisit une valeur pour le composant i, on vérifie que le vecteur partiel $(x_1, \ldots, x_i)$ satisfait un critère partiel d'acceptation intégrant deux

aspects : la valeur est compatible avec les valeurs choisies précédemment et le vecteur partiel est un début possible de solution ; si tel n'est pas le cas, on n'examinera pas les solutions candidates ayant ce début ce qui constitue une économie.

3.2 Exemple introductif

Reprenons l'exemple cité précédemment et cherchons à définir l'espace de toutes les solutions candidates. Il faut tout d'abord choisir une représentation de ces solutions sous forme d'un vecteur. Ici, il s'agit d'un vecteur de n doublets (x_i, y_i) représentant les positions des n reines sur l'échiquier. Le critère d'acceptation P d'une solution peut se décomposer en trois sous-critères :

P1 : il n'existe pas deux reines sur la même ligne,
P2 : il n'existe pas deux reines sur la même colonne,
P3 : il n'existe pas deux reines sur la même diagonale.

Le nombre de solutions candidates est donc la façon de choisir n cases dans un échiquier de n^2 cases soit

$$C(n,\, n^2) = \frac{n^2!}{(n!\,(n^2-n)!)}.$$

Pour $n = 4$, on obtient $\frac{16!}{4!\,12!} = \frac{13*14*15*16}{2*3*4}$, soit 1820 solutions.

Or, on a vu qu'il ne peut y avoir deux reines sur la même ligne ; on peut donc imposer une certaine forme aux doublets en imposant que chacun d'eux corresponde à une ligne distincte des autres, soit : $\{(1,\, x_1),\, (2,\, x_2),\, \ldots,\, (n,\, x_n)\}$. On se ramène alors à n^4, soit avec $n = 4$, 256 solutions candidates.

On arrive à la procédure suivante :

procédure *solcandidates1* ;
var ent x_1, x_2, x_3, x_4 ;
début
  $S \leftarrow \emptyset$;
  pour x_1 **de** 1 **à** 4 **faire**
    pour x_2 **de** 1 **à** 4 **faire**
      pour x_3 **de** 1 **à** 4 **faire**
        pour x_4 **de** 1 **à** 4 **faire**
          $S \leftarrow S \cup \langle (1,\, x_1),\, (2,\, x_2),\, (3,\, x_3),\, (4,\, x_4) \rangle$
        fait
      fait
    fait
  fait
fin ;

choisir dans S les éléments satisfaisant P2 et P3 ;

On peut aussi remarquer que comme deux reines ne doivent pas être sur la même colonne, il est possible de restreindre encore l'espace des solutions candidates qui doivent être une permutation de $(1, \ldots, n)$, ce qui pour $n = 4$ conduit à seulement $4! = 24$ solutions candidates au lieu des 1820 initiales ! (on est passé de $C(n, n^2)$ à n^n, puis à $n!$).

Le modèle de programmation itérative associé est :

procédure *solcandidates2* ;
var ent x_1, x_2, x_3, x_4 ;
début
  $S \leftarrow \emptyset$;
  pour x_1 **de** 1 **à** 4 **faire**
    pour x_2 **de** 1 **à** 4 **faire**
      si $x_1 \neq x_2$ **alors**
        pour x_3 **de** 1 **à** 4 **faire**
          si $x_3 \neq x_1$ **et** $x_3 \neq x_2$ **alors**
            pour x_4 **de** 1 **à** 4 **faire**
              si $x_4 \neq x_1$ **et** $x_4 \neq x_2$ **et** $x_4 \neq x_3$ **alors**
                $S \leftarrow S \cup \langle (1, x_1), (2, x_2), (3, x_3), (4, x_4) \rangle$
              fsi
            fait
          fsi
        fait
      fsi
    fait
  fait
fin ;

choisir dans S les éléments vérifiant P3 ;

Cette solution élimine des pas d'itération par rapport à la précédente, cependant on n'élimine pas certaines solutions partielles qui ne peuvent conduire à aucune solution candidate complète ($x_1 = 1$, $x_2 = 2$ par exemple). Pour pallier cet inconvénient, il faut faire intervenir P3 le plus tôt possible, d'où l'algorithme itératif final :

