1 Problèmes d'optimisation

Là où les problèmes de décision correspondent aux questions de la forme "est-il vrai que X ?", les problèmes d'optimisations correspondent aux questions de la forme "quel est le meilleur moyen de faire X en respectant la contrainte Y ?".

A Premier exemple : le problème du sac à dos

Dans le problème du sac à dos, on dispose de n objets. Chaque objet $i \in [\![1, n]\!]$ a un poids w_i et un prix c_i. On veut mettre les objets de plus grande valeur possible dans notre sac, mais celui-ci ne peut contenir que jusqu'à un poids limite W. On veut donc choisir une collection d'objets $I \subseteq [\![1, n]\!]$ telle que :

$$\sum_{i \in I} c_i \text{ est maximale, sachant que}$$
$$\sum_{i \in I} w_i \leq W$$

Exemple 1

Déterminer une solution optimale, pour l'instance suivante du problème du sac à dos avec un poids limite de sac $W = 18$:

objet :	1	2	3	4	5	6
poids :	4	5	5	3	2	14
prix :	9	7	8	1	3	17

Plus formellement :

Définition 1

Un problème d'optimisation $\mathcal{P}$ est un ensemble d'instances. Chaque instance $\mathcal{I}$ est constituée de :

- un $\mathcal{S}$ appelé **espace des solutions admissibles**, ou espace des solutions ;
- une fonction $f : \mathcal{S} \to \mathbb{R}$ appelée **fonction objectif**.
- Un sens d'optimisation : minimiser ou maximiser

On notera une telle instance :

$$\max_{X \in \mathcal{S}} f(X) \quad \text{ou bien} \quad \min_{X \in \mathcal{S}} f(X)$$

Comme la notation l'indique, une instance d'un problème problème d'optimisation consiste à trouver la solution pour laquelle la fonction objectif atteint sa valeur maximale (ou minimale, selon le problème). On appelle **valeur optimale** de l'instance $\mathcal{I}$ la valeur maximale / minimale atteinte par f, et on la note $\mathbf{val}(\mathcal{I})$. On appelle **ensemble des solutions optimales** l'ensemble $\{X^* \in \mathcal{S} \mid f(X^*) = \mathbf{val}(\mathcal{I})\}$

En pratique, un problème d'optimisation est constitué d'instances similaires et paramétrées. On décrira donc un problème en décrivant une instance générale.

Par exemple, étant donné $w_1, \dots, w_n \in \mathbb{R}^+$, $c_1, \dots, c_n \in \mathbb{R}^+$ et $W \in \mathbb{R}^+$, le problème du sac à dos sur l'instance $((w_1, \dots, w_n), (c_1, \dots, c_n), W)$ est :

$$\textbf{KNAPSACK} : \max_{I \in S} f(I)$$

avec :

— $S = \{I \subseteq [\![1, n]\!] \mid \sum_{i \in I} w_i \leq W\}$ l'espace des solutions ;

— $f : I \mapsto \sum_{i \in I} c_i$ la fonction objectif.

Ainsi, la fonction objectif exprime la quantité que l'on cherche à **optimiser** (ici, la somme des coûts des objets), et l'espace des solutions exprime les **contraintes** à respecter (ici, le fait que la somme des poids ne doit pas dépasser la limite).

B Rendu de monnaie

Le problème de rendu de monnaie consiste à trouver la manière de rendre la monnaie sur un certain montant en utilisant **le moins de pièces** possible.

Exemple 2

On considère des pièces de 1, 2, 5, 10, 20, 50 centimes. Quelle est la manière de rendre 97 centimes qui permet d'utiliser le moins de pièces possible ?

Formellement, étant donné des entiers $a_1, \dots, a_p$ (les montants des pièces), et un entier M (le montant à décomposer), le problème d'optimisation du rendu de monnaie sur l'instance $a_1, \dots, a_p, M$ est :

$$\textbf{MONNAIE} : \min_{(x_1, \dots, x_p) \in S} (x_1 + \cdots + x_p)$$

avec $S = \{(x_1, \dots, x_n) \in \mathbb{N}^n \mid x_1 a_1 + \cdots + x_p a_p = M\}$ l'ensemble des solutions admissibles. Chaque x_i représente le nombre de fois où la pièce a_i a été utilisée.

C Festival

On considère le problème suivant : Un grand festival propose n évènements $E_1, \dots E_n$ ayant chacun un horaire de début $d_i \in \mathbb{R}$ et un horaire de fin $f_i \geq d_i$ (pour $i \in [\![1, n]\!]$. On veut assister à un maximum d'évènements, mais certains se chevauchent, on doit donc choisir auxquels assister. On considèrera que l'on peut assister à deux évènements même si le deuxième commence à l'instant où le premier finit.

Exemple 3

Voici un exemple d'instance du problème :

n° d'évènement	1	2	3	4	5	6	7
début	1	3	6	9	11	13	16
fin	5	8	14	19	12	17	19

Exercice 1

Formaliser le problème d'optimisation, en précisant, pour une instance $(d_1, f_1), \dots, (d_n, f_n)$ donnée, l'ensemble des solutions admissibles, et la fonction objectif.

D Décomposition en sous-problèmes

Considérons un problème d'optimisation $\mathcal{P}$ tel que pour toute instance $\mathcal{I}$, on peut, au choix :

— Résoudre $\mathcal{I}$ de manière triviale ;

— Transformer $\mathcal{I}$ en des sous-instances $\mathcal{I}_1, \dots \mathcal{I}_n$ telles qu'à partir de solutions $X_1, \dots X_n$ optimales pour les sous-instances, on peut reconstruire une solution optimale X de l'instance originale.

On dit que $\mathcal{P}$ possède une propriété de **sous-structure optimale**.
On peut voir le procédé cassant l'instance en sous-instances comme un **choix** à faire. Par exemple, revenons sur les problèmes vus jusqu'à maintenant :

Rendu de monnaie On considère une instance $\mathcal{I} = (M, a_1, \dots, a_p)$ du problème de rendu de monnaie, où M est le montant à décomposer et $a_1, \dots a_p$ les montants de pièces disponibles. On peut utiliser au plus $\lfloor \frac{M}{a_p} \rfloor$ fois la pièce a_p. On peut alors construire des instances $\mathcal{I}_0, \dots \mathcal{I}_{\lfloor \frac{M}{a_p} \rfloor}$ où $\mathcal{I}_k$ revient à avoir utilisé k pièces de valeur a_p :

— $\mathcal{I}_0 = (M, a_1, \dots, a_{p-1})$

— $\mathcal{I}_1 = (M - a_p, a_1, \dots, a_{p-1})$

— ...

— $\mathcal{I}_k = (M - ka_p, a_1, \dots, a_{p-1})$

— ...

Remarquons que si l'on trouve une solution optimale pour chaque instance $\mathcal{I}_k$, on peut immédiatement en déduire une solution optimale pour l'instance initiale $\mathcal{I}$. En effet, une solution à $\mathcal{I}_k$ utilisant m pièces donne une solution à $\mathcal{I}$ utilisant $m + k$ pièces. De plus, puisque l'on couvre toutes les possibilités pour $\mathcal{I}$, on trouve forcément une solution optimale.

