# Schémas algorithmiques

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

28 avril 2024

# 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 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 $I \subseteq [\![1, n]\!]$ tel que :

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

**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 |

Définissons plus formellement ce qu'est un problème d'optimisation :

**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**.

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 prend en entrée des paramètres qui déterminent l'instance. On décrira donc un problème en en décrivant une instance générale. Par exemple :

Étant donné $w_1, \ldots, w_n \in \mathbb{R}^+$, $c_1, \ldots, c_n \in \mathbb{R}^+$ et $W \in \mathbb{R}^+$, le problème du sac à dos sur l'instance $((w_1, \ldots, w_n), (c_1, \ldots, c_n), W)$ 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 maximiser / minimiser, et l'espace des solutions exprime les **contraintes** à respecter.

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

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

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

**Exercice 1.** Déterminez une solution optimale lorsque les pièces ont comme montants $1, B, B^2, \ldots, B^{k-1}$ avec $B \in \mathbb{N}, B \geq 2$ et $k \in \mathbb{N}$, et montrez qu'on ne peut pas faire mieux.

## 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 $i \in [\![1, n]\!]$. Vous voulez assister à un maximum de spectacles, mais certains se chevauchent, il faut donc faire des choix. On autorisera les situations où un évènement commence à l'instant exact où un autre évènement finit.

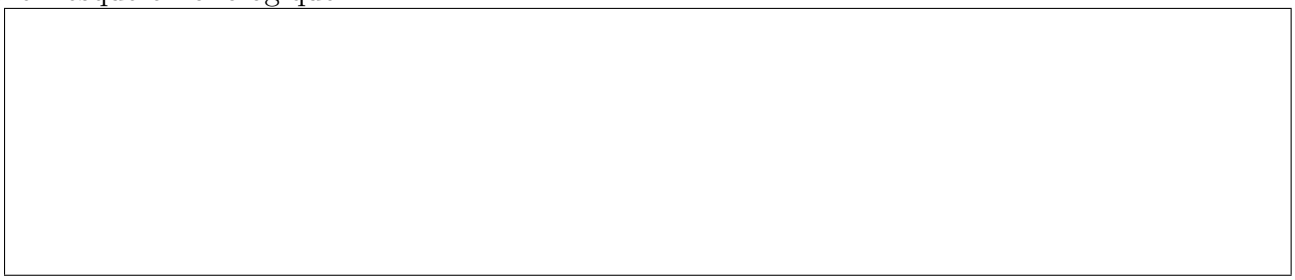
**Problème 1 : Spectacles**

**Entrée(s) :** $(d_1, f_1), \dots (d_n, f_n) \in \mathbb{R}^2$ avec $d_i < f_i$, $n$ horaires d'évènements
**Sortie(s) :** Nombre maximal d'évènements deux à deux disjoints

**Exemple 3.** On considère les évènements suivants :

| 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 |

Afin de bien visualiser le problème, on peut représenter les différents évènements à la manière d'une fresque chronologique :



Quel est le nombre maximal d'évènements auquel une seule personne peut assister ?

Formellement : étant donné $(d_1, f_1), \dots (d_n, f_n) \in \mathbb{R}^2$ avec $d_i < f_i$, le problème des spectacles sur l'instance $((d_1, f_1), \dots (d_n, f_n))$ est :

$$\textbf{SPECTACLES} : \max_{I \in S} |I|$$

Avec $S = \{I \subseteq [\![1, n]\!] \mid \forall i \neq j \in [\![1, n]\!], ]d_i, f_i[ \cap ]d_j, f_j[ = \emptyset\}$ l'ensemble des solutions admissibles, i.e. l'ensemble des parties de $[\![1, n]\!]$ ne contenant pas d'évènements se chevauchant.

## D Tri de tableau

Le problème du tri de tableau peut être vu comme un problème d'optimisation, comme suit. Étant donné un tableau $T = [x_1, \dots, x_n]$, le problème du tri est :

$$\textbf{TRI} : \min_{\sigma \in S_n} \left( \sum_{i=1}^{n-1} |x_{\sigma(i)} - x_{\sigma(i+1)}| \right)$$

Avec $S_n$ l'ensemble des permutations de $[\![1, n]\!]$. En effet, $\sum_{i=1}^{n-1} |x_{\sigma(i)} - x_{\sigma(i+1)}|$ est minimal lorsque les $x_i$ sont ordonnés par ordre croissant et vaut alors $x_{\sigma(n)} - x_{\sigma(1)} = \max(x_i) - \min(x_i)$.

## E Décomposition en sous-problèmes

De nombreux problèmes d'optimisation et de décision peuvent se décomposer 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 $X$ de l'instance originale.

On dit qu'un problème 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 $a_p$ :

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

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

— $\mathcal{I}_2 = (M - 2a_p, a_1, \dots, a_{p-1})$

— ...

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

— ...

Remarquons que si l'on trouve une solution optimale pour chaque instance $\mathcal{I}_k$, on peut immédiatement en déduire une solution 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 : on utilise les $m$ pièces nécessaires pour décomposer $M - ka_p$, puis en $k$ pièces de $a_p$ on complète la somme.

