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 $w_1, \ldots, w_n \in \mathbb{R}^+$, $c_1, \ldots, c_n \in \mathbb{R}^+$ 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 (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 $a_1, \ldots, a_p$ (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 :

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

avec $S = \{(x_1, \ldots, x_n) \in \mathbb{N}^n \mid x_1a_1 + \cdots + x_pa_p = M\}$ 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 $E_1, \ldots E_n$ 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 d'évènements, mais certains se chevauchent, il faut donc faire des choix. On considèrera que l'on peut assister à deux évènements même si le deuxième commence à l'instant où le premier finit.

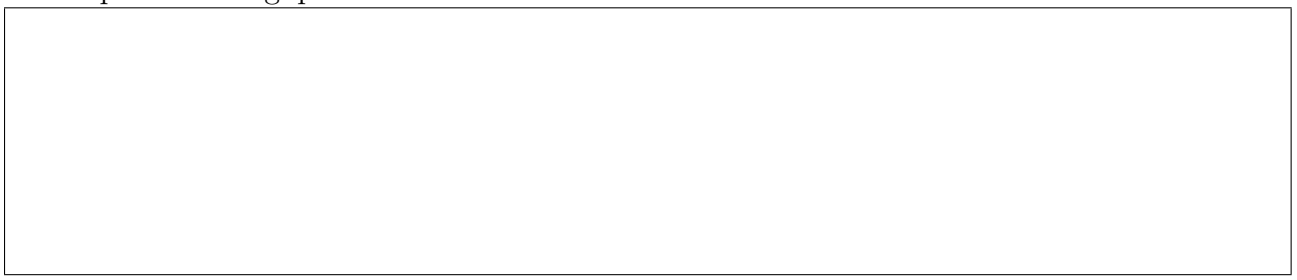
Problème 1 : Spectacles

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) : Nombre maximal d'évènements deux à deux disjoints

Exercice 1. 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 :



Question 1. Sur cet exemple, quel est le nombre maximal d'évènements auquel une seule personne peut assister ?

Question 2. On considère n intervalles $(d_1, f_1), \ldots (d_n, f_n) \in \mathbb{R}^2$ avec $d_i < f_i$. Décrire l'ensemble des solutions admissibles, ainsi que la fonction objectif, et définir formellement le problème des spectacles sur l'instance $(d_1, f_1), \ldots (d_n, f_n)$.

D Tri de tableau

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

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 $\mathcal{I}_1, \ldots \mathcal{I}_n$ telles qu'à partir de solutions $X_1, \ldots X_n$ 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 $\mathcal{I} = (M, a_1, \ldots, a_p)$ du problème de rendu de monnaie, où M est le montant à décomposer et $a_1, \ldots a_p$ 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 $\mathcal{I}_0, \ldots \mathcal{I}_{\lfloor \frac{M}{a_p} \rfloor}$ où $\mathcal{I}_k$ revient à avoir utilisé k pièces de valeur a_p :

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

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

— ...

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

— ...

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 : on utilise les m pièces nécessaires pour décomposer $M - ka_p$, puis en k pièces de a_p on complète la somme. 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 :

Algorithme 2 : `ForceBrute`$(M, a_1, \ldots, a_p)$

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

si $M = 0$ **alors**
  retourner $(0, \ldots, 0)$
si $p = 0$ ***(i.e. il n'y a pas de pièces)*** **alors**
  retourner ***IMPOSSIBLE***
$X \leftarrow NULL$ `// meilleure solution actuelle`
$s \leftarrow +\infty$ `// valeur de la solution actuelle`
pour $k = 0$ **à** $\lfloor \frac{M}{a_p} \rfloor$ **faire**
  $y_1, \ldots y_{p-1} \leftarrow$ `ForceBrute`$(M - ka_p, a_1, \ldots a_{p-1})$;
  si $y_1 + \cdots + y_{p-1} + k < s$ **alors**
    $X \leftarrow (y_1, \ldots, y_{p-1}, k)$;
    $s \leftarrow y_1 + \cdots + y_{p-1} + k$;
retourner X

Cet algorithme est bien trop coûteux pour être utilisable en pratique, puisqu'il énumère toutes les possibilités.