Ce procédé de décomposition nous donne un algorithme dit **force-brute**, qui va consister à lister et tester toutes les possibilités, récursivement :

Algorithme 1 : `ForceBrute`$(M, a_1, \dots, a_p)$

Entrée(s) : $M \in \mathbb{N}$ montant à décomposer, $a_1, \dots, a_p$ pièces utilisables
Sortie(s) : $x_1, \dots, x_p$ solution optimale au problème de rendu de monnaie

```
si M = 0 alors
    retourner (0, ..., 0)
si p = 0 (i.e. il n'y a pas de pièces) alors
    retourner IMPOSSIBLE
X ← NULL // meilleure solution actuelle
s ← +∞ // valeur de la solution actuelle
pour k = 0 à ⌊M/a_p⌋ faire
    y_1, ... y_{p-1} ← ForceBrute(M − k a_p, a_1, ... a_{p-1});
    si y_1 + ··· + y_{p-1} + k < s alors
        X ← (y_1, ..., y_{p-1}, k);
        s ← y_1 + ··· + y_{p-1} + k;
retourner X
```

Notons que cet algorithme est bien trop coûteux pour être utilisable en pratique, mais il fait bien apparaître **l'arbre d'exploration** généré par l'instance initiale.

Festival Pour le problème du festival, on peut subdiviser une instance en choisissant un évènement auquel on se rend, et en éliminant les évènements entrant en conflit. Étant donné $\mathcal{I} = \{(d_1, f_1), \ldots, (d_n, f_n)\}$, on peut considérer les sous-instances $\mathcal{I}_1, \ldots, \mathcal{I}_n$ comme suit :

$$\forall k \in [\![1, n]\!], \mathcal{I}_k = \{(d_i, f_i) \mid i \in [\![1, n]\!], (d_i, f_i) \cap (d_k, f_k) = \emptyset\}$$

Autrement dit, résoudre l'instance $\mathcal{I}_k$ consiste à trouver le moyen d'assister au plus d'évènements possible parmi ceux disjoints de l'évènement k. A nouveau, on peut trouver une solution à $\mathcal{I}$ à partir de solutions aux $\mathcal{I}_k$, en considérant l'instance $\mathcal{I}_k$ qui permet d'assister au plus d'évènements. On en déduit un algorithme en force brute :

Algorithme 2 : `ForceBrute`$((d_1, f_1), \ldots, (d_n, f_n))$

Entrée(s) : $(d_1, f_1), \ldots, (d_n, f_n)$ intervalles, avec $d_i < f_i$
Sortie(s) : $I \subseteq [\![1, n]\!]$ ensemble de cardinal maximal d'indices d'intervalles 2 à 2 disjoints

```
si n = 0 alors
    retourner ∅
I ← ∅ // meilleure solution actuelle
pour k = 1 à n faire
    (i_1, ..., i_p) ← indices des intervalles disjoints de (d_k, f_k);
    I' ← ForceBrute((d_{i_1}, f_{i_1}), ..., (d_{i_p}, f_{i_p}));
    si |I'| + 1 > |I| alors
        I ← I' ∪ {k}
retourner X
```

A nouveau, cet algorithme est bien trop coûteux pour être utilisable en pratique. De plus, contrairement à l'algorithme en force brute pour le rendu de monnaie, qui énumérait de manière unique toutes les solutions, ici rien n'empêche l'algorithme de choisir les évènements 1 et 2 dans cet ordre, et ensuite de choisir les évènements 2 et 1.

Exercice 2

Proposer un algorithme force brute pour le problème du sac à dos.

2 Algorithmes gloutons

On considère un problème d'optimisation, dont on suppose qu'il peut se modéliser comme une série de choix successifs (sac à dos, programmation d'évènements, rendu de monnaie...). Un algorithme est dit glouton si il fait des choix **localement optimaux** et ne **revient jamais en arrière**.
Autrement dit, un algorithme glouton va consister à ne chercher à résoudre **qu'une seule** sous-instance du problème, choisie intelligemment, sans en tester une deuxième ensuite.
En comparaison avec les algorithmes force-brute qui explorent l'intégralité de l'arbre des sous-instances, les algorithmes gloutons vont généralement explorer **une seule branche**, et donc potentiellement renvoyer une solution sous-optimale. Ce sont des algorithmes généralement assez simple à concevoir et à implémenter, avec des complexités faibles, et, même s'ils donnent rarement une solution optimale, il permettent au moins de trouver une solution, qui peut être éventuellement raffinée ultérieurement.

A Rendu de monnaie

Sur le problème de rendu de monnaie, l'algorithme que l'on emploie dans la vie de tous les jours, qui est le plus naturel, est un algorithme glouton : il consiste à toujours utiliser la pièce de plus grande valeur possible. S'il reste un montant M à rendre, et que la plus grande pièce utilisable est a_i, on va donc utiliser autant de pièces de valeur a_i que possible : $\lfloor \frac{M}{a_i} \rfloor$.

On considèrera dans la suite que les pièces du système de monnaie sont données dans l'ordre croissant : $a_1 < a_2 < \dots$.

Algorithme 3 : Rendu de monnaie glouton

Entrée(s) : $a_1, \dots, a_p \in \mathbb{N}^*$ les valeurs d'un système de monnaie, $M \in \mathbb{N}$ montant à décomposer

Sortie(s) : Nombre minimal de pièces/billets à utiliser pour rendre la monnaie sur M

```
x_1, ..., x_p ← 0, ..., 0;
pour i = p à 1 faire
    x_i ← ⌊M / a_i⌋;
    M ← M − x_i a_i;
si M > 0 alors
    retourner Pas de solution
sinon
    retourner x_1 + ··· + x_p
```

Exercice 3

Q1. Appliquez cet algorithme sur le même système qu'au dessus, avec $M = 144$

Q2. Trouver une instance (système de pièces + montant) pour lequel l'algorithme glouton renvoie une solution non-optimale, ou même ne trouve pas du tout de solution alors qu'il en existe une.

Cet algorithme s'effectue en $O(p)$, ou$\mathcal{O}(p \log p)$ s'il faut trier les pièces au départ. Il est donc polynomial en la taille de l'entrée, mais nous venons de voir qu'il n'est pas optimal dans tous les systèmes de pièce.

On dit qu'un système de pièces est **canonique** si l'algorithme glouton est optimal pour ce système.

Exercice 4

Soit $B \in \mathbb{N} \setminus \{0,1\}$ et $k \in \mathbb{N}^*$. On considère le système de pièces $S_k = (1, B, B^2, \dots, B^{k-1})$.

Q1. Que renvoie l'algorithme glouton lors de la décomposition d'un entier M dans S_k ?

Q2. Montrer que le système S_k est canonique. On pourra commencer par montrer que dans une solution optimale, aucune pièce n'est utilisée B fois ou plus, à part la plus grande.

Montrons de même que le système $1, 2, 5, 10$ est canonique. On considère M un entier à décomposer. On s'intéresse donc à l'instance $(1, 2, 5, 10), M$ du problème de rendu de monnaie. Notons que l'ensemble des solutions admissibles n'est pas vide car $M = M \times 1$ donc $(M, 0, 0, 0)$ est une solution admissible. De plus, l'ensemble des solutions admissibles est fini, il existe donc une solution optimale.