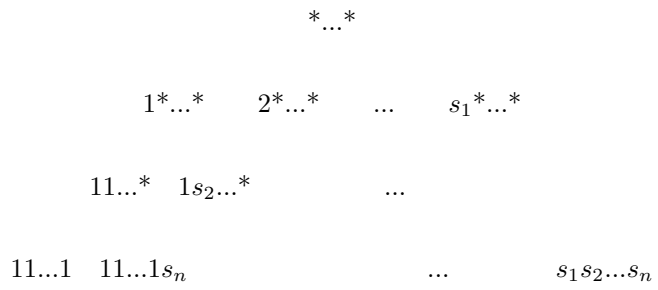
procédure *solcandidates3* ;
var ent x_1, x_2, x_3, x_4 ;
début
  $S \leftarrow \emptyset$;
  pour x_1 **de** 1 **à** 4 **faire**
    pour x_2 **de** 1 **à** 4 **faire**
      si $x_1 \neq x_2$ **et** P3 **alors**
        pour x_3 **de** 1 **à** 4 **faire**

```
                    si x3 ≠ x1 et x3 ≠ x2 et P3 alors
                        pour x4 de 1 à 4 faire
                            si x4 ≠ x1 et x4 ≠ x2 et x4 ≠ x3 et P3 alors
                                S ← S ∪ ⟨(1, x1), (2, x2), (3, x3), (4, x4)⟩
                            fsi
                        fait
                    fsi
                fait
            fsi
        fait
    fait
fin ;
```

De cette façon, on n'engendre effectivement les seules solutions candidates si on admet que la procédure P3 vérifie que la dernière reine placée n'est en prise avec aucune autre.

3.3 Représentation arborescente, limitation des développements

On peut voir l'ensemble de toutes les solutions candidates comme les feuilles d'une arborescence dont la trame est :



Dès lors, on va chercher un procédé produisant effectivement la partie de cette arborescence qui mène à des feuilles pouvant être des solutions acceptables.

3.4 Modèles de mise en oeuvre et récursivité

On peut s'intéresser à différents problèmes. Les solutions auront toutes comme base une procédure récursive ayant pour fonction principale de choisir la valeur de l'élément x_i.

3.4.1 *Recherche de toutes les solutions*

Le modèle est le suivant :

procédure *toutessol* (ent i) ;
var ... x_i ;
début
  calculer l'ensemble S_i des valeurs possibles de x_i ;
  pour x_i parcourant S_i **faire**
    si *satisfaisant*(x_i) **alors** *enregistrer* le choix x_i ;
      si *soltrouvée* **alors** *écriresolution*
      sinon *toutessol*(i+1)
      fsi ;
      défaire le choix x_i
    fsi
  fait
fin ;

Les éléments appelés *satisfaisant*, *enregistrer*, *soltrouvé*, *écriresolution* et *défaire* dépendent évidemment du problème traité.

On constate que cette procédure construit un arbre de solutions candidates conforme à ce qui a été suggéré auparavant (descendant gauche droit).

Example 3.1. Reprenons le problème du placement de n reines et déterminons le contenu de chacun des éléments précédents. D'après l'analyse faite précédemment, on sait que l'on ne peut mettre deux reines ni sur une même ligne, ni sur une même colonne. On va donc décider qu'à l'étape i, on met une reine sur la ligne i dans une colonne à déterminer.

valeurs possibles de x_i (S_i) : 1 à n ;

satisfaisant(x_i) : la $i^{ème}$ reine est en (i, x_i), la colonne x_i est libre, de même que les deux diagonales passant par (i, x_i) ;

enregistrer(x_i) : $POS[i] \leftarrow x_i$, la colonne x_i et les deux diagonales passant par (i, x_i) sont occupées ;

soltrouvée : on a placé la $n^{ème}$ reine ($i = n$) ;

écriresolution : afficher les n couples (i, $POS[i]$) ;

défaire(x_i) : la colonne x_i et les deux diagonales passant par la case (i, x_i) sont libérées.

d'où l'algorithme ci-dessous :

procédure *reines* (ent i) ;
var ent x_i ;

début
  pour x_i **de** 1 **à** n **faire**
    si $A[x_i]$ **et** $B[i+x_i]$ **et** $C[i-x_i]$ **alors**
      $X[i] \leftarrow x_i$; $A[x_i] \leftarrow false$; $B[i+x_i] \leftarrow false$; $C[i-x_i] \leftarrow false$;
      si $(i = n)$ **alors** *écriresolution*
      sinon *reines*$(i+1)$
      fsi ;
      $A[x_i] \leftarrow true$; $B[i+x_i] \leftarrow true$; $C[i-x_i] \leftarrow true$;
    fsi
  fait
fin ;

où :

- $A[1 \ldots n]$ est un tableau de booléens servant à représenter la non-occupation des colonnes, initialisé à *vrai* ;
- $B[2 \ldots 2n]$ est un tableau de booléens servant à représenter la non-occupation des diagonales descendantes, initialisé à *vrai* ;
- $C[1-n \ \ldots \ n-1]$ est un tableau de booléens servant à représenter la non-occupation des diagonales montantes, initialisé à *vrai* ;

Appel initial : *reines*(1) ;

3.4.2 *Recherche d'une solution quelconque*

Modification triviale du modèle précédent (on stoppe la boucle dès qu'on a trouvé une solution).

3.4.3 *Recherche d'une solution optimale*

- notion d'optimalité et fonction d'évaluation
- conservation de la meilleure solution trouvée quand une nouvelle solution est exhibée
- réduction de l'espace des choix par introduction d'une condition d'élagage dans *satisfait* : voir plus loin.