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 : `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 $\emptyset$
$I \leftarrow \emptyset$ `// meilleure solution actuelle`
pour $k = 1$ **à** n **faire**
  $(i_1, \ldots, i_p) \leftarrow$ indices des intervalles disjoints de (d_k, f_k);
  $I' \leftarrow$ `ForceBrute`$((d_{i_1}, f_{i_1}), \ldots, (d_{i_p}, f_{i_p}))$;
  si $|I'| + 1 > |I|$ **alors**
    $I \leftarrow I' \cup \{k\}$
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.

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.

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 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. Si 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 : $\lfloor \frac{M}{a_i} \rfloor$.

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

```
Trier a_1, ..., a_p par ordre décroissant;
x_1, ..., x_p ← 0, ..., 0;
pour i = 1 à p 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 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 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 $\mathcal{O}(p \log p)$ (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.

Exercice 3. Soit $B \in \mathbb{N} \setminus \{0,1\}$ et $k \in \mathbb{N}^*$. On considère le système de pièces $S_k = (1, B, B^2, \ldots, B^{k-1})$.

Question 1. Que renvoie l'algorithme glouton lors de la décomposition d'un entier M dans S_k ?

Question 2. 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.

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.

Question 3. 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 4. De même, donner des bornes supérieures sur x_2^* et x_5^*

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

Question 6. 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 5 : Sac à dos glouton

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

```
Trier les objets selon un certain critère;
t ← W // taille actuelle restante dans le sac
p ← 0 // prix actuel du sac
pour i = 1 à n faire
    si w_i ≤ t alors
        t ← t − w_i;
        p ← p + c_i;
retourner p
```

Par exemple, une première idée est de choisir en priorité les objets les plus chers, i.e. de trier par **prix décroissant**, afin d'avoir $c_1 \geq c_2 \geq \cdots \geq c_n$. En effet, un objet cher est, a priori, avantageux pour construire une bonne solution.

Exercice 4. 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**.

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.

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. 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 6 : 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 $(d_1, f_1), \dots, (d_n, f_n)$ 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.

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 le premier évènement de I^* par (d_1, f_1).*

Question 4. En suivant le même raisonnement, montrez par récurrence sur $t \in [\![0, n]\!]$ que $I_t^* = \{j_1, \dots, j_t, i_{t+1}, i_k\}$ 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

Dans les problèmes étudiés pour l'instant, les sous-instances que l'on construit sont généralement de tailles proches de l'instance de base, et se recouvrent les unes les autres. 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 deux 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 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 $C(n) = \mathcal{O}(n \log n)$.

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 $[3, -8, 5]$. Notons qu'alors, la somme d'un polynôme P à n coefficients et d'un polynôme Q à m coefficients se fait en temps linéaire $\mathcal{O}(n + m)$. De plus, ajouter un terme aX^i à un polynôme 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 :

Algorithme 7 : Produit naïf de polynomes

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 degré $< n$
Sortie(s) : $R = PQ(X)$

```
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. 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 :

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

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 8 : `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 $C(n)$:

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

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

Autrement dit, on retrouve les trois termes qui apparaissent dans l'algorithme de multiplication précédent $(R_0 + (R_1 + R_2)X^{\frac{n}{2}} + R_3X^n)$ 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 9 : Multiplication de polynômes : algorithme de Karatsuba

Entrée(s) : P, Q deux polynômes de degré n
Sortie(s) : PQ
1 **si** $n = 0$ **alors**
2 Calculer $R = PQ$ par calcul direct ;
3 **retourner** R
4 Décomposer $P = P_0 + X^{\frac{n}{2}}P_1$;
5 Décomposer $Q = Q_0 + X^{\frac{n}{2}}Q_1$;
6 Calculer $A = P_0Q_0$ récursivement ;
7 Calculer $B = P_1Q_1$ récursivement ;
8 Calculer $C = (P_0 + P_1) \times (Q_0 + Q_1)$ récursivement ;
9 **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 8. Dérouler cette relation et déterminer la complexité asymptotique de l'algorithme de Karatsuba.