Soit $x_1^*, x_2^*, x_5^*, x_{10}^*$ une solution optimale et soit x_1, x_2, x_5, x_{10} la solution renvoyée par l'algorithme glouton.

Q3. Montrer que $x_1^* \leq 1$ en montrant que si $x_1^* \geq 2$, alors on peut construire une solution $x_1', x_2', x_5', x_{10}'$ strictement meilleure que $x_1^*, x_2^*, x_5^*, x_{10}^*$.

Q4. De même, donner des bornes supérieures sur x_2^* et x_5^*

Q5. Montrer que $x_1^* + 2x_2^* + 5x_5^* < 10$.

Q6. Montrer que $x_{10} = x_{10}^*$, puis que $x_5 = x_5^*$, $x_2 = x_2^*$ et $x_1 = x_1^*$. Conclure.

B Sac à dos

Tentons de résoudre le problème du sac à dos avec un algorithme glouton. On choisit donc les objets un par un, en décidant de manière gloutonne. De manière générale, les algorithmes que l'on va tester auront tous la même structure :

Algorithme 4 : Sac à dos glouton

Entrée(s) : n objets de poids $w_1, \dots, w_n$ et de prix $c_1, \dots, c_n$, W taille limite du sac

```
1 Trier les objets selon un certain critère;
2 w ← W // taille actuelle restante dans le sac
3 c ← 0 // prix actuel du sac
4 pour i = 1 à n faire
5     si w_i ≤ t alors
6         w ← w − w_i;
7         c ← c + c_i;
8 retourner c
```

Première idée : On choisit en priorité les objets les plus chers : en effet, un objet cher est a priori avantageux. On trie donc les objets par **prix décroissant**, afin d'avoir $c_1 \geq c_2 \geq \cdots \geq c_n$.

Exercice 5

Appliquer cet algorithme sur l'instance suivante (taille de sac max $W = 18$) :

poids :	4	5	5	3	2	14
prix :	9	7	8	1	3	17

On remarque donc que cet algorithme ne donne pas nécessairement une solution optimale. En revanche, il construit bien une solution valide, car les objets utilisés ont bien, au total, un poids inférieur à la borne W imposée. La complexité est en $\mathcal{O}(n \log n)$, car il faut trier les objets par prix décroissant au départ.

Deuxième idée : Choisir les objets par **poids croissant**, afin de prioriser les objets prenant peu de place.

Exercice 6

Appliquer cet algorithme sur l'instance de l'exemple précédent. La solution obtenue est-elle optimale ?

Troisième idée : Choisir les objets par rapport prix/poids décroissant. Autrement dit, on choisit d'abord l'objet le plus rentable au kilo.

Exercice 7

Q1. Appliquer cet algorithme sur l'exemple précédent.

Q2. Exhiber un contre-exemple montrant que cet algorithme n'est PAS optimal.

On peut néanmoins se demander si les solutions générées sont proches de la solution optimale dans une certaine mesure. Pour l'algorithme précédent, on peut trouver des instances pour lesquelles il est arbitrairement mauvais :

Q3. Soit $k \in \mathbb{R}^{+*}$. Donner une instance du problème du sac à dos telle que, en notant C^* la valeur optimale de l'instance et C la valeur trouvée par l'algorithme glouton, on a $C \leq \frac{C^*}{k}$

C Spectacles

On considère le schéma suivant d'algorithme glouton :

Algorithme 5 : Spectacles

Entrée(s) : $(d_1, f_1), \ldots (d_n, f_n) \in \mathbb{R}^2$ avec $d_i < f_i$, n horaires d'évènements
Sortie(s) : $I \subseteq [\![1, n]\!]$ ensemble d'évènements deux à deux disjoints

```
I ← ∅;
Trier les évènements selon un certain critère;
pour i = 1 à n faire
    si (d_i, f_i) n'intersecte aucun évènement de I alors
        I ← I ∪ {i};
retourner I
```

A nouveau, cet algorithme est glouton car il ne réfléchit pas aux évènements qu'il va rencontrer plus tard, et ne revient pas sur ses choix. On propose trois critères :

— Choisir les évènements par longueur croissante (i.e. privilégier les évènements courts)

— Choisir les évènements par horaire de début croissant

— Choisir les évènements par horaire de fin croissant

Exercice 8

Q1. Montrez qu'aucun des deux premiers critères ne donne un algorithme optimal, en exhibant des contre-exemples.

Cependant, nous allons voir que le troisième critère, lui, donne lieu à un algorithme optimal. On considère à partir de maintenant l'algorithme glouton où les évènements sont choisis par horaire de fin croissant.

Q2. Appliquer cet algorithme sur les contre-exemples trouvés à l'exercice précédent et vérifier qu'on obtient bien des solutions optimales.

Montrons maintenant que l'algorithme renvoie bien une solution optimale. Nous allons procéder par un **argument d'échange**, qui va consister à considérer une solution optimale quelconque et la solution renvoyée par l'algorithme glouton, et à échanger certaines évènements afin de transformer la solution optimale en la solution gloutonne. En particulier, on montrera ainsi que les deux solutions ont la même valeur (i.e. le même nombre d'intervalles). On considère donc $(d_1, f_1), \dots, (d_n, f_n)$ une instance du problème de programmation d'évènements. On suppose que les évènements sont triés par horaire de fin croissante, i.e. $f_1 \leq f_2 \leq \cdots \leq f_n$. On note $I^* = \{i_1, \dots, i_k\} \subseteq [\![1, n]\!]$ une solution optimale, et $I = \{j_1, \dots, j_l\} \subseteq [\![1, n]\!]$ la solution renvoyée par l'algorithme glouton. Remarquons que $j_1 = 1$ car le premier intervalle est toujours pris par l'algorithme glouton.

Q3. Montrez qu'il existe une solution I' optimale qui contient le premier évènement (d_1, f_1). *Indication : montrez que l'on peut remplacer le premier évènement de I^* par (d_1, f_1).*

Q4. En suivant le même raisonnement, montrez par récurrence sur $t \in [\![0, n]\!]$ que $I_t^* = \{j_1, \dots, j_t, i_{t+1}, i_k\}$ est une solution admissible, et qu'elle est optimale.

Q5. En déduire que la solution renvoyée par l'algorithme glouton est optimale.

Exercice 9

On propose une autre stratégie gloutonne : toujours choisir l'évènement ayant le moins d'intersections possibles. Notons que l'on ne peut pas exprimer cet algorithme comme un simple choix de critère en utilisant le schéma précédent, car le nombre d'intersections change au fil de l'algorithme.

Q1. Écrire le pseudo-code de l'algorithme proposé, et évaluer sa complexité.

Q2. Est-il optimal ?

3 Diviser pour régner

Dans les problèmes étudiés pour l'instant, les sous-instances que l'on construit sont généralement de tailles proches de l'instance de base : on passe d'une instance de taille n à plusieurs instances de taille $n-1$. De plus, les sous-instances générées se chevauchent potentiellement. La stratégie "diviser pour régner" (DPR) s'applique lorsque l'on peut séparer une instance $\mathcal{I}$ d'un problème en plusieurs petites instances $\mathcal{I}_1, \ldots, \mathcal{I}_p$ **disjointes** et **de tailles comparables**. En pratique, cela peut revenir par exemple à diviser une instance de taille n en deux instances de tailles $\frac{n}{2}$.