Nouvel algorithme :

procédure *solopt* (ent i) ;
var ent x_i ;
début
  calculer S_i ;
  pour x_i parcourant S_i **faire**
    si *satisfaisant*(x_i) **alors** *enregistrer*(x_i) ;
      si *soltrouvée* **alors**
        si *meilleure* **alors** $Y \leftarrow X$; *majvalopt* fsi
      sinon

```
            si encorepossible alors solopt(i + 1) fsi
        fsi ;
        défaire(x_i)
    fsi
  fait
fin ;
```

Appel :

$Y \leftarrow valinit$; $valopt \leftarrow eval(Y)$; $solopt(1)$;

3.4.4 *Solutions "avec trous"*

Le principe général reste le même et donc le canevas d'algorithme aussi. La nouveauté réside dans le fait que lors du choix de valeur de x_i (donc dans la spécification de S_i) il y a un cas supplémentaire : "rien à faire".

Considérons le problème des guetteurs (nombre minimum de reines pour "couvrir" un échiquier, sans que les reines soient en prise). Pour explorer toute la combinatoire des candidats, il faut pouvoir ne pas mettre de reine dans une ligne, ce qui va correspondre à un choix additionnel (par rapport à la pose d'une reine). Techniquement, on va s'arranger pour que le codage soit cohérent avec les autres cas.

Analyse :

- vecteur, x_i, S_i : vecteur X de n cases tel que $X[i] = j$ signifie que l'on a mis une reine en colonne j sur la ligne i. Si $X(i] = 0$, cela signifie qu'il n'y a pas de reine sur la ligne i ; $S_i = \{0, 1, \ldots, n\}$;
- *satisfaisant*(x_i) : si $x_i \neq 0$: la colonne x_i doit être libre et les diagonales passant par (i, x_i) également ; sinon *vrai* ;
- *enregistrer*(x_i) : $X[i] \leftarrow x_i$; si $x_i > 0$, $nbreines \leftarrow nbreines + 1$; colonne x_i occupée, diagonales par (i, x_i) occupées, calculer le nouvel état des cases en prise ;
- *soltrouvée* : toutes les cases sont en prise ;
- *meilleure* : $nb\_reines < nb\_reines\_opt$;
- *encorepossible* : toutes les cases ne sont pas en prise et $i < n$;
- *défaire*(x_i) : si $x_i > 0$, libérer colonne et diagonales, $nb\_reines \leftarrow nb\_reines - 1$, calculer l'état des cases en prise.

L'algorithme est le suivant :

```
procédure guetteurs (ent i) ;
var ent x_i, j ; ens case nouvcasatt ;
début
  pour x_i de 0 à n faire
    si x_i = 0 alors
      X[i] ← 0 ; si i < n alors guetteurs(i + 1) fsi
```

```
        sinon
            si A[x_i] et B[i + x_i] et C[i − x_i] alors
                X[i] ← x_i ; A[x_i] ← faux ; B[i + x_i] ← faux ; C[i − x_i] ← faux ;
                nouvcases(i, x_i, nouvcasatt) ;
                nonatt ← nonatt − nouvcasatt ;
                nbreines ← nbreines + 1 ;
                si nonatt = ∅ alors
                    si nbreines < minreines alors
                        minreines ← nbreines ; gardersol(i)
                    fsi
                sinon
                    si i < n et nbreines < minreines − 1 alors guetteurs(i + 1) fsi
                fsi
                nbreines ← nbreines − 1 ;
                nonatt ← nonatt ∪ nouvcasatt ;
                A[x_i] ← vrai ; B[i + x_i] ← vrai ; C[i − x_i] ← vrai
            fsi
        fsi
    fait
fin ;

procédure gardersol (ent i) ;
var ent j ;
début
    pour j de 1 à i faire
        sopt[j] ← X[j]
    fait ;
    pour j de (i + 1) à n faire
        sopt[j] ← 0
    fait
fin ;

procédure nouvcases (ent i, j ; var ens case E) ;
var case c ;
début
    E ← ∅ ;
    pour tout c de (1 : n) × (1 : n) faire
        si c.lig = i ou c.col = j ou c.lig + c.col = i + j ou c.lig − c.col = i − j alors
            E ← E ∪ {c} ;
        fsi
    fait
    E ← E ∩ nonatt
fin ;
```

Déclarations, variables et appel :

type case **c'est struct ent** lig, col **fstruct** ;
type tabent **c'est tableau** $[1 : n]$ ent ;
type tabool1 **c'est tableau** $[1 : n]$ bool ;
type tabool2 **c'est tableau** $[2 : 2 * n]$ bool ;
type tabool3 **c'est tableau** $[-n + 1 : n - 1]$ bool ;
var ens case *nonatt* **init** $(1 : n) \times (1 : n)$;
var tabent X, *sopt* ;
var tabool1 A ; tabool2 B ; tabool3 C ;
var ent *nbreines*, *minreines*, n, j ;

lire(n) ;
$A \leftarrow vrai$; $B \leftarrow vrai$; $C \leftarrow vrai$; $nonatt \leftarrow (1 : n) \times (1 : n)$;
$minreines \leftarrow n + 1$; $nbreines \leftarrow 0$; guetteurs(1) ;
pour j **de** 1 **à** n **faire écrire**($sopt[j]$) **fait** ;