B Actions

Exercice 9. 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 $\mathcal{O}(n \log n)$.

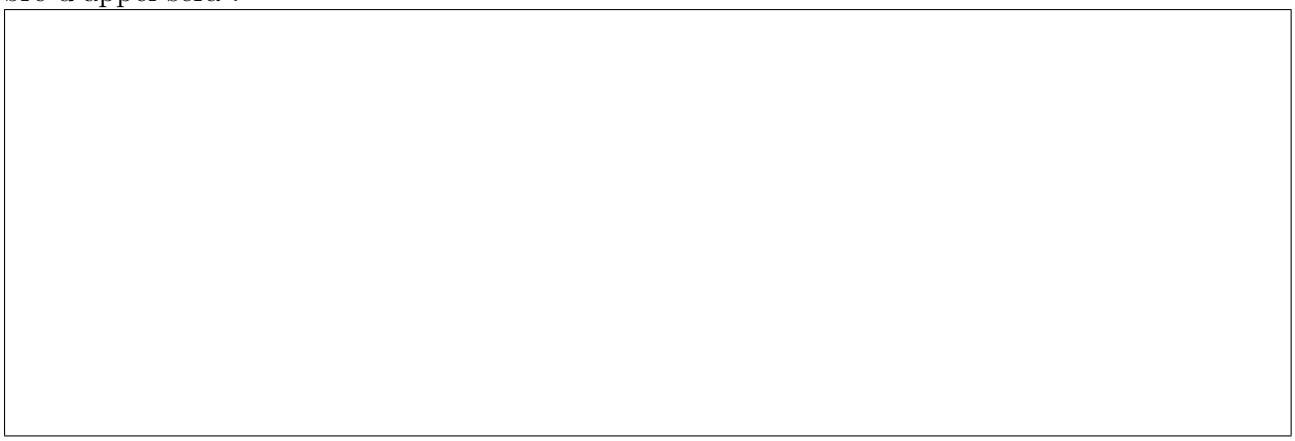
Question 4. Améliorer l'algorithme précédent en $\mathcal{O}(n)$. (*Indication : calculez toutes les grandeurs dont vous avez besoin en même temps, avec une seule fonction*).

4 Programmation dynamique

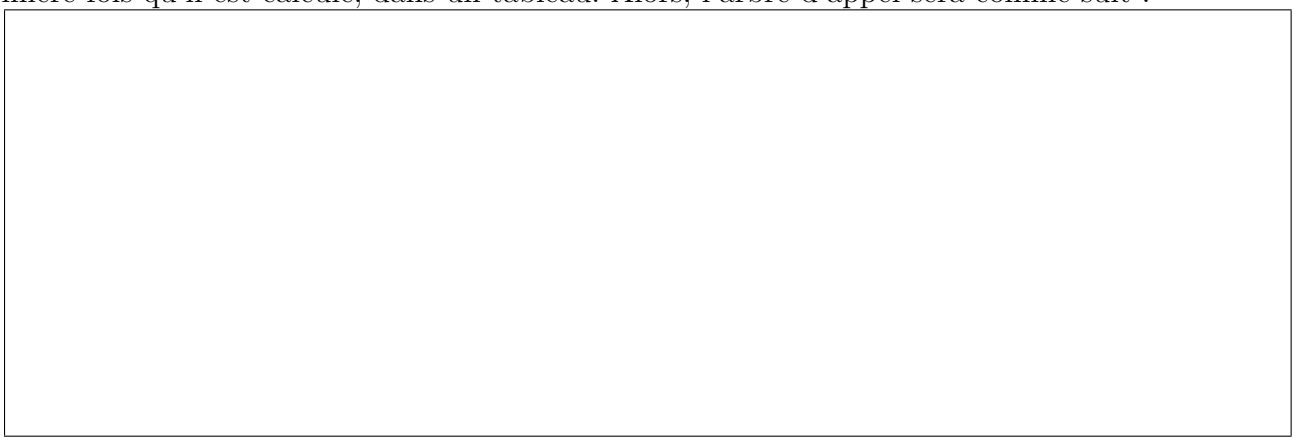
Dans la section précédente, les problèmes que nous avons résolu par la stratégie DPR avaient un point commun : les sous-instances construites à partir d'une instance donnée étaient totalement disjointes. Par exemple, pour le tri fusion, on divise le tableau à trier en deux parties, que l'on peut traiter indépendamment. 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 [1] .

Lorsque l'on résout un problème par programmation dynamique, on doit calculer le tableau des différentes valeurs. On s'intéresse à deux méthodes de calcul :

1. 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.

— De bas en haut, en remplissant les cases itérativement avec des boucles ;

— De haut en bas, en modifiant l'algorithme récursif pour qu'il interagisse avec le tableau.

A Construction de bas en haut

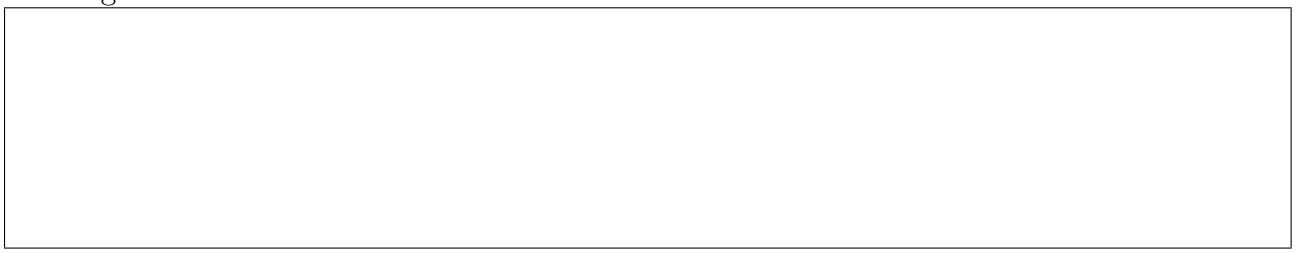
On reprend le problème du rendu de pièce. On se donne $a_1, \ldots, a_n \in \mathbb{N}^*$ des pièces et $M \in \mathbb{N}$ un montant à décomposer. Pour $x \in \mathbb{N}$ et $i \in [\![0, n]\!]$, on note $C(x, i)$ le nombre minimal de pièces de valeurs $a_1, a_2, \ldots, a_i$ que l'on peut utiliser 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 $a_1, \ldots, a_{i+1}$, alors soit on n'utilise pas a_{i+1}, ce qui veut dire qu'on n'utilise que $a_1, \ldots, a_i$, soit on utilise une fois a_{i+1}, auquel cas il nous reste $M - a_{i+1}$ à décomposer en utilisant $a_1, \ldots, a_{i+1}$. 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 $a_1, \ldots, a_n, M$ 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 $C(x, i)$. Remarquons que certaines cases 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 $T[x][i]$, il suffit de connaître la valeur des cases $T[x - a_i][i]$ et $T[x][i-1]$. 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 10 : 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[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 10. 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.

Notons qu'il n'y a pas forcément un unique ordre 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** sont respectées par l'ordre choisi.

B Construction de haut en bas

Dans la partie précédente, 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.

L'approche de **haut en bas** 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 11 : `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 $I_1, \dots I_p$;
  $S_1 \leftarrow$ `top_down`(I_1);
  $\dots$;
  $S_p \leftarrow$ `top_down`(I_p) ; Combiner $S_1, \dots, S_p$ en une solution S de I;
  $T[I] \leftarrow S$;
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 $p \in [\![0, W]\!]$.

Formule de récurrence 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 :

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

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 $3, 4, 5$: (Schéma)

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

Exercice 12. 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 ?

C 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 3. 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 1. 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 2. 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 13.

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 14. 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

On considère un problème de **décision** (i.e. attendant une réponse oui-non), où chaque instance $\mathcal{I}$ non-triviale peut se décomposer en sous-instances $\mathcal{I}_1, \dots \mathcal{I}_n$ telles que si **l'une** des $\mathcal{I}_k$ admet une solution, alors $\mathcal{I}$ en admet une. Ce type de problème apparaît très naturellement, et correspond à des situations où l'on doit faire une suite de choix, et où l'on cherche une suite valide de choix. Par exemple, si l'on considère le problème du Sudoku 4×4, étant donné une grille partielle G, et une case non-remplie de cette grille, on peut écrire 4 chiffres distincts dans cette case. Chaque chiffre $i \in [\![1, 4]\!]$ donne lieu à une nouvelle grille partielle G_i, et si l'une des G_i admet une solution, alors G en admet une aussi.

Les problèmes de ce type font naturellement apparaître une structure d'arbre, où les nœuds correspondent aux sous-instances, et les feuilles aux sous-instances triviales ne pouvant être décomposées. Savoir si un problème est résoluble revient alors à trouver une feuille correspondant à une instance positive dans l'arbre correspondant.

Prenons par exemple le jeu du Sudoku. L'algorithme naïf, consistant à essayer tous les choix, reviendrait en fait à un parcours en profondeur. On observe deux choses. D'une part, l'arbre est immense : en pratique, il est donc impensable d'effectuer une 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.

Un algorithme de **retour sur trace**, ou **backtracking**, est un algorithme qui explore l'arbre du problème, tout en disposant de critères permettant d'éliminer ou de valider immédiatement certaines sous-instances, sans les décomposer.

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

Algorithme 12 : 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) ← 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
```

Plus généralement, pour un problème $\mathcal{P}$, le schéma de backtracking est :

Algorithme 13 : `backtrack`($\mathcal{I}$)

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

```
si le critère permet d'éliminer ou de valider I directement alors
    retourner Oui ou Non (selon ce que dit le critère)
si I est une instance de base 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
```

L'algorithme de Quine est un exemple typique de retour sur trace : à chaque étape, on choisit une variable, et l'on essaie de la remplacer par vrai ou faux. Les règles de simplification donnent un critère permettant d'éliminer ou de valider certains cas.

Quelques remarques d'ordre pratique :

— On peut utiliser le critère d'élimination directement au moment de la division en sous-instances. Par exemple, pour le Sudoku, lorsque l'on remplit une case, plutôt que d'essayer chaque chiffre, et d'attendre l'étape suivante pour éliminer les grilles invalides, on va uniquement tester les chiffres potentiellement valides, i.e. n'apparaissant pas déjà dans la ligne, la colonne, ou le carré de la case.

— Comme on s'intéresse à des problèmes de décision, les algorithmes de résolution ne doivent renvoyer que Oui ou Non. En général, c'est les solutions qui nous intéresse et pas la question de leur existence. On implémente donc généralement les algorithmes de backtracking de façon à garder une **trace** des choix faits. Nous avons vu sur l'algorithme de Quine comment faire ça.

Exercice 15. On se donne un alphabet Σ 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 $t = u_1.u_2 \ldots u_n$. 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.

Question 1. Proposer un algorithme glouton, donner sa complexité et montrer qu'il n'est pas optimal.

Question 2. Proposer un algorithme de backtracking.

Question 3. On suppose maintenant que le texte peut admettre de très nombreuses décompositions, et on veut compter le nombre de telles décompositions. Proposer un algorithme de programmation dynamique.