Notons qu'on dispose alors d'un algorithme permettant de résoudre le problème du rendu de monnaie :

**Algorithme 2 :** Rendu de monnaie : force brute

**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 ← 0 = +∞ // valeur de la solution actuelle
pour k = 0 à ⌊M/a_p⌋ faire
    y_1, ... y_{p-1} ← Résoudre l'instance (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
```

Cet algorithme est bien trop coûteux pour être utilisable en pratique.

**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.

A nouveau, cette remarque nous donne un algorithme en force brute :

**Algorithme 3 :** Programmation d'évènements : force brute

**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

```
1 si n = 0 alors
2     retourner ∅
3 I ← ∅ // meilleure solution actuelle
4 pour k = 1 à n faire
5     (i_1, ..., i_p ← indices des intervalles disjoints de (d_k, f_k);
6     I' ← Résoudre l'instance ((d_{i_1}, f_{i_1}), ..., (d_{i_p}, f_{i_p}));
7     si |I'| + 1 > |I| alors
8         I ← I' ∪ {k}
9 retourner X
```

Et a nouveau, cet algorithme est bien trop coûteux pour être utilisable en pratique.

# 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 ;

— Il 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.

Les algorithmes gloutons sont généralement simple à imaginer et à mettre en place, et ont des complexités assez faibles. Même si pour de nombreux problèmes, ils ne suffisent pas à trouver une solution optimale, ils permettent au moins de trouver une solution, plus ou moins proche de l'optimal.

## A Rendu de monnaie

Sur le problème de rendu de monnaie, l'algorithme le plus naturel (que l'on emploie dans la vie de tous les jours) est un algorithme glouton : il consiste à toujours utiliser la pièce de plus grande valeur possible :

**Algorithme 4 :** 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 $M$

```
1 Trier a_1, ..., a_p par ordre décroissant;
2 x_1, ..., x_p ← 0, ..., 0;
3 pour i = 1 à p faire
4 |   x_i ← ⌊M / a_i⌋;
5 |   M ← M − x_i a_i;
6 si M > 0 alors
7 |   retourner Pas de solution
8 sinon
9 |   retourner x_1 + ··· + x_p
```

**Exercice 2.**

**Question 1.** Appliquez cet algorithme sur le même système qu'au dessus, avec $M = 144$

**Question 2.** Trouver une instance (système de pièces + montant) pour lequel l'algorithme glouton ne renvoie pas la valeur optimale

**Question 3.** Trouver une instance où l'algorithme glouton ne trouve pas de solution alors qu'il y en a une.

Cet algorithme s'effectue en $\mathcal{O}(p \log p)$ (car il faut trier les pièces au départ). Il est donc polynomial en la taille de l'entrée. Cependant, 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. Montrons que le système $1, 2, 5, 10$ est canonique.

**Exercice 3.**

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.

**Question 1.** 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 $x_1^*, x_2^*, x_5^*, x_{10}^*$.

**Question 2.** De même, donner des bornes supérieures sur $x_2^*$ et $x_5^*$

**Question 3.** Montrer que $x_1^* + 2x_2^* + 5x_5^* < 10$.

**Question 4.** 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. Une première idée est d'essayer de choisir les objets par ordre décroissant de prix, autrement dit de toujours prendre l'objet le plus cher disponible parmi ceux qui rentrent dans le sac :

**Algorithme 5 :** Sac à dos glouton : prix décroissants

**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 par prix décroissant;
2 t ← W // taille actuelle du sac
3 p ← 0 // prix actuel du sac
4 pour i = 1 à n faire
5     si w_i ≤ t alors
6         t ← t − w_i;
7         p ← p + c_i;
8 retourner p
```

**Exercice 4.** Appliquer cet algorithme sur l'instance suivante (taille de sac max = 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 :

**Algorithme 6 :** Sac à dos glouton : poids croissant

**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 par poids croissant;
2 t ← W // taille actuelle du sac
3 p ← 0 // prix actuel du sac
4 pour i = 1 à n faire
5     si w_i ≤ t alors
6         t ← t − w_i;
7         p ← p + c_i;
8 retourner p
```

**Exercice 5.** 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.

**Algorithme 7 :** Sac à dos glouton : ratio décroissant

**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 par p_i/w_i décroissant;
2 t ← W // taille actuelle du sac
3 p ← 0 // prix actuel du sac
4 pour i = 1 à n faire
5     si w_i ≤ t alors
6         t ← t − w_i;
7         p ← p + c_i;
8 retourner p
```

**Exercice 6.**

**Question 1.** Appliquer cet algorithme sur l'exemple précédent.

**Question 2.** Montrer que cet algorithme glouton n'est PAS optimal, en exhibant un contre-exemple.

Bien que cet algorithme n'est pas optimal, on peut se demander s'il donne une solution proche de la solution optimale. L'année prochaine, vous étudierez les algorithmes d'approximation, dont le but est de résoudre des problèmes de manière approchée, en garantissant un certain ratio avec la solution optimale. Pour cet algorithme, on peut trouver des instances pour lesquelles il est arbitrairement mauvais :

**Question 3.** 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 $n$ évènements $(d_1, f_1), \ldots, (d_n, f_n)$. On considère le schéma suivant d'algorithme glouton :