Nous avons déjà vu trois DPR : le tri fusion, le tri rapide, et la recherche par dichotomie. Dans les deux premiers cas (en considérant un bon choix de pivot pour le tri rapide), pour trier un tableau de n case, on se ramène à devoir trier deux tableaux d'au plus $\frac{n}{2}$ cases. On rappelle que le calcul de la complexité fait alors intervenir l'équation $C(n) = 2C(\frac{n}{2}) + \Theta(n)$, qui se résout en $C(n) = \mathcal{O}(n \log n)$.

La mise en place d'algorithmes DPR va souvent nécessiter d'établir et de résoudre des équations de récurrence de ce style.

A Multiplication de polynômes

On représentera les polynômes par des tableaux donnant leurs coefficients. Par exemple, $P = 3X^2 - 8X + 5$ sera représenté par le tableau $[5, -8, 3]$. Notons qu'un polynôme de degré $d \geq 0$ est représenté par un tableau de $d+1$ cases : on dira "polynôme de taille n" pour parler d'un polynôme ayant n coefficients, et donc de degré au plus $n-1$.

Calculer la somme d'un polynôme P à n coefficients et d'un polynôme Q à m coefficients prend un temps linéaire $\mathcal{O}(n+m)$. De même, ajouter un terme aX^i à un polynôme de taille au moins $i+1$ se fait en $\mathcal{O}(1)$: il suffit d'ajouter a à la case d'indice i du tableau des coefficients.

On considère $P(X) = \sum_{i=0}^{n-1} a_i X^i$ et $Q = \sum_{i=0}^{n-1} b_i X^i$ deux polynômes de degrés strictement inférieurs à n, et l'on souhaite calculer le produit :

$$PQ(X) = \sum_{i=0}^{2n-2} (\sum_{j=0}^{i} a_j b_{i-j}) X^i$$

Quitte à ajouter des coefficients nuls, on suppose que n est une puissance de 2 (ce choix permettra de simplifier les explications). L'algorithme naïf consiste à calculer un par un chacun des coefficients selon la formule précédente :

Algorithme 6 : `produitNaif`(P, Q)

Entrée(s) : $P(X) = \sum_{i=0}^{n-1} a_i X^i$ et $Q = \sum_{i=0}^{n-1} b_i X^i$ deux polynômes de taille n
Sortie(s) : $PQ(X)$

```
1 R ← 0;
2 pour i = 0 à 2n − 2 faire
3     pour j = 0 à i − 1 faire
4         R ← R + a_j b_{i−j} X^i;
5 retourner R
```

Si l'on utilise une représentation des polynômes par tableau de coefficients, alors l'opération ligne 4 se fait en temps constant $\mathcal{O}(1)$. Donc, l'algorithme est au total en $\mathcal{O}(n^2)$.

Tentons d'appliquer une stratégie DPR sur ce problème. Supposons n pair, et décomposons P et Q en les coupant à la moitié, comme suit :

$$P = P_0 + X^{\frac{n}{2}} P_1 \qquad\qquad Q = Q_0 + X^{\frac{n}{2}} Q_1$$

avec $\mathbf{deg}P_0, \mathbf{deg}P_1, \mathbf{deg}Q_0, \mathbf{deg}Q_1 < \frac{n}{2}$. Par exemple, si $P = 2 + X - 3X^2 + 7X^3$, alors $P_0 = 2 + X$ et $P_1 = 3 + 7X$.

Alors, on a :

$$PQ(X) = P_0Q_0 + (P_0Q_1 + P_1Q_0)X^{\frac{n}{2}} + P_1Q_1X^n$$

Cette égalité montre que pour effectuer la multiplication de deux polynômes de degré n, P et Q, alors il suffit d'effectuer 4 multiplications de polynômes de degré $\frac{n}{2}$. On peut donc proposer l'algorithme suivant :

Algorithme 7 : `MultNaive`(P, Q)

Entrée(s) : P, Q deux polynômes de degré $< n$
Sortie(s) : PQ polynôme produit de P et Q

```
si n = 1 alors
    Calculer R = PQ par calcul direct ;
    retourner R
Décomposer P = P_0 + X^{n/2} P_1;
Décomposer Q = Q_0 + X^{n/2} Q_1;
R_0 ← MultNaive(P_0, Q_0);
R_1 ← MultNaive(P_0, Q_1);
R_2 ← MultNaive(P_1, Q_0);
R_3 ← MultNaive(P_1, Q_1);
retourner R_0 + (R_1 + R_2)X^{n/2} + R_3X^n
```

Sa complexité vérifie l'équation $C(n) = 4C(\frac{n}{2}) + \Theta(n)$.
Pour simplifier le calcul, disons $C(n) = 4C(\frac{n}{2}) + n$.
Déroulons la récurrence et trouver une expression asymptotique de $C(n)$:

Finalement, cet algorithme DPR n'est pas meilleur que l'algorithme naïf ! En pratique il serait même pire car la constante de complexité est supérieure. Si l'on voulait que la méthode DPR soit plus efficace, il faudrait se débrouiller pour résoudre strictement moins que 4 sous-instances de taille $\frac{n}{2}$.

Posons $A = P_0Q_0$, $B = P_1Q_1$ et $C = (P_0 + P_1)(Q_0 + Q_1)$ alors on a :

$$\begin{array}{lcl} P_0Q_0 & = & A \\ P_1Q_1 & = & B \\ P_0Q_1 + P_1Q_0 & = & C - A - B \end{array}$$

Autrement dit, on retrouve les trois termes qui apparaissent dans l'algorithme de multiplication précédent $(P_0Q_0 + (P_0Q_1 + P_1Q_0)X^{\frac{n}{2}} + P_1Q_1X^n)$ mais on n'a eu besoin d'effectuer que 3 multiplications sur des polynômes de taille moitié. Cette remarque nous donne l'algorithme de Karatsuba [1]

Algorithme 8 : `karatsuba`(P, Q)

Entrée(s) : P, Q deux polynômes de degré n
Sortie(s) : PQ

si $n = 0$ **alors**
  Calculer $R = PQ$ par calcul direct ;
  retourner R
Décomposer $P = P_0 + X^{\frac{n}{2}}P_1$;
Décomposer $Q = Q_0 + X^{\frac{n}{2}}Q_1$;
$A \leftarrow$ `karatsuba`(P_0, Q_0);
$B \leftarrow$ `karatsuba`(P_1, Q_1);
$C \leftarrow$ `karatsuba`$(P_0 + P_1, Q_0 + Q_1)$;
retourner $A + (C - B - A)X^{\frac{n}{2}} + BX^n$

La complexité $C(n)$ de l'algorithme vérifie alors :

$$C(n) = 3C(\frac{n}{2}) + \Theta(n)$$

Exercice 10

Dérouler cette relation et déterminer la complexité asymptotique de l'algorithme de Karatsuba.

Remarque 1

Il existe un algorithme DPR plus efficace en $\mathcal{O}(n \log n)$, utilisant la transformée de Fourier rapide.

