ALGO1 – Programmation dynamique

François Schwarzentruber

19 novembre 2020

1 Plus longue sous-suite croissante

Définition 1 Soit une suite finie $a_1, \ldots, a_n$.
Une sous-suite croissante de $a_1, \ldots, a_n$ est une sous-suite $a_{i_1}, \ldots, a_{i_k}$ avec $i_k < i_{k+1}$ et $a_{i_k} \leq a_{i_{k+1}}$ pour tout k.

Définition 2 Le problème de calculer une plus longue sous-suite croissante est :
— Entrée : une suite finie $a_1, \ldots, a_n$;
— Sortie : une sous-suite croissante de $a_1, \ldots, a_n$ de longueur maximale.

Exemple 3 Une plus longue sous-suite croissante de 5, 2, 3, 8, 6, 3, 9, 7 est 2, 3, 6, 9.

1.1 Se concentrer sur la quantité à optimiser

Définition 4 Le problème de calculer la longueur d'une plus longue sous-suite croissante est :
— Entrée : une suite finie $a_1, \ldots, a_n$;
— Sortie : la longueur d'une plus longue sous-suite croissante.

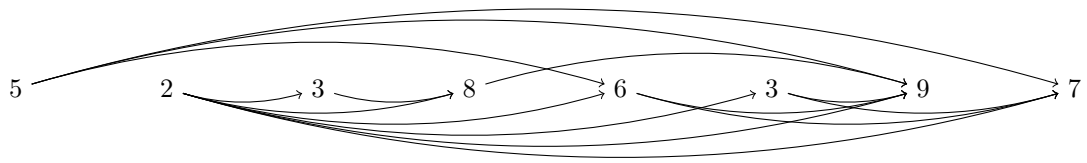
Exemple 5 La longueur d'une plus longue sous-suite croissante de 5, 2, 3, 8, 6, 3, 9, 7 est 4.

1.2 Définir des sous-problèmes

Définition 6 (sous-problèmes) Soit une suite finie $a_1, \ldots, a_n$.
Soit $L(j)$ la longueur d'une sous-suite croissante maximale, qui termine par l'élément en position j.

Proposition 7 La longueur d'une plus longue sous-suite croissante est $\max_{j=1,..,n} L(j)$.

1.3 Trouver une relation de récurrence



Proposition 8 (relation de récurrence) $L(j) = \max(L(i) \mid i < j \text{ et } a_i \leq a_j) + 1$.

Démonstration. Soit $\mathcal{S}(j)$ l'ensemble des sous-suites croissantes maximales finissant par a_j.

$\boxed{\geq}$ Si pour tout $i < j$, $a_i > a_j$, alors $\mathcal{S}(j) = \{(a_j)\}$ et $L(j) = 1 = 0 + 1$.

Sinon, soit $i_0 < j$ avec $a_{i_0} \leq a_j$ tel que $L(i_0) = \max(L(i) \mid i < j \text{ et } a_i \leq a_j)$. Considérons $s \in \mathcal{S}(i_0)$. La longueur de s est $L(i_0)$. Si on la complète avec a_j, on obtient une sous-suite croissante finissant par a_j, de longueur $L(i_0)+1$. Donc $L(i_0) + 1 \leq L(j)$.

$\boxed{\leq}$ Soit $s \in \mathcal{S}(j)$, de longueur $L(j)$. Si s est de longueur 1, alors l'inégalité $\leq$ est triviale.

Sinon, $s = s'a_j$ où s' sous-suite croissante finissant par un certain i_0 avec $i_0 < j$ et $a_{i_0} \leq a_j$. On a $s' \in \mathcal{S}(i_0)$ car sinon, on construit $s''a_j$ avec $s'' \in \mathcal{S}(i_0)$, qui est de longueur strictement plus grande que s. Contradiction. Ainsi, $L(j) = L(i_0) + 1 \leq \max(L(i) \mid i < j \text{ et } a_i \leq a_j) + 1$.

■

Programmation dynamique.
— Il y a un ordre sur les sous-problèmes.
— Une relation de récurrence explique comment résoudre un sous-problème à partir des problèmes plus petits.

1.4 Algorithme itératif

entrée : une suite finie de nombres $a_1, \dots, a_n$
sortie : la longueur d'une plus longue sous-croissante de $a_1, \dots, a_n$

fonction longueurPLSSC($a_1, \dots, a_n$)
  Soit $L[1, \dots, n]$ un tableau
  pour $j = 1, 2, \dots, n$ **faire**
    $L[j] := \max(L(i) \mid i < j \text{ et } a_i \leq a_j) + 1$
  retourner $max_j(L[j])$

1.5 Construire une sous-suite croissante

On ajoute un tableau π tel que, à la fin, $\pi[j]$ est le prédécesseur de a_j dans une plus longue sous-suite croissante où le dernier élément est a_j ou alors est $\bullet$ si la plus longue sous-suite croissante finissant par a_j est (a_j) elle même.