**Algorithme 8 : 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 maximal 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
```

Donc, on regarde tous les évènements dans un certain ordre, et on ajoute tous les évènements que l'on peut ajouter à notre programmation. C'est un algorithme glouton car il ne réfléchit pas aux évènements qu'il va rencontrer plus tard, et car il 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 7.**

**Question 1.** 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.

**Question 2.** 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 $(d_1, f_1), \ldots, (d_n, f_n)$ 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 $I^* = \{i_1, \ldots, i_k\} \subseteq [\![1, n]\!]$ une solution optimale, et $I = \{j_1, \ldots, j_l\} \subseteq [\![1, n]\!]$ la solution renvoyée par l'algorithme glouton. Remarquons que $j_1 = 1$ car le premier intervalle est toujours pris.

**Question 3.** Montrez qu'il existe une solution $I'$ optimale qui contient le premier évènement $(d_1, f_1)$. *Indication : montrez que l'on peut remplacer un des évènements de $I^*$ par $(d_1, f_1)$.*

**Question 4.** En suivant le même raisonnement, montrez par récurrence sur $t \in [\![0, n]\!]$ que $I_t^* = \{j_1, \ldots, j_t, i_{t+1}, i_k\}$ est une solution admissible, et qu'elle est optimale.

**Question 5.** En déduire que la solution renvoyée par l'algorithme glouton est optimale.

# 3 Diviser pour régner

La stratégie "diviser pour régner" (DPR) s'applique lorsque l'on peut séparer une instance $\mathcal{I}$ d'un problème en plusieurs petites instances $\mathcal{I}_1, \ldots, \mathcal{I}_p$, de telle sorte qu'à partir d'une solution sur les $\mathcal{I}_i$, on peut construire une solution pour $\mathcal{I}$.

Les algorithmes DPR sont généralement récursifs, et leurs complexités font toujours apparaître des relations de récurrence, qu'il faut résoudre pour déterminer la complexité asymptotique.

## A Tri de tableau

Nous avons déjà vu deux algorithmes de tri DPR : le tri fusion et le tri rapide. Dans les deux 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 de $\frac{n}{2}$ cases.

## B Multiplication de polynomes

On considère $P(X) = \sum_{i=0}^{n} a_i X^i$ et $Q = \sum_{i=0}^{n} b_i X^i$ deux polynômes de degré $n$. On souhaite calculer leur produit $PQ(X) = \sum_{i=0}^{n} (\sum_{j=0}^{i} a_j b_{i-j}) X^i$. Pour simplifier les calculs, on suppose que $n$ est une puissance de 2. L'algorithme naïf consiste à calculer un par un chacun des coefficients :

**Algorithme 9 :** Produit naïf de polynomes

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

```
1 R ← 0;
2 pour i = 0 à n faire
3 |   pour j = 0 à i 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. On commence par décomposer $P$ 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 :

$$PQ(X) = P_0 Q_0 + (P_0 Q_1 + P_1 Q_0) X^{\frac{n}{2}} + P_1 Q_1 X^n$$

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 10 :** Multiplication naïve de polynomes

**Entrée(s) :** $P, Q$ deux polynomes de degré $n$
**Sortie(s) :** $PQ$ polynome produit de $P$ et $Q$

**si** $n = 1$ **alors**
  Calculer $R = PQ$ par calcul direct ;
  **retourner** $R$
Décomposer $P = P_0 + X^{\frac{n}{2}} P_1$;
Décomposer $Q = Q_0 + X^{\frac{n}{2}} Q_1$;
Calculer $R_0 = P_0 Q_0$ récursivement ;
Calculer $R_1 = P_0 Q_1$ récursivement ;
Calculer $R_2 = P_1 Q_0$ récursivement ;
Calculer $R_3 = P_1 Q_1$ récursivement ;
**retourner** $R_0 + (R_1 + R_2)X^{\frac{n}{2}} + R_3 X^n$

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 $C(n)$ :

Donc, cet algorithme DPR n'est pas meilleur que l'algorithme naïf ! 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 $\frac{n}{2}$. C'est précisément l'objet de l'algorithme de **Karatsuba**. Cet algorithme vient de l'observation suivante : si l'on note $A = P_0 Q_0$, $B = P_1 Q_1$ et $C = (P_0 + Q_1)(Q_0 + Q_1)$ alors on a :

$$\begin{aligned} R_0 &= A \\ R_3 &= B \\ R_1 + R_2 &= C - A - B \end{aligned}$$

Autrement dit, on retrouve les trois termes qui apparaissent dans l'algorithme de multiplication précédent $(R_0 + (R_1 + R_2)X^{\frac{n}{2}} + R_3 X^n)$ mais on n'a eu besoin d'effectuer que 3 multiplications sur des polynômes de taille moitié.

Donc, on peut modifier l'algorithme DPR précédent comme suit.

**Algorithme 11 :** Algorithme de Karatsuba

**Entrée(s) :** $P, Q$ deux polynomes de degré $n$
**Sortie(s) :** $PQ$ polynome produit de $P$ et $Q$

**si** $n = 1$ **alors**
    Calculer $R = PQ$ par calcul direct ;
    **retourner** $R$
Décomposer $P = P_0 + X^{\frac{n}{2}} P_1$;
Décomposer $Q = Q_0 + X^{\frac{n}{2}} Q_1$;
Calculer $A = P_0 Q_0$ récursivement ;
Calculer $B = P_1 Q_1$ récursivement ;
Calculer $C = (P_0 + P_1) \times (Q_0 + Q_1)$ récursivement ;
**retourner** $A + (C - B - A)X^{\frac{n}{2}} + BX^n$

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

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

**Proposition 1.** Cette relation de récurrence se résoud en $C(n) = \Theta(n^{\log_2 3})$

De plus, on a $\log_2 3 \approx 1.585$, ce qui donne une complexité asymptotique en $\mathcal{O}(n^{1.585})$, nettement meilleure que celle de l'algorithme naïf en $\mathcal{O}(n^2)$.

## C Actions

**Exercice 8.** 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.

**Question 1.** Formaliser le problème d'optimisation.

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

**Question 3.** En utilisant la méthode diviser pour régner, proposer un algorithme plus efficace en $\mathcal{O}(n \log n)$.