1. Du nom d'Anatoly Karatsuba, le chercheur ayant découvert cette méthode en 1960.

B Actions

Exercice 11

On considère un tableau T d'entiers $a_1, \dots a_n$, avec a_i qui représente le prix d'une action à la date i. On se demande quel est le profit maximal que l'on peut obtenir en achetant puis en vendant une action.

Q1. Formaliser le problème d'optimisation.

Q2. Proposer un algorithme naïf en $\mathcal{O}(n^2)$.

Q3. En utilisant la méthode diviser pour régner, proposer un algorithme plus efficace en $\mathcal{O}(n \log n)$.

Q4. Améliorer l'algorithme précédent en $\mathcal{O}(n)$. (*Indication : calculez toutes les grandeurs dont vous avez besoin en même temps, avec une seule fonction*).

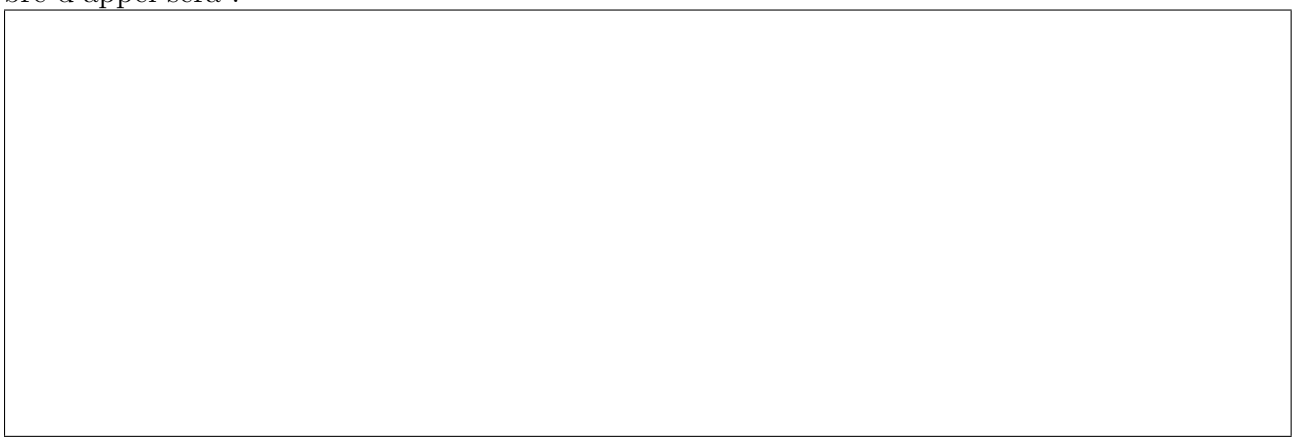
4 Programmation dynamique

Dans la section précédente, les problèmes que nous avons résolu par la stratégie DPR avaient un point commun : les sous-instances construites à partir d'une instance donnée étaient totalement disjointes. Par exemple, pour le tri fusion, on divise le tableau à trier en deux parties, que l'on peut traiter indépendamment. La programmation dynamique, au contraire, est utile lorsque les sous-instances sont très similaires. Cette méthode consiste à stocker en mémoire les solutions de toutes les sous-instances que l'on traite, ce qui permet d'éviter beaucoup de calculs redondants.

Définition 2

On dit qu'un problème a des sous-problèmes se chevauchant si un algorithme naïf récursif pour le résoudre doit résoudre plusieurs fois les mêmes sous-instances.

Un exemple simple et expliquant bien le principe est le calcul de la suite de Fibonacci. On pose $u_0 = 0, u_1 = 1$ et $u_{n+2} = u_{n+1} + u_n$ pour $n \in \mathbb{N}$. Cette formule de récurrence donne immédiatement un algorithme récursif, mais celui ci est très lent. En effet, pour calculer u_5, l'arbre d'appel sera :



La complexité de cet algorithme est exponentielle : on ne pourrait même pas calculer u_{50} avant la fin de l'année. On peut voir que le terme u_3 est calculé indépendamment par l'algorithme deux fois. Le principe de programmation dynamique est de stocker chaque résultat la première fois qu'il est calculé, dans un tableau. Alors, l'arbre d'appel sera comme suit :

Avec cette amélioration, l'algorithme devient **linéaire** en temps ! Notons que cette amélioration en temps a un coût en terme d'espace, puisque l'on doit utiliser un tableau de n cases pour calculer u_n [2].

2. Il existe de bien meilleurs algorithmes, plus rapides et utilisant moins d'espace mémoire. Par exemple, en utilisant l'exponentiation rapide pour calculer les puissances de la matrice associée à la relation linéaire de u, on trouve un algorithme assez simple en temps logarithmique.

A Principe général

Pour mettre en place un algorithme de programmation dynamique :

1. On identifie une formule de récurrence reliant les valeurs optimales des différentes instances / sous-instances du problème ;
2. On introduit un tableau (éventuellement multidimensionnel) pour stocker ces valeurs. Une des cases correspond à l'instance initial à résoudre ;
3. On remplit le tableau à l'aide de la formule de récurrence, jusqu'à avoir rempli la case objectif.

On s'intéresse à deux méthodes permettant de remplir ce tableau :

— De **bas en haut**, en remplissant les cases itérativement avec des boucles, en partant des instances triviales ;

— De **haut en bas**, en partant de la case objectif, et en calculant récursivement les cases dont on a besoin.

B Construction de bas en haut

On reprend le problème du rendu de pièce. On se donne $a_1, \dots, a_n \in \mathbb{N}^*$ des pièces et $M \in \mathbb{N}$ un montant à décomposer. Pour $x \in \mathbb{N}$ et $i \in [\![0, n]\!]$, on note $C(x, i)$ le nombre minimal de pièces de valeurs $a_1, a_2, \dots, a_i$ que l'on peut utiliser pour décomposer x. On remarque qu'alors, on a :

$$\begin{array}{lcll} C(0,i) & = & 0 & \text{pour } 0 \leq i \leq n \\ C(x,0) & = & +\infty & \text{pour } x > 0 \\ C(x,i+1) & = & \min(C(x,i), C + (x - a_{i+1}, i+1)) & \text{si } x \geq a_{i+1}, \text{ pour } i \geq 0 \\ C(x,i+1) & = & C(x,i) & \text{sinon, pour } i \geq 0 \end{array}$$

En effet, pour décomposer x en utilisant $a_1, \dots, a_{i+1}$, alors soit on n'utilise pas a_{i+1}, ce qui veut dire qu'on n'utilise que $a_1, \dots, a_i$, soit on utilise une fois a_{i+1}, auquel cas il nous reste $M - a_{i+1}$ à décomposer en utilisant $a_1, \dots, a_{i+1}$. Le min exprime donc que l'on calcule les deux possibilités, et que l'on choisit la meilleure des deux.

Nous avons trouvé donc trouvé une formule de récurrence que l'on pourrait utiliser pour résoudre ce problème facilement par un algorithme récursif : pour calculer la valeur optimale de l'instance $a_1, \dots, a_n, M$ du problème de rendu de monnaie, on calcule $C(M, n)$ récursivement. Comme pour le calcul de la suite de Fibonacci, cet algorithme est extrêmement inefficace, car il va demander de calculer de très nombreuses fois les mêmes valeurs de $C(x, i)$.