Quelques résultats intéressants :

côté échiquier	nbsol "n reines"	*minreines* "guetteurs"	nbsol "guetteurs"
1	1	1	1
2	0	1	4
3	0	1	1
4	2	3	16
5	10	3	16
6	4	4	120
7	40	4	8
8	92	5	728

3.5 Efficacité et conditions d'élagage

Le principe consiste à insérer (préférentiellement dans *encorepossible*) une ou des conditions testant l'intérêt que présente le sous-arbre dans lequel on va évoluer. On cherche à identifier des cas où toutes les feuilles du sous-arbre sont "perdantes" (ne peuvent mener à des solutions meilleures).

En particulier, lors de la recherche d'une solution optimale, on introduira une solution relative à la possibilité de faire mieux que la solution optimale courante.

Une condition d'élagage correspondra à un cas où on peut arrêter l'exploration. Du point de vue spécification de *encorepossible*, on utilisera la négation.

En général, il est conseillé de procéder en deux temps :

– un algorithme de base (spécification des "briques") sans se préoccuper d'efficacité/élagage,

– on introduit les conditions d'élagage.

Exemple illustratif simple : Sous-ensembles de somme donnée d'un ensemble d'entiers positifs.

Soit une suite de n entiers positifs w_1, ..., w_n et un nombre M positif donné. On cherche les sous-ensembles de $W = \{w_1, ..., w_n\}$ tels que la somme de leurs éléments soit égale à M.

Exemple : $n = 4$, $W = \{11, 13, 24, 7\}$, $M = 31$.

solution 1 : $\{11, 13, 7\}$
solution 2 : $\{24, 7\}$.

1. Faire l'analyse de ce problème sans se préoccuper d'élagage dans un premier temps et en se plaçant dans le cadre d'une "solution à trous".

Une solution est représentée par un vecteur de 0/1 précisant si l'élément de W de rang i est ou non dans la solution, soit dans notre exemple : $\langle 1, 1, 0, 1\rangle$ et $\langle 0, 0, 1, 1\rangle$. Il y a a priori 2^n solutions candidates (nombre de sous-ensembles d'un ensemble de n éléments).

Au niveau représentation des objets manipulés :
- domaine des x_i : $S_i = \{0/1\}$;
- W : tableau d'entiers $[1 : n]$;
- solution : tableau $X[1 : n]$;

Une solution est représentée par un vecteur X tel que $\sum W[k] * X[k] = M$.

vecteur, x_i, S_i : vecteur de valeurs 0/1

satisfaisant : $somcour + x_i * W[i] \leq M$

enregistrer : $X[i] \leftarrow x_i$; $somcour \leftarrow somcour + x_i * W[i]$

soltrouvée : $somcour = M$

encorepossible : $i < n$

défaire : $somcour \leftarrow somcour - x_i * W[i]$

2. Proposer des mécanismes d'élagage appropriés à ce problème.

On va étudier de façon plus approfondie la condition *satisfaisant*. Le vecteur partiel $\{x_1, ..., x_i\}$ est satisfaisant si on a encore une chance d'atteindre M en prenant tous les éléments restants de W, soit :

$$W[1] \cdot X[1] + \ldots + W[i-1] \cdot X[i-1] + W[i] \cdot X[i] + W[i+1] + \ldots + W[n] \geq M$$

Pour aider à ce calcul, on va introduire la variable rk représentant la somme des valeurs restant dans W (de i+1 à n). Cette variable sera mise à jour dans enregistrer et défaire.

3. Donner le code d'un algorithme associé.

procédure *sommesousens* (ent i)
var ent x_i ;
début
    pour x_i **de** 0 **à** 1 **faire**
        si $somcour + W[i] * xi + rk \geq M$ **et** $somcour + W[i] * x_i \leq M$ **alors**
            $X[i] \leftarrow x_i$;
            $somcour \leftarrow somcour + W[i] * x_i$;
            $rk \leftarrow rk - W[i+1]$; **co** $W[n+1]$ vaut 0 **fco**
            si $i < n$ **alors** $sommesousens(i+1)$
            sinon co compte tenu du premier test, on a une solution **fco**
                conserversolution
            fsi ;
            $somcour \leftarrow somcour - W[i] * x_i$;
            $rk \leftarrow rk + W[i+1]$
        fsi
    fait
fin

appel :
$somcour \leftarrow 0$;
$rk \leftarrow W[2] + ... + W[n+1]$;
$sommesousens(1)$;

Remarque : on trouve les solutions quand on a un vecteur de longueur n ; on pourrait évidemment tester que l'on atteint M et s'arrêter à ce moment en complétant avec des 0.