**Question 4.** 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. Cependant, dans certains problèmes, même si l'on peut trouver une formule de récurrence qui permettrait d'appliquer un algorithme de type DPR, on se retrouvait à résoudre plusieurs fois les mêmes sous-instances, ce qui donnerait une complexité très mauvaise.

La programmation dynamique consiste à **stocker en mémoire** les solutions de toutes les sous-instances que l'on calcule, afin de ne pas avoir à les recalculer plusieurs fois.

## A 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 :

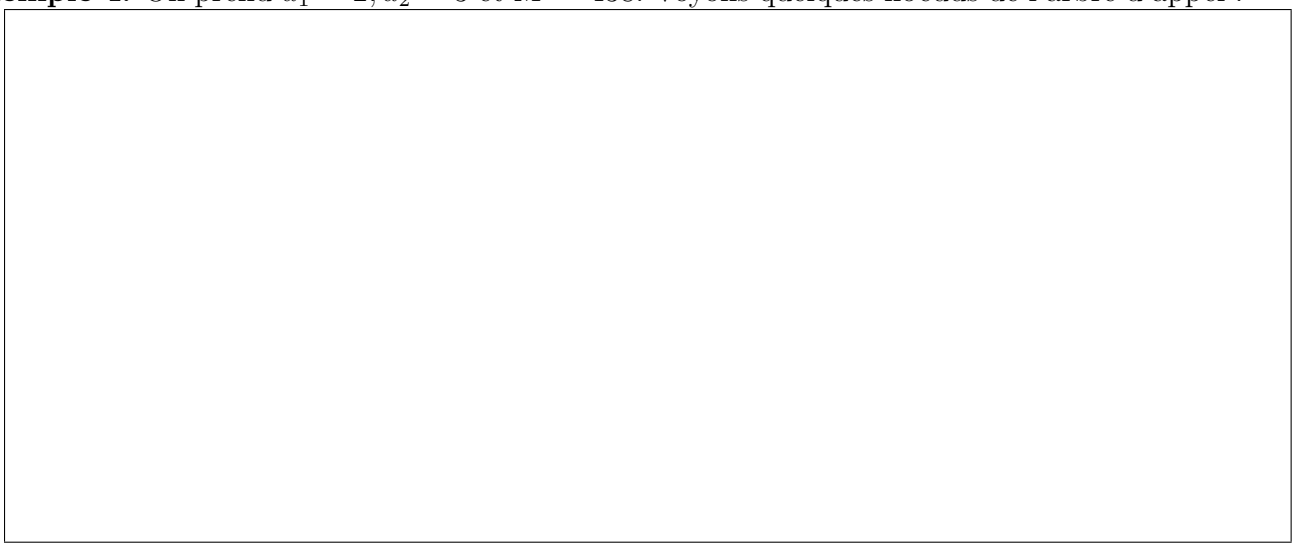
$$\begin{array}{lcll} C(0,i) & = & 0 & \text{pour } 0 \leq i \leq n \\ C(x,0) & = & +\infty & \text{pour } x > 0 \\ C(x,i+1) & = & \min(C(x,i), C + (x - a_{i+1}, i+1)) & \text{si } x \geq a_i, \text{ pour } i \geq 0 \\ C(x,i+1) & = & C(x,i) & \text{sinon, pour } i \geq 0 \end{array}$$

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 DPR : 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.

Déroulons la récurrence sur un exemple pour voir pourquoi cette méthode est extrêmement inefficace.

**Exemple 4.** On prend $a_1 = 2, a_2 = 3$ et $M = 438$. Voyons quelques noeuds de l'arbre d'appel :



On se rend donc compte que l'on va calculer de très nombreuses fois les mêmes valeurs de $C(x, i)$. En fait, dans le pire des cas, la complexité de notre algorithme sera en $\mathcal{O}(2^{n+M})$.

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 $(M+1)\times(n+1)$. Le but est donc que chaque case $T[x][i]$ contienne $C(x,i)$.
Remarquons que certaines cases sont triviales à remplir : $T[0][i]$ vaut 0 pour $i \in [\![0,n]\!]$ et $T[x][0]$ vaut $+\infty$ pour $x > 0$.
Remarquons ensuite que pour remplir une case $T[x][i]$, il suffit de connaître la valeur des cases $T[y][i]$ avec $y < x$ et de la case $T[x][i-1]$. Autrement dit, pour remplir une case, il suffit d'avoir déjà rempli les cases à sa gauche et au dessus :