Appliquons le principe de programmation dynamique. On stocke $C(x, i)$ pour $x \in [\![0, M]\!], i \in [\![0, n]\!]$, dans un tableau T de taille $(M+1) \times (n+1)$. Chaque case $T[x][i]$ devra contenir $C(x, i)$, et notre objectif est donc de remplir la case $T[M][n]$.
Les cas de base de notre formule de récurrence disent quelles casent sont triviales à remplir : $T[0][i]$ vaut 0 pour $i \in [\![0, n]\!]$, car décomposer 0 demande 0 pièces, et $T[x][0]$ vaut $+\infty$ pour $x > 0$, car il est impossible de décomposer x sans utiliser de pièces.
Remarquons ensuite que pour remplir une case $T[x][i+1]$, il suffit de connaître la valeur des cases $T[x - a_{i+1}][i+1]$ et $T[x][i]$. Autrement dit, pour remplir une case, il suffit d'avoir déjà rempli les cases à sa gauche et au dessus.

On peut donc remplir le tableau T ligne par ligne, de gauche à droite :

Algorithme 9 : `progDyn`$(a_1, \dots, a_n, M)$

Entrée(s) : $a_1, \dots a_n \in \mathbb{N}^*$ des pièces, $M \in \mathbb{N}$ un montant
Sortie(s) : Nombre minimal de pièces à utiliser pour décomposer M

```
1  T ← tableau de taille (M + 1) × (n + 1);
2  pour x = 1 à M faire
3      T[x][0] = +∞;
4  pour i = 0 à n faire
5      T[0][i] = 0;
6  pour i = 1 à n faire
7      pour x = 1 à M faire
8          si x ≥ a_i alors
9              T[x][i] ← min(T[x][i − 1], 1 + T[x − a_i][i]);
10         sinon
11             T[x][i] ← T[x][i − 1];
12 retourner T[M][n]
```

La complexité de cet algorithme est en $\mathcal{O}(nM)$, car il faut un temps constant pour remplir chaque case. Sur cahier de prépa : une implémentation de cet algorithme en C.

Exercice 12

Pour $a_1 = 2, a_2 = 3, a_3 = 7$ et $M = 13$, appliquer l'algorithme de programmation dynamique.

De manière générale, lorsque l'on conçoit un algorithme en programmation dynamique de bas en haut, les étapes sont :

1. Décomposer le problème en sous-instances ;
2. identifier les instances triviales ;
3. identifier une formule de récurrence sur les solutions des instances ;
4. en déduire dans quel ordre remplir le tableau (faire un schéma si besoin).

On parle d'approche de bas en haut car on remplit le tableau en partant des instances triviales, et en remplissant petit à petit les instances plus complexes. Notons que l'on calcule forcément l'intégralité du tableau, même si ce n'est pas nécessaire pour calculer la case objectif. Notons aussi qu'il n'y a pas forcément un unique ordre de remplissage valide. Pour le rendu de monnaie, on aurait aussi pu remplir le tableau colonne par colonne, ou par anti-diagonale. L'important est que les **dépendances** doivent être respectées par l'ordre choisi.

C Construction de haut en bas

L'approche **de haut en bas** (ou **top-down**) est récursive, et colle plus fidèlement à la structure de l'algorithme naïf. On considère un problème d'optimisation P, et une instance $I_0 \in P$ que l'on veut résoudre par programmation dynamique.

On utilise un tableau T global pour stocker les solutions des différentes sous-instances. Pour I une sous-instance de I_0, on notera $T[I]$ la valeur stockée dans T à la case correspondant aux paramètres de l'instance I. On suppose que l'on a initialisé T en remplissant les cases des instances triviales, et en écrivant **NULL** dans les autres. Alors, le schéma général de prog. dyn. de haut en bas sera :

Algorithme 10 : `top_down`(I)

Entrée(s) : I une sous-instance du problème
Entrée(s) : S Solution optimale de l'instance I

```
si T[I] = NULL alors
    Décomposer I en sous-instances I_1, ... I_p;
    S_1 ← top_down(I_1);
    ...;
    S_p ← top_down(I_p) ; Combiner S_1, ..., S_p en une solution S de I;
    T[I] ← S;
retourner T[I]
```

Autrement dit, on a modifié l'algorithme récursif naïf pour le rendre plus intelligent : il garde en mémoire les valeurs déjà calculées dans T, et, lorsqu'il rencontre une instance qu'il a déjà résolu, renvoie immédiatement la solution. Pour répondre à l'instance I_0 initiale, il suffit donc de calculer `top_down`(I_0).

Cette méthode consistant à garder en mémoire les valeurs calculées par une fonction afin de les réutiliser est appelée la **mémoïsation**, ou la mise en cache. Appliquons cette méthode sur le problème du sac à dos. On considère un sac de contenance W, et des objets $O_1, \dots O_n$, avec chaque O_i ayant un poids w_i et une valeur c_i.

Sous-instances Pour $p \in [\![0, W]\!]$ et $j \in [\![0, n]\!]$, on note $C(p, j)$ la valeur maximale que l'on peut atteindre en remplissant un sac de contenance p, en utilisant seulement les j premiers objets. La valeur qui nous intéresse à la fin est $C(W, n)$.

Cas triviaux

— Pour $p = 0$, on ne peut rien mettre dans le sac : $C(0, j) = 0$ pour tout $j \in [\![0, n]\!]$.

— Pour $j = 0$, il n'y a rien à mettre dans le sac : $C(p, 0) = 0$ pour tout $p \in [\![0, W]\!]$.

Cas récursifs Soient $p \in [\![1, W]\!]$ et $j \in [\![1, n]\!]$. On s'intéresse au calcul de $C(p, j)$. On peut soit prendre l'objet O_j soit ne pas le prendre. Dans le premier cas, il reste une contenance $p - w_j$ à remplir avec les $j - 1$ premiers objets, et on a gagné une valeur c_j. Dans le deuxième cas, on n'a rien gagné, et il reste à remplir p avec les $j - 1$ premiers objets.
Si $p < w_j$, on ne peut pas prendre l'objet O_j. On en déduit la formule suivante :

$$C(p, j) = \begin{cases} C(p, j-1) & \text{si } p < w_j \\ \min(C(p, j-1), c_j + C(p - w_j, j-1)) & \text{sinon} \end{cases}$$

Voyons comment transformer cette formule en algorithme de prog. dyn. de haut en bas. Pour cela, regardons le tableau de prog. dyn. sur un exemple, et tentons de le remplir récursivement. On prend $W = 6$, et 3 objets, de poids respectifs $3, 2, 2$ et de valeurs $3, 4, 5$:

Implémentons cette idée en OCaml. On aura une fonction principale prenant en entrée les données du problème et initialisant le tableau de programmation dynamique, et une fonction auxiliaire récursive servant à remplir le tableau.

```
let sac_a_dos (poids: int array) (valeurs: int array) (w: int) : int =
  let n = Array.length poids in
  (* Création du tableau. La valeur -1 indique une case non-remplie *)
  let t = Array.make_matrix (w+1) (n+1) (-1) in
  Remplir les conditions initiales;
  let rec calc_t (p: int) (j: int) : int =
    if t.(p).(j) = -1 then begin
      Remplir la case p j en calculant récursivement sa valeur;
    end;
    t.(p).(j)
  in
  calc_t w n (* calcul de la case qui nous intéresse *)
```