Chapitre 4

Programmation dynamique

4.1 Introduction

Quelques mots-clés (aspects) :
- formule de récurrence
- calcul tabulaire.

Exemple : la suite de Fibonacci (formule de récurrence donnée).

Remarque : qui dit récurrence dit possibilité d'utiliser DpR. Les différences entre programmation dynamique et DpR sont au moins au nombre de deux :
- en programmation dynamique, on calcule *toujours* une grandeur (valeur d'une fonction = nombre). On ne peut donc pas traiter, par exemple, le problème des tours de Hanoï.
- le calcul n'est pas effectué de la même façon. En DpR, on calcule "de haut en bas" (récursif) alors qu'en programmation dynamique, c'est tabulaire et ceci correspond plutôt à un calcul "de bas en haut". L'idée est de remplacer la récursivité liée à la relation de récurrence par une itération ; on mémorise la solution à un sous-problème et ensuite on consulte la table plutôt que de recalculer un résultat.

Remarque : le prix à payer en termes de place mémoire peut être élevé, d'où compromis espace/temps.

Exemple du calcul de $Fib(27)$ avec $Fib(1) = Fib(2) = 1$. Dans le cas DpR, on développe un arbre et on peut se retrouver à calculer plusieurs fois les mêmes valeurs. En programmation dynamique, on n'a besoin de sauvegarder que les deux valeurs permettant de calculer $Fib(n)$.

Par rapport à DpR, la programmation dynamique est préférable car plus efficace.

Le point commun à DpR et à programmation dynamique concerne l'identification du modèle de division qui, alors, se ramène à une équation de récurrence.

Exemple : Dans le cas de la suite de Fibonacci, on a :

$$\begin{cases} Pb(n) \to Pb(n-1) \oplus Pb(n-2), \text{ pour } n > 2, \\ Pb(1) \text{ et } Pb(2) \text{ élémentaires.} \end{cases}$$

4.2 Principe de la programmation dynamique

La programmation dynamique "pure" a pour but de calculer la valeur optimale d'une fonction.

Le point de départ consiste à identifier une formule de récurrence du type :

$$fopt(n) = ... \quad fopt(n-k) \quad ... \quad fopt(n-q) \quad ...$$

La valeur optimale de rang n se calcule à partir de valeurs optimales à d'autres rangs. Cette récurrence doit être *complète*.

On applique le *principe d'optimalité de Bellman* qui dit que toute sous-solution d'une solution optimale est optimale. On va donc traiter des problèmes où la solution recherchée est obtenue par composition des valeurs optimales associées à des sous-problèmes, ce que traduit la formule de récurrence. Une fois la formule trouvée (ensemble complet d'équations), il faut identifier une structure tabulaire associée dans laquelle tout élément correspond à la valeur optimale d'un sous-problème. On doit alors trouver une procédure de remplissage utilisant la récurrence telle que le calcul d'une case (un élément de la structure) ne fasse appel qu'à des éléments déjà calculés, jusqu'à remplir la case correspondant à la valeur cherchée.

4.3 Exemple : le World Series

On considère deux équipes A et B de force égale. Le gagnant est la première équipe qui remporte p parties. Les parties sont indépendantes.

On doit calculer $P(i,\, j) = \text{proba}(A \text{ gagne} \mid \text{pour vaincre, } A \text{ doit encore remporter } i \text{ parties tandis que pour vaincre, } B \text{ doit encore en remporter } j)$.

On peut montrer que l'on aboutit à la récurrence :

$$\begin{aligned} &P(0,\, j) = 1,\ \forall j > 0, \\ &P(i,\, 0) = 0,\ \forall i > 0, \\ &P(i,\, j) = \frac{P(i-1,\, j) + P(i,\, j-1)}{2},\ \forall i,\, j > 0. \end{aligned}$$

Complexité par une procédure récursive "normale" :

$$\begin{cases} T(n) \leq 2\, T(n-1) + d \\ T(1) = c. \end{cases}$$

en notant $n = i + j$.

D'où

$$T(n) \leq 2^{n-1}\, c + (2^{n-1} - 1)\, d.$$

Donc $T(n)$ est en $O(2^n)$ (complexité exponentielle).

En développant l'arbre des appels générés par $P(2, 3)$, on voit que $P(1, 2)$ est calculé

deux fois.

Analyse d'une autre solution, fondée sur la programmation dynamique

Soit un tableau de taille $n \times n$, où l'indice i sert à représenter les ordonnées (axe pointant vers le bas) et l'indice j les abscisses. Les diagonales montantes sont telles que $i + j = cte$.

On peut initialiser $P(i,\ 0)\ \forall i$ et $P(0,\ j)\ \forall j$ (problèmes élémentaires).