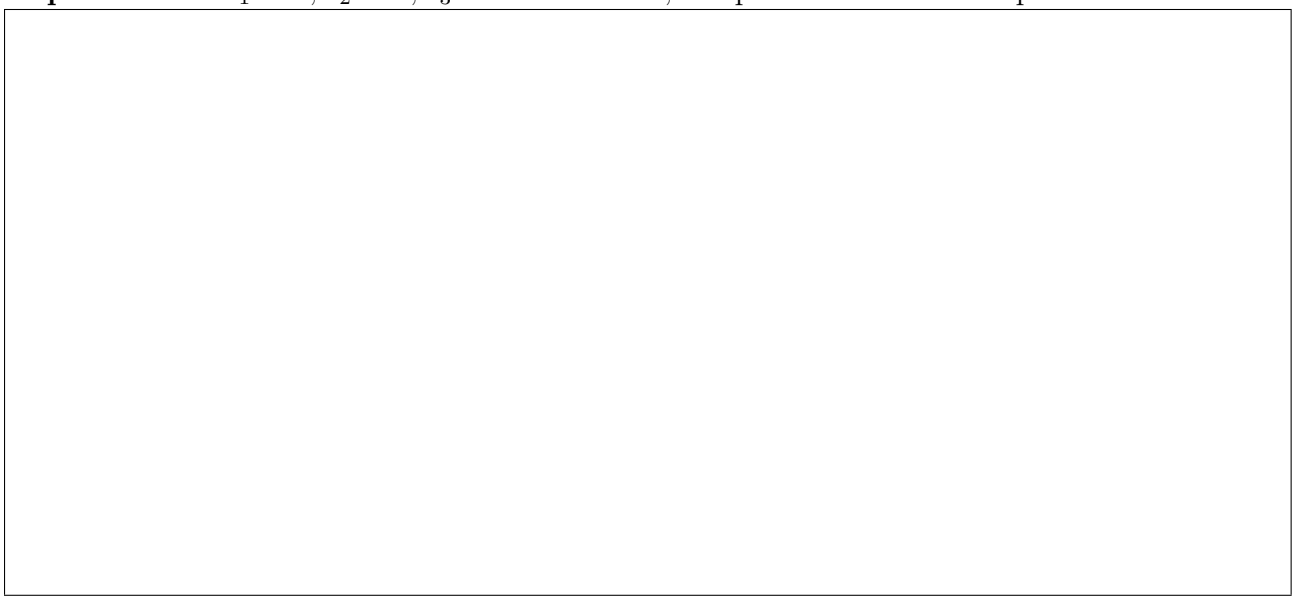
On peut donc remplir le tableau $T$ ligne par ligne, de gauche à droite :

**Algorithme 12 :** Rendu de monnaie : Prog. Dyn.

**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[M − a_i][i]);
10         sinon
11             T[x][i] ← T[x][i−1];
12 retourner T[M][n]
```

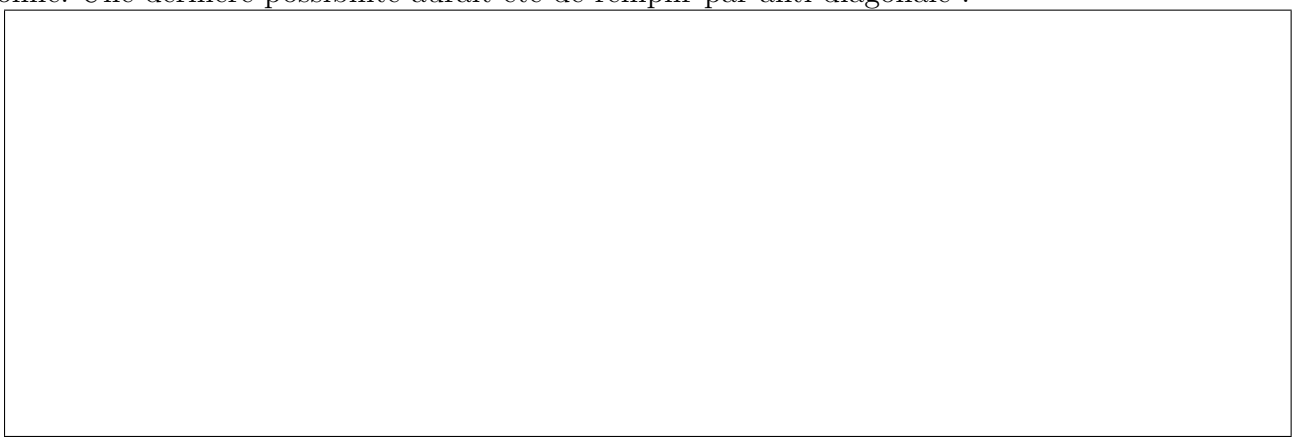
**Exemple 5.** Pour $a_1 = 2, a_2 = 3, a_3 = 7$ et $M = 13$, remplir le tableau correspondant :



La complexité de cet algorithme est en $\mathcal{O}(nM)$, car il faut un temps constant pour remplir chaque case. Notons que ce n'est **pas** polynomial en la taille de l'instance, car $M$ sera donné en écriture binaire, sa taille est donc $\log_2 M$.

**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 pour résoudre une instance donnée.

En pratique, pour implémenter de la programmation dynamique, on utilise un tableau pour stocker les solutions aux différentes sous-instances. On initialise ce tableau en utilisant les cas de base du problème, puis on résout les instances, des plus petites aux plus grandes, comme dans l'exemple précédent. Il faut donc pouvoir identifier l'ordre dans lequel remplir le tableau. Pour le rendu de monnaie, on pouvait remplir ligne par ligne, i.e. de haut en bas puis de gauche à droite. On aurait aussi pu remplir de gauche à droite puis de haut en bas, i.e. colonne par colonne. Une dernière possibilité aurait été de remplir par anti-diagonale :



Lorsque l'on conçoit un algorithme en programmation dynamique de bas en haut, les étapes sont :

1. Identifier une formule de récurrence sur les solutions des différentes instances du problème ;
2. Déterminer dans quel ordre remplir le tableau.

**En annexe** (voir sur le site du cours) : une implémentation de l'algorithme de Prog. Dyn. de rendu de monnaie en C.

## B Construction de haut en bas

Dans le problème précédent, on a opéré par une construction de **bas en haut** : on initialise le tableau avec les cas de base, puis on calcule les solutions d'instances de plus en plus grandes.

En Prog. Dyn., on peut appliquer une autre approche, dite de **haut en bas**, récursive. On considère un problème d'optimisation, que l'on veut résoudre par programmation dynamique, et on utilise un tableau $T$ global pour stocker les solutions des différentes sous-instances. On notera donc $T[I]$ la valeur stockée dans $T$ à la case donnée par les paramètres de l'instance $I$.

Autrement dit, cet algorithme récursif garde en mémoire les valeurs déjà calculées, et évite de les recalculer. On parle de **mémoïsation** (ou de mise en cache).

Appliquons cette idée sur un exemple, et profitons-en pour introduire un nouvel aspect de la programmation OCaml : la **programmation impérative**. On peut manipuler des tableaux et des variables mutables classiques en OCaml.

Pour créer un tableau en OCaml, on utilise la fonction `Array.make`, qui est de type `int -> 'a -> 'a array` Ainsi, `Array.make n x` crée un tableau de taille $n$ rempli de $x$.

**Algorithme 13 :** Prog. Dyn. de haut en bas

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

1 **si** $T[I]$ ***est vide*** **alors**
2  Décomposer $I$ en sous-instances $I_1, \dots I_p$;
3  Calculer récursivement des solutions optimales $S_1, \dots, S_p$ pour $I_1, \dots, I_p$;
4  Combiner $S_1, \dots, S_p$ en une solution $S$ de $I$;
5  $T[I] \leftarrow S$;
6 **retourner** $T[I]$

On peut ensuite accéder à la $i$-ème case de $t$ avec `t.(i)`. De plus, les tableaux sont des structures mutables, ce qui veut dire qu'on peut modifier leur contenu (contrairement aux listes par exemple). Pour écrire $x$ dans la $i$-ème case de $t$, on écrit `t.(i) <- x`. Comme tout en OCaml, c'est une expression, et elle a pour type `unit`. On peut donc la voir comme une "instruction", au même titre que les fonction d'affichage comme `print_int` ou `print_string`. Voici un exemple de code permettant d'utiliser les tableaux :