Pour remplir les conditions initiales :

```
  for p = 0 to contenance do
    t.(p).(0) <- 0
  done;
  for j = 0 to n do
    t.(0).(j) <- 0
  done;
```

Enfin, lorsque l'on rencontre une case (p, j) pas encore calculée, on applique la formule de récurrence pour la remplir :

```
      if p < poids.(j-1) then
        t.(p).(j) <- calc_t p (j-1) (* on n'utilise pas l'objet j-1 *)
      else
        let sol_avec = valeurs.(j-1) + calc_t (p-poids.(j-1)) (j-1) in
        let sol_sans = calc_t (p) (j-1) in
        t.(p).(j) <- max sol_avec sol_sans
```

La complexité de cet algorithme est en $\mathcal{O}(nW)$. En effet, le contenu de chaque case n'est calculé qu'une fois, et demande un temps $\mathcal{O}(1)$.

Exercice 13

Appliquer cet algorithme avec des objets de poids $(3, 1, 4)$ et de valeurs $(5, 2, 6)$, pour un sac de contenance 6.

Exercice 14

Le problème du sac à dos est NP-complet, ce qui signifie entre autres que l'**on ne connaît pas d'algorithme polynomial** pour le résoudre. Cette déclaration semble contredire directement l'algorithme trouvé juste au dessus. Comment expliquer cette contradiction apparente ?

D Distance de Levenshtein

On s'intéresse à un problème classique d'algorithmique du texte. On considère un alphabet Σ fini.

Pour $u \in \Sigma^*$, on considère trois types d'opérations :

— Supprimer une lettre de u ;

— Insérer une lettre entre deux lettres de u, ou au début, ou à la fin ;

— Remplacer une lettre de u par une autre lettre de l'alphabet Σ.

La distance de Levenshtein de deux mots $u, v \in \Sigma^*$, notée $d_L(u, v)$, est le plus petit nombre d'opérations à appliquer sur u pour obtenir v. Cette distance est un type de **distance d'édition**, elle sert à quantifier la similarité entre deux mots, deux textes, etc...

Exemple 4

On considère $u =$ `ALGORITHME` et $v =$ `BAGORYMSE`. Donner $d_L(u, v)$ en exhibant une suite minimale d'opérations permettant de transformer u en v.

Remarque 2

La distance de Levenshtein est bien une distance au sens mathématique :

1. $d_L(u, v) \geq 0$ pour tout $u, v \in \Sigma^*$.
2. $d_L(u, u) = 0$ pour tout $u \in \Sigma^*$.
3. $d_L(u, v) = d_L(v, u)$ pour tout $u, v \in \Sigma^*$.
4. $d_L(u, w) \leq d_L(u, v) + d_L(v, w)$ pour tout $u, v, w \in \Sigma^*$.

Ce problème se prête bien à la programmation dynamique, car on peut exhiber une structure récursive comme suit.

Proposition 1

Pour $u, v \in \Sigma^*$ notons $u = u_1 u_2 \dots u_n$ et $v = v_1 v_2 \dots v_m$ avec $n = |u|$ et $m = |v|$. Alors :

- Si $n = 0$, alors $d_L(u, v) = m$: la solution optimale est d'insérer chaque lettre de v dans u.
- Si $m = 0$, alors $d_L(u, v) = n$: la solution optimale est de supprimer chaque lettre de u.
- Sinon :
  — Si $u_n = v_m$, alors $d_L(u, v) = d_L(u_1 \dots u_{n-1}, v_1 \dots v_{m-1})$
  — Sinon, alors $d_L(u, v) = 1 + \min \begin{cases} d_L(u_1 \dots u_{n-1}, v) \\ d_L(u, v_1 \dots v_{m-1}) \\ d_L(u_1 \dots u_{n-1}, v_1 \dots v_{m-1}) \end{cases}$

Les trois cas du min reflètent respectivement l'effet de la suppression, de l'insertion et du remplacement : on teste chaque opération pour faire correspondre les bouts de u et v, et on voit laquelle donne lieu à la meilleure solution.

Ainsi, pour calculer la distance de Levenshtein entre u et v, on dresse un tableau T de taille $(|u| + 1) \times (|v| + 1)$ tel que $T[i][j]$ contient la distance de Levenshtein entre $u[1..i + 1[$ et $v[1..j + 1[$. La propriété précédente se traduit ainsi sur les cases de T :

— $T[0, j] = j$ pour $0 \leq j \leq |v|$
— $T[i, 0] = i$ pour $0 \leq i \leq |u|$
— $T[i+1, j+1] = T[i, j]$ si $u_i = v_j$ pour $0 \leq i < |u|$ et $0 \leq j < |v|$
— $T[i+1, j+1] = 1 + \min \begin{cases} T[i, j+1] \\ T[i+1, j] \\ T[i, j] \end{cases}$ pour $0 \leq i < |u|$ et $0 \leq j < |v|$ sinon

Pour remplir une case, il faut donc que les trois cases en haut, à gauche et en haut à gauche soient déjà remplies. Pour un calcul de bas-en-haut, on pourra donc remplir le tableau ligne par ligne, ou bien colonne par colonne, ou bien même par anti-diagonales.

Exercice 15

Q1. Écrire l'algorithme de programmation dynamique de bas-en-haut permettant de calculer la distance de Levenshtein de deux mots ainsi.

Q2. Appliquer cet algorithme sur `BOOLEEN` et `TABLE`.

A nouveau, cet algorithme nous donne la valeur optimale du problème, c'est à dire la distance de Levenshtein, mais pas la suite d'opérations correspondante. Cependant, on peut modifier l'algorithme pour qu'il stocke dans le tableau T non seulement la valeur optimale mais aussi des informations permettant de reconstruire la solution. Plus précisément, on stocke dans chaque case $T[i][j]$ un symbole en plus de la valeur optimale :

— "-" si aucune opération n'a été effectuée, i.e. si $u_i = v_j$;
— "S" si l'on a supprimé u_i, autrement dit si dans le dernier cas de la formule de récurrence, le min était atteint par $T[i, j+1]$;
— "I" si l'on a inséré v_j avant u_{i+1}, autrement dit si dans le dernier cas de la formule de récurrence, le min était atteint par $T[i+1, j]$;
— "R" si l'on a remplacé u_i par v_j, autrement dit si dans le dernier cas de la formule de récurrence, le min était atteint par $T[i, j]$.

Pour reconstruire la solution, on lit le contenu de la case $T[|u|][|v|]$, et on suit le chemin fléché par les symboles que l'on lit, ce qui permet de reconstruire la suite d'opérations.

Exercice 16

On reprend le tableau dressé à l'exercice précédent. Annoter chaque case avec le type d'opération qui a été utilisé pour la remplir, et reconstruire la suite des opérations permettant de transformer `BOOLEEN` en `TABLE` en "suivant les flèches".