- diagonale 1 :
  - $P(1,\ 0)$ (init)
  - $P(0,\ 1)$ (init)
- diagonale 2 :
  - $P(2,\ 0)$ (init),
  - $P(1,\ 1)$: requiert les valeurs de $P(1,\ 0)$ et de $P(0,\ 1)$ déjà stockées,
  - $P(0,\ 2)$ (init).
- diagonale 3 :
  - $P(3,\ 0)$ (init),
  - $P(2,\ 1)$: requiert $P(1,\ 1)$ et $P(2,\ 0)$,
  - $P(1,\ 2)$: requiert $P(1,\ 1)$ et $P(0,\ 2)$,
  - $P(0,\ 3)$ (init)

d'où la procédure suivante :

```
fonction calproba (ent i, j) résultat réel ;
var ent d, k ;
début
   pour d de 1 à (i + j) faire
      P[0, d] ← 1 ; P[d, 0] ← 0 ;
      pour k de 1 à (d − 1) faire
         P[k, d − k] ← (P[k − 1, d − k] + P[k, d − k − 1])/2 ;
      fait ;
   fait ;
   résult ← P[i, j]   co se trouve sur la dernière diagonale fco
fin.
```

Complexité de cet algorithme :

$$\theta(n^2) \text{ avec } n = i + j \text{ (temporelle) + matrice triangulaire } \theta(n^2) \text{ (spatiale)}$$

au lieu de

$$O(2^n) \text{ (temporelle) } + \theta(n) \text{ (spatiale)}$$

(ce dernier terme correspondant à la longueur de la branche la plus longue, ce qui est tout ce qu'on a besoin de stocker en mémoire dans le cas DpR).

Exemple de calcul.

	0	1	2	3	4
0		1	1	1	1
1	0	1/2	3/4	7/8	15/16
2	0	1/4	1/2	11/16	
3	0	1/8	5/16		
4	0	1/16			

4.4 Exemple : le produit de matrices

On veut calculer la matrice M résultant du produit $M_1 \times \ldots \times M_n$ avec des conditions de bon sens sur les tailles des matrices : $nblignes(M_i) = nbcolonnes(M_{i-1})$.

Or, on observe que le coût du calcul varie (plus ou moins fortement) avec la façon de l'effectuer.

Example 4.1. Soit $M_1[10, 20] \times M_2[20, 50] \times M_3[50, 1] \times M_4[1, 100]$.

Première façon : de gauche à droite

Soit n (resp. m) le nombre de colonnes (resp. lignes) dans la première matrice, et k le nombre de colonnes dans la deuxième matrice (son nombre de lignes est bien sûr égal à n). La matrice résultante comporte m lignes et k colonnes.

Nombre de multiplications pour obtenir la matrice résultante : une case de la matrice résultante : n multiplications ; toutes les cases : $n * k * m$.

Sur l'exemple :

$$(10 * 20 * 50) + (10 * 50 * 1) + (10 * 1 * 100) = 10000 + 500 + 1000 = 11500.$$

Deuxième façon : $M_2 \times M_3 \rightarrow M_{23}$; $M_1 \times M_{23} \rightarrow M_{123}$. M_{23} est de taille [20, 1] et M_{123} de taille [10, 1]. On calcule finalement $M_{123} \times M_4$. Nombre de multiplications générées :

$$(20 * 50 * 1) + (10 * 20 * 1) + (10 * 1 * 100) = 1000 + 200 + 1000 = 2200.$$

On constate un rapport de 5 entre les deux méthodes.

Troisième façon : de droite à gauche

$M_3 \times M_4 \rightarrow M_{34}$; $M_2 \times M_{34} \rightarrow M_{234}$; On calcule finalement $M_1 \times M_{234}$. Nombre de multiplications générées :

$(50 * 1 * 100) + (20 * 50 * 100) + (10 * 20 * 100) = 5000 + 100000 + 20000 = 125000.$

On voit que c'est une très mauvaise idée.

Le problème à résoudre est celui de déterminer la *façon optimale* de faire la suite de produits.

Remarques

– on ne sait qu'effectuer des produits binaires, i.e., de deux matrices ;
– on ne peut pas modifier leur ordre !

On effectue donc forcément des produits de matrices adjacentes, un produit rendant une nouvelle matrice.

Etude de la combinatoire
– $(n-1)$ possibilités pour le premier produit ;
– $(n-2)$ possibilités pour le deuxième produit ;
– ...

donc on a $(n-1)!$ solutions possibles.

Cependant, on observe que deux séquences différentes peuvent faire apparaître un même produit. Par exemple :

$$M_1 \times M_2;\ M_3 \times M_4; M_{12} \times M_{34}$$
$$M_3 \times M_4;\ M_1 \times M_2;\ M_{12} \times M_{34}.$$

En fait, la notion pertinente est celle de parenthésage :

$$((M_1 \times M_2) \times (M_3 \times M_4)).$$

Question : nombre de parenthésages différents ?

$$(M_1 \times M_2 \times \ldots \times M_k) \times (M_{k+1} \times \ldots \times M_{n-1} \times M_n)$$

avec k pouvant varier de 1 à $n-1$ (choix "arbitraire").