```
let t = Array.make 10 3 (* crée un tableau de 10 cases valant 3 *)

(* double la valeur contenue dans t.(i) et renvoie la nouvelle valeur *)
let double_et_renvoie (t: int array) (i: int) : int =
    t.(i) <- 2*t.(i);
    t.(i)

let x = double_et_renvoie t 2 (* x = 6 , t.(2) contient maintenant 6 *)
let y = double_et_renvoie t 2 (* y = 12 , t.(2) contient maintenant 12 *)
```

Utilisons les tableaux pour calculer par programmation dynamique les termes de la suite de Fibonacci. L'algorithme récursif naïf de calcul de la suite s'écrit en OCaml :

```
let rec fibo (n: int) : int =
  if n <= 1 then 1
  else fibo (n-1) + fibo(n-2);;
```

Sa complexité est en $\mathcal{O}(\Phi^n)$, avec $\Phi = 1.618$ le nombre d'or. Ainsi, même le calcul de `fibo 60` prendra plus d'une heure. Appliquons maintenant le principe de mémoïsation à cette fonction :

```
let fibo (n: int) =
   (* initialisation: -1 indique une case vide *)
  let t = Array.make (n+1) (-1) in
  t.(0) <- 0;
  t.(1) <- 1;
  (* renvoie le i-ème terme de la suite de fibonacci et stocke le résultat
     dans t.(i) s'il n'y est pas déjà *)
  let rec calcul (i: int) : int =
    (if t.(i) == -1 then t.(i) <- calcul(i-1) + calcul(i-2));
    t.(i)
  in calcul n
```

Avec cette version, la complexité est en $\mathcal{O}(n)$. En effet, on ne calcule le contenu de chaque case qu'une seule fois.

## C Distance de Levenshtein

On s'intéresse à un problème classique d'algorithmique du texte. On considère un alphabet $\Sigma$ 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 $\Sigma$.

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

**Proposition 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 3.** 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 derniers cas reflètent respectivement l'effet de la suppression, de l'insertion et du remplacement.

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

Ainsi, pour remplir une case, il faut 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 9.**

**Question 1.** Écrire l'algorithme de programmation dynamique de bas-en-haut permettant de calculer la distance de Levenshtein de deux mots ainsi.

**Question 2.** 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 $T[i][j]$ de 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 $T[i+1,j[$ ;

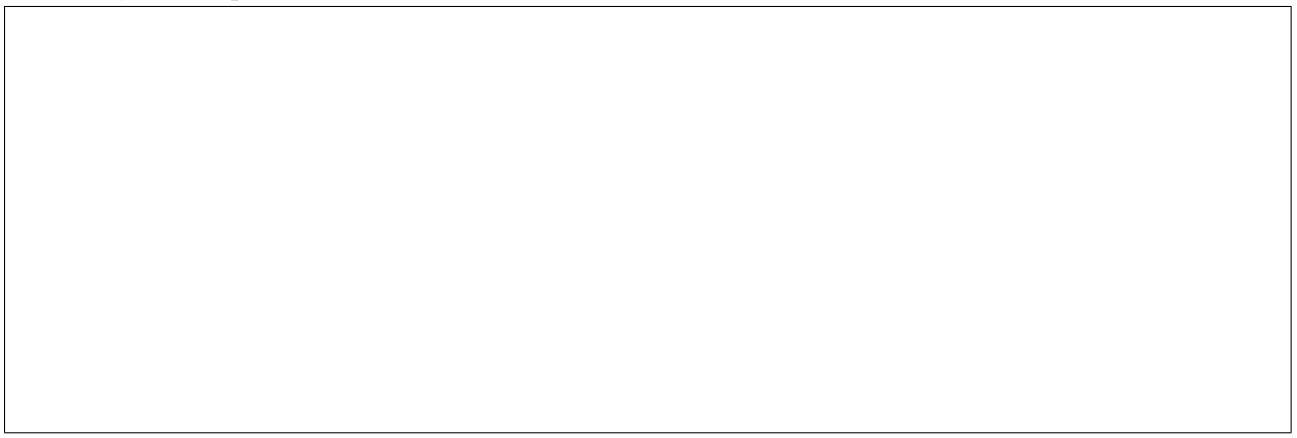
— "R" si l'on a remplacé $u_i$ par $v_j$, autrement dit si dans le dernier cas de la formule de récurrence, le min était atteint par $T[i,j[$.

Puis, pour reconstruire la solution, on lit le contenu de la case $T[|u|][|v|]$. Si on lit "-" ou "R", alors on sait que la case a été construite à partir de la case en haut à gauche. Si on lit "I" on sait que la case a été construite à partir de la case à gauche, et si on lit "S", alors on sait que la case a été construite à partir de la case en haut. On peut donc récursivement lire le contenu de la case en question, et procéder ainsi jusqu'à atteindre le bord du tableau.