Remarque 3

En réalité, on peut même déduire les opérations utilisées uniquement à partir du tableau des valeurs, en "refaisant" les calculs !

E Réutiliser l'espace

Lorsque l'on fait de la Prog. Dyn. de bas en haut, il arrive qu'une formule de récurrence permette de remplir une ligne k du tableau uniquement en fonction de la ligne précédente $k-1$, sans utiliser les valeurs des lignes $0, 1, \ldots, k-2$. Dans ce cas, plutôt que d'allouer l'intégralité du tableau de Prog. Dyn., on peut allouer uniquement deux lignes, et les remplir l'une après l'autre alternativement.

Exercice 17

Prog. Dyn. : $\binom{n}{k}$

On s'intéresse au calcul de $\binom{n}{k}$ pour $0 \leq k \leq n$. L'algorithme consistant à utiliser la formule $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ a comme inconvénient que le calcul de $n!$ peut ne pas tenir sur un entier. On cherche un algorithme qui ne manipule jamais d'entier plus grand que le résultat attendu.

Q1. On définit la complexité spatiale d'un algorithme comme l'espace total qu'il doit réserver pour s'exécuter. Expliciter l'algorithme décrit ci-dessus, et donner ses complexités temporelles et spatiales.

Q2. Rappeler la formule classique du triangle de Pascal, et en déduire un algorithme récursif. Quelle est sa complexité temporelle ? et spatiale ?

Q3. Améliorer l'algorithme précédent avec de la programmation dynamique du bas vers le haut. Donner les complexités temporelles et spatiales.

Q4. Modifier l'algorithme précédent pour avoir une complexité spatiale linéaire, en ne gardant en mémoire que la ligne courante et celle qui vient d'être calculée.

5 Retour sur trace

On considère un problème de **décision** (i.e. attendant une réponse oui-non), où l'on peut décomposer toute instance $\mathcal{I}$ non-triviale en sous-instances $\mathcal{I}_1, \dots \mathcal{I}_n$ telles que si **au moins l'une** des $\mathcal{I}_k$ admet une solution, alors $\mathcal{I}$ en admet une. Les sous-instances correspondent à un ensemble de choix que l'on peut faire pour trouver une solution : il suffit que **l'un** des choix mène à une solution.
Dans les problèmes d'optimisations précédents, chercher une solution revient à explorer l'arbre des sous-instances : un algorithme brute-force explore de manière exhaustive, un algorithme glouton choisit une unique branche sans garantie que ce soit la bonne. Le principe du **retour sur trace** (ou **backtracking**) est de rajouter à l'exploration brute-force un critère permettant d'éliminer immédiatement certaines instances non-triviales.

Nous avons déjà vu un exemple fondamental d'algorithme de backtracking : l'algorithùe de Quine pour la satisfiabilité booléenne. Pour déterminer si une formule ϕ est satisfiable, on choisit une variable arbitraire, et on fait un choix : la fixer à 0 ou à 1. On explore alors chaque choix récursivement. Le critère qui permet de court-circuiter les calculs est le schéma de simplification, qui permet, dans certains cas, de transformer la formule d'entrée en $\top$ ou $\bot$, et donc de déterminer immédiatement qu'elle est satisfiable ou insatisfiable sans avoir à considérer toutes les variables de la formule initiale.

Voyons un nouvel exemple de problème pour lequel le backtracking est adapté. On considère le problème du Sudoku : étant donné une grille de Sudoku 4×4 partiellement remplie, est-il possible de la compléter tout en respectant les diverses contraintes ?
Une instance de ce problème est une grille partiellement remplie. A partir d'une grille G donnée, on peut fixer une case vide (par exemple la première dans l'ordre de lecture), et **choisir** la valeur que l'on y écrit : 1, 2, 3, ou 4. On obtient alors 4 sous-instances G_1, G_2, G_3, G_4, et G admet une solution si et seulement si l'une de ces G_i en admet une.
L'algorithme naïf consisterait ici à générer récursivement toutes les manières de compléter la grille, et de tester la validité seulement lorsque la grille a été complètement remplie. On observe deux choses :

— d'une part, l'arbre est immense (jusqu'à 16^4 feuilles) : en pratique, il est donc impensable d'effectuer cette recherche exhaustive ;

— d'autre part, certaines configurations incomplètes peuvent être immédiatement supprimées. Par exemple, si la grille possède deux 1 sur la première ligne, même s'il reste d'autres cases à remplir, il ne sert à rien de continuer à explorer l'arbre : on peut éliminer immédiatement tout le sous-arbre.

L'algorithme de retour sur trace est donc ici l'amélioration assez naturelle de l'algorithme brute-force : lorsque l'on remplit une case, on ne teste que les valeurs qui peuvent potentiellement convenir, et si aucune valeur ne convient, c'est que l'on peut arrêter l'exploration :

Algorithme 11 : `sudoku_backtrack`(G)

Entrée(s) : G une grille de Sudoku $n \times n$
Sortie(s) : G_s copie de G remplie et valide

```
si G est complètement remplie alors
    retourner Oui
(i, j) ← première case vide de G;
pour k = 1 à n faire
    si k n'apparaît ni sur la ligne i ni sur la colonne j, ni dans le carré
     contenant (i, j) alors
        G[i, j] ← k;
        si sudoku_backtrack(G) alors
            retourner Oui
// vider la case pour ne pas modifier la grille en sortie
G[i, j] ← 0;
retourner Non
```

Si aucune valeur k ne peut être écrite dans la case (i, j), alors l'algorithme ne teste aucune branche récursivement, et renvoie immédiatement Non.

Plus généralement, pour un problème $\mathcal{P}$, en supposant que l'on dispose d'un critère permettant de résoudre certaines instances trivialement, le schéma de backtracking est :

Algorithme 12 : `backtrack`$(\mathcal{I})$

Entrée(s) : $\mathcal{I}$ une instance de $\mathcal{P}$
Sortie(s) : Oui si $\mathcal{I}$ est positive, Non sinon

```
si I est une instance de base ou est gérée par le critère alors
    Résoudre directement I;
    retourner le résultat
Diviser I en sous-instances I_1, ... I_n pour k = 1 à n faire
    b ← backtrack(I_k);
    si b alors
        retourner Oui
retourner Non
```

Dans l'algorithme de résolution de Sudoku, le critère était appliqué non pas au début de l'appel récursif, mais directement au niveau des choix. A l'inverse, pour l'algorithme de Quine, les règles de simplification sont plutôt appliquées au début de l'appel récursif, pour voir si la formule est équivalente à $\top$ ou $\bot$: ces choix dépendent du problème, des choix d'implémentation, etc...

Remarque 4

En général, ce sont les solutions qui nous intéressent et pas simplement leur existence. En pratique, lorsque l'on écrit un algorithme de backtracking, on garde une trace des choix faits, afin de pouvoir reconstruire une solution plutôt que de simplement renvoyer Oui lorsqu'il en existe une.

Par exemple, pour l'algorithme du Sudoku proposé plus haut, la grille est modifiée en place, si bien que s'il existe une solution, elle sera stockée dans la grille en sortie. On aurait aussi pu créer une copie de la grille une fois la solution trouvée, pour ne pas modifier la matrice d'entrée.