On obtient :

$$\begin{cases} NBP(n) = \sum_{k=1}^{n-1} (NBP(k) * NBP(n-k)) \\ NBP(1) = 1. \end{cases}$$

Ce système d'équations correspond à la définition des *nombres de Catalan*. Ces

n	$NBP(n)$
1	1
2	2
3	6
4	18
5	54

derniers s'expriment également de la façon suivante :

$$\begin{cases} NBP(n) = \frac{1}{n} C_{2n-2}^{n-1}, \text{ pour } n > 1, \\ NBP(1) = 1. \end{cases}$$

Une approximation de $NBP(n)$ est

$$\frac{4^{n-1}}{n\sqrt{\pi\, n}}.$$

Donc le nombre de parenthésages est exponentiel. Comme la combinatoire est élevée, il est indispensable de rechercher (si possible) une solution plus praticable. On va tenter d'en trouver une grâce à la programmation dynamique, qui aura une complexité polynomiale (plus précisément en n^3).

Principe d'un calcul par programmation dynamique

On va chercher à calculer le coût optimal de produits de matrices de plus en plus longs.

Soit $copt(1,\ n)$ le coût optimal pour effectuer le produit des matrices M_1 jusqu'à M_n. De la même façon $copt(d,\ m)$ correspond au coût optimal pour effectuer le produit des matrices M_d jusqu'à M_m.

On connaît la façon optimale d'effectuer n'importe quel produit de deux matrices, c'est-à-dire $copt(k,\ k+1)$. On sait aussi calculer de façon optimale tout produit d'*une* matrice (il n'y a rien à faire).

On découpe en deux paquets :

$$(M_1 \times \ldots \times M_{k-1})\,(M_k \times \ldots \times M_n).$$

Cas général (correspondant à $copt(i,\ i+j)$) :

$$(M_i \times \ldots \times M_{k-1})\,(M_k \times \ldots \times M_{i+j}).$$

On note $D[i]$ le nombre de colonnes de la matrice M_i avec $i \in [0,\ n]$, D_0 étant le nombre de lignes de M_1. Le coût du produit d'une matrice ayant $nblignes(M_i)$ lignes et $nbcolonnes(M_{k-1})$ colonnes par une matrice ayant $nblignes(M_k)$ lignes et $nbcolonnes(M_{i+j})$ colonnes est égal à $D[i-1] * D[k-1] * D[i+j]$. On a donc :

$$copt(i,\ i+j) =$$
$$min_{(i+1)\leq k\leq (i+j)}\ (copt(i,\ k-1) + copt(k,\ i+j) + D[i-1] * D[k-1] * D[i+j])$$

pour $1 \leq j \leq n-1$ et $1 \leq i \leq n-j$. Pour avoir un système de récurrence complet, on doit ajouter le cas :

$$\forall i \in [1,\ n],\ copt(i,\ i) = 0.$$

La deuxième étape concerne la définition d'une structure tabulaire et d'une stratégie de remplissage (de façon à ce que la valeur d'une case ne fasse appel qu'à des cases déjà remplies).

Ici, la structure est un tableau à deux dimensions $COPT[1:n, 1:n]$. La solution au problème se trouve dans la case $COPT[1,\ n]$. En fait, seules les cases dont l'indice de colonne est supérieur à l'indice de ligne sont utiles (triangle supérieur droit).

$COPT[i,\ i+j]$ fait appel à des valeurs de la même ligne ($COPT[i,\ k-1]$) plus à gauche et à des valeurs de la même colonne ($COPT[k,\ i+j]$) en-dessous.

Pour démarrer, on commence par initialiser la partie non récurrente du système

d'équations, ici la diagonale "principale". L'algorithme est le suivant :

procédure *prodmatchaînée* ;
var ent i, j, k, t ;
début
  pour i **de** 1 **à** n **faire**
    pour j **de** $i+1$ **à** n **faire**
      $COPT[i,\ j] \leftarrow \infty$
    fait
  fait ;
  pour i **de** 1 **à** n **faire** $COPT[i,\ i] \leftarrow 0$ fait ;
  pour j **de** 1 **à** $n-1$ **faire**
    pour i **de** 1 **à** $n-j$ **faire**
      pour k **de** $i+1$ **à** $i+j$ **faire**
        $t \leftarrow COPT[i,\ k-1] + COPT[k,\ i+j] + D[i-1] * D[k-1] * D[i+j]$;
        si $t < COPT[i,\ i+j]$ **alors**
          $COPT[i,\ i+j] \leftarrow t$;
          $meilleur[i,\ j] \leftarrow k$;
          co on mémorise la meilleure séquence **fco**
        fsi
      fait
    fait
  fait
fin.