**Exercice 10.** 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".

# 5 Retour sur trace

Les problèmes étudiés précédemment, qui peuvent se modéliser comme des suites de choix, font naturellement apparaître une structure d'arbre. Dans une telle modélisation, un noeud de l'arbre correspond à une instance. Les enfants d'un noeud sont les sous-instances générées lors de la résolution, et la racine est l'instance initiale étudiée. Cette situation s'adapte particulièrement bien pour les jeux, où les choix à faire correspondent à des coups. Prenons par exemple le jeu du sudoku. Pour simplifier, on considère des grilles de Sudoku $4 \times 4$. On veut modéliser le jeu de Sudoku par un arbre : on considère une grille de départ, qui sera la racine de notre arbre. Puis, on construit les noeuds de profondeur 1, qui sont toutes les grilles obtenues à partir de la grille racine, en remplissant exactement une case :



On construit ensuite tous les noeuds de profondeur 2 selon la même méthode :

Ainsi de suite, on construit tout l'arbre. Les feuilles sont des grilles remplies, mais celles qui ne satisfont pas les conditions du Sudoku sont invalides. Ainsi, chercher une solution de la grille initiale, c'est chercher une feuille valide.

On observe deux choses. D'une part, l'arbre est immense. En pratique, il est donc impensable d'effectuer une recherche exhaustive, c'est à dire de parcourir tout l'arbre dans son intégralité. 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.

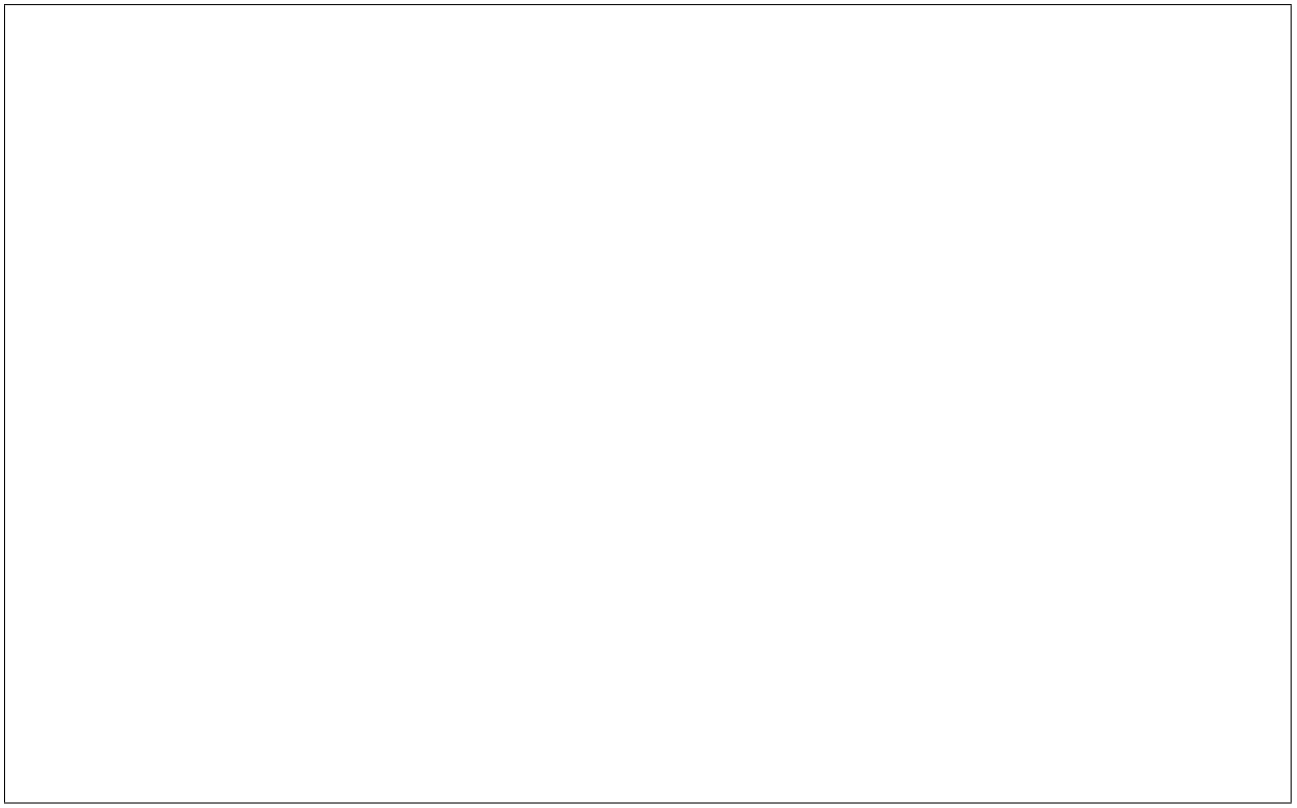
Voici un algorithme de résolution de Sudoku, qui exploite cette remarque :

**Algorithme 14 :** Résolution de Sudoku

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

**si** $G$ ***est invalide*** **alors**
  **retourner** ***Pas de solution***
**si** $G$ ***est complète*** **alors**
  **retourner** $G$
$(i, j) \leftarrow$ première case vide de $G$`// dans l'ordre ligne par ligne par exemple`
**pour** $k = 1$ **à** $n$ **faire**
  Résoudre récursivement la grille de Sudoku $G'$ obtenue en copiant $G$ avec $G'[i, j] = k$;
  **si** ***une solution*** $G_s$ ***a été trouvée*** **alors**
    **retourner** $G_s$
**retourner** ***Pas de solution***

Appliquons cet algorithme sur un exemple pour voir à quoi ressemble schématiquement son exécution :



Cet algorithme utilise le principe de retour sur trace (ou **backtracking**), qui est donc un parcours d'arbre en profondeur, en revenant en arrière dès qu'il ne sert plus à rien d'explorer un noeud.

Le backtracking sert donc lorsque l'on étudie un problème de décision $\mathcal{P}$, tel que toute instance $\mathcal{I}$ peut être cassée en sous-instances $\mathcal{I}_\infty, \ldots, \mathcal{I}_\backslash$ telles que $\mathcal{I}$ est positive si et seulement si au moins une des $\mathcal{I}_\parallel$ l'est. Il faut également disposer d'un **critère** permettant de détecter rapidement les instances négatives. Pour les sudoku par exemple, si un même chiffre figure dans une ligne, un colonne ou un bloc plus de deux fois, on sait immédiatement que l'instance est négative : elle est invalide donc n'a pas de solution.

Alors, on peut utiliser le principe de retour sur trace pour résoudre $\mathcal{P}$, comme suit :

**Algorithme 15 :** Schéma de retour sur trace

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

```
si le critère détermine que I n'est pas une instance valide alors
    retourner Non
si I est une instance de base alors
    Résoudre directement I;
    retourner le résultat
pour J sous-instance de I faire
    Résoudre récursivement J si J est positive alors
        retourner Oui
retourner Non
```

Remarquons que l'on s'intéresse à des problèmes de décision. Donc, a proprement parler, les algorithmes de résolution ne doivent renvoyer que Oui ou Non. En pratique, ce qui nous intéresse est plutôt de pouvoir donner une solution si la réponse est Oui. Par exemple pour le problème du Sudoku on ne veut pas juste savoir si une grille donnée admet une solution, mais aussi quelle est cette solution. En pratique, on modifie donc l'algorithme de backtracking pour garder une **trace** des choix faits.

Nous avons vu dans le chapitre précédent un exemple fondamental d'algorithme de retour sur trace : **l'algorithme de Quine**. Dans l'implémentation, nous avons vu commenter modifier l'algorithme pour qu'il renvoie une solution, et pas juste une réponse booléenne.

**Exercice 11.** On considère le problème **SUBSET-SUM** : Étant donné $x_1, \ldots, x_n, S \in \mathbb{N}$, existe-t'il $I \subseteq [\![1, n]\!]$ tel que $\sum_{i \in I} x_i = S$ ? Autrement dit, étant donné un ensemble d'éléments $\{x_1, \ldots x_n\}$, existe t-il un sous-ensemble de somme $S$ ?

**Question 1.** Proposer un algorithme par force brute, et donner sa complexité.

**Question 2.** Améliorer l'algorithme précédent en utilisant le principe de backtracking.

**Question 3.** Modifier l'algorithme pour qu'il renvoie le sous-ensemble réalisant la somme, lorsqu'il existe.

**Exercice 12.**

On considère le problème de segmentation de texte. On se donne un texte dans lequel les espaces ont été enlevés, et l'on souhaite reconstruire le texte d'origine. Par exemple, on considère le texte `DEPARTINITIALEMENTPREVUAVINGTHEURES` : un texte d'origine possible est `DEPART INITIALEMENT PREVU A VINGT HEURES`. Cependant, on pourrait segmenter le début du texte comme `DE PARTI NI`, et de manière générale, pour un texte long il peut être compliqué de reconstruire un texte d'origine possible.

On se donne un alphabet $\Sigma$ fini, et un ensemble $D \subseteq \Sigma^*$ fini de mots. Le problème de segmentation de texte est de savoir, étant donné un texte $t \in \Sigma^*$, si l'on peut peut trouver $u_1, \ldots, u_n \in D$ tels que $t = u_1.u_2 \ldots u_n$.

**Question 1.** Proposer un algorithme naïf, et donner sa complexité.

**Question 2.** Proposer un algorithme glouton, donner sa complexité et étudier sa correction.

**Question 3.** Proposer un algorithme utilisant le principe de retour sur trace pour améliorer l'algorithme naïf.