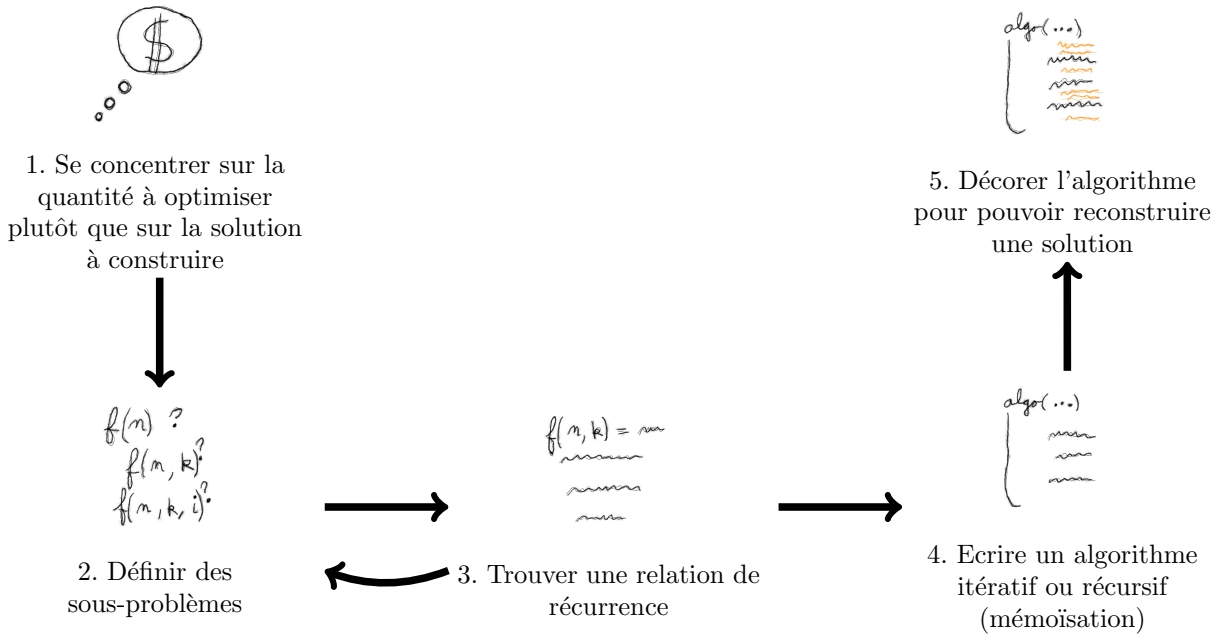
entrée : une suite finie de nombres $a_1, \dots, a_n$
sortie : une représentation d'une plus longue sous-croissante de $a_1, \dots, a_n$

fonction PLSSC($a_1, \dots, a_n$)
  Soit $L[1, \dots, n]$ un tableau
  pour $j = 1, 2, \dots, n$ **faire**
    $L[j] := \max(L(i) \mid i < j \text{ et } a_i \leq a_j) + 1$
    S'il existe $i_0 < j$ tel que $L[i_0] = \max(L[i] \mid i < j \text{ et } a_i \leq a_j)$ alors
      $\pi[j] := i_0$
    sinon
      $\pi[j] := \bullet$
  retourner $argmax_j(L[j]), \pi$

Proposition 9 Complexité temporelle : $\mathcal{O}(n^2)$.

Proposition 10 Complexité spatiale : $\mathcal{O}(n)$.

Conception d'algorithmes en programmation dynamique



2 Distance d'édition

Définition 11 La distance d'édition entre deux mots x et y est le nombre minimale d'opération à effectuer pour passer de x à y où les opérations sont :
- Ajouter une lettre
- Supprimer une lettre
- Changer une lettre

Exemple 12 *alkaarthe* → *alkarthe* → *algarthe* → *algorthe* → *algorithme*

a	l	k	a	a	r	t	h	–	e
a	l	g	o	–	r	t	h	m	e

La distance d'édition entre *alkaarthe* et *algorithme* est 5.

Proposition 13 La distance d'édition est une distance.

Définition 14 (problème de trouver une suite de longueur minimale d'opérations)
- entrée : deux mots x et y
- sortie : une suite de longueur minimale d'opérations à effectuer pour passer de x à y

2.1 Se concentrer sur la quantité à optimiser

Définition 15 (problème de la distance d'édition)
- entrée : deux mots x et y
- sortie : la distance d'édition entre x à y

2.2 Sous-problèmes

Définition 16 (sous-problèmes) $E(i,j) :=$ la distance d'édition de $x[1..i]$ à $y[1..j]$.

Exemple 17 Le sous-problème $E(3,6)$ consiste à calculer la distance d'édition de $x[1..3]$ et $y[1..6]$.

a	l	k	a	a	r	t	h	e	
a	l	g	o	r	i	t	h	m	e

2.3 Trouver une relation de récurrence