Remarquons qu'on a inclus une information permettant de retrouver la façon optimale de faire le calcul (routage).

Cet algorithme a une complexité temporelle en $\theta(n^3)$ et une complexité spatiale en $\theta(n^2)$.

Example 4.2. On cherche, comme précédemment, à optimiser le calcul de :

$$M_1[10,\ 20] \times M_2[20,\ 50] \times M_3[50,\ 1] \times M_4[1,\ 100].$$

- On initialise la diagonale principale à zéro ;
- Calcul de la case [1, 2] :
  $COPT[1,\ 1] + COPT[2,\ 2] + 10 * 20 * 50 = 0 + 0 + 10000 = 10000;\ k = 2$;
- Calcul de la case [2, 3] :
  $COPT[2,\ 2] + COPT[3,\ 3] + 20 * 50 * 1 = 0 + 0 + 1000 = 1000;\ k = 3$;
- Calcul de la case [3, 4] :
  $COPT[3,\ 3] + COPT[4,\ 4] + 50 * 1 * 100 = 0 + 0 + 5000 = 5000;\ k = 4$;

– Calcul de la case [1, 3] :

$$min \begin{cases} COPT[1,\ 1] + COPT[2,\ 3] + 10 * 20 * 1 = 1200, \\ COPT[1,\ 2] + COPT[3,\ 3] + 10 * 50 * 1 = 10500 \end{cases}$$
$$= 1200;\ k = 2;$$

– Calcul de la case [2, 4] :

$$min \begin{cases} COPT[2,\ 2] + COPT[3,\ 4] + 20 * 50 * 100 = 105000, \\ COPT[2,\ 3] + COPT[4,\ 4] + 20 * 1 * 100 = 3000 \end{cases}$$
$$= 3000;\ k = 4;$$

– Calcul de la case [1, 4] :

$$min \begin{cases} COPT[1,\ 1] + COPT[2,\ 4] + 10 * 20 * 100 = 23000, \\ COPT[1,\ 2] + COPT[3,\ 4] + 10 * 50 * 100 = 65000, \\ COPT[1,\ 3] + COPT[4,\ 4] + 10 * 1 * 100 = 2200 \end{cases}$$
$$= 2200;\ k = 4;$$

Grâce aux informations de “routage”, on peut reconstituer la solution optimale qui est :

$$((M_1 \times (M_2 \times M_3)) \times M_4).$$

Chapitre 5

Algorithmes gloutons

5.1 Principe

Les algorithmes gloutons ou voraces (*greedy*) ont la particularité de ne jamais remettre en cause les choix qu'ils effectuent. La succession de leurs choix ne les mène pas forcément à une bonne solution ni *a fortiori* à une solution optimale. Il existe cependant des algorithmes gloutons qui trouvent toujours la solution cherchée, qu'on appelle *exacts*.

Il faut prouver ou infirmer qu'un algorithme glouton est exact pour chaque problème traité. Dans le second cas, exhiber un contre-exemple suffit. Dans le premier, la démonstration n'est pas toujours facile. L'avantage des algorithmes gloutons est leur faible complexité. Dans certains problèmes d'optimisation, il est parfois préférable d'obtenir une solution approchée par un algorithme glouton qu'une solution optimale par un algorithme trop coûteux.

Comment démontrer qu'un algorithme glouton est exact ?

Il existe deux principales techniques pour prouver l'exactitude d'un algorithme glouton. L'une consiste à établir par récurrence que l'algorithme fait "la course en tête" du début à la fin, c'est à dire que la solution partielle en construction est constamment optimale. L'autre consiste à partir de la solution gloutonne, que l'on exprime comme une permutation particulière d'un ensemble de choix. On montre alors que la transformer en une autre solution par un "échange" de deux choix quelconques (donc la transformer en une autre permutation des choix) ne peut jamais l'améliorer.

5.2 Exemple

Remplir un carré magique d'ordre n impair. Les nombres 1 à n^2 sont utilisés une fois et une seule. La somme des lignes, des colonnes et des diagonales vaut la même valeur. Application : $n = 5$. Il existe au total 275 305 224 carrés magiques d'ordre 5, on n'en demande qu'un !

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Solution :

Enrouler le carré en tore. Partir de la case centrale de la ligne supérieure. Y mettre la valeur 1. Quand on a mis la valeur i, regarder dans la case diagonale au-dessus à gauche. Si elle est vide, y mettre $i+1$. Sinon, mettre $i+1$ dans la case en-dessous.

Chapitre 6

Conclusion

Critères pour le choix d'un algorithme :
- exactitude (optimalité) de la réponse ;
- complexités temporelle et spatiale ;
- difficulté de mise au point.

Autre méthode : PSEP (voir cours d'intelligence artificielle, algorithme A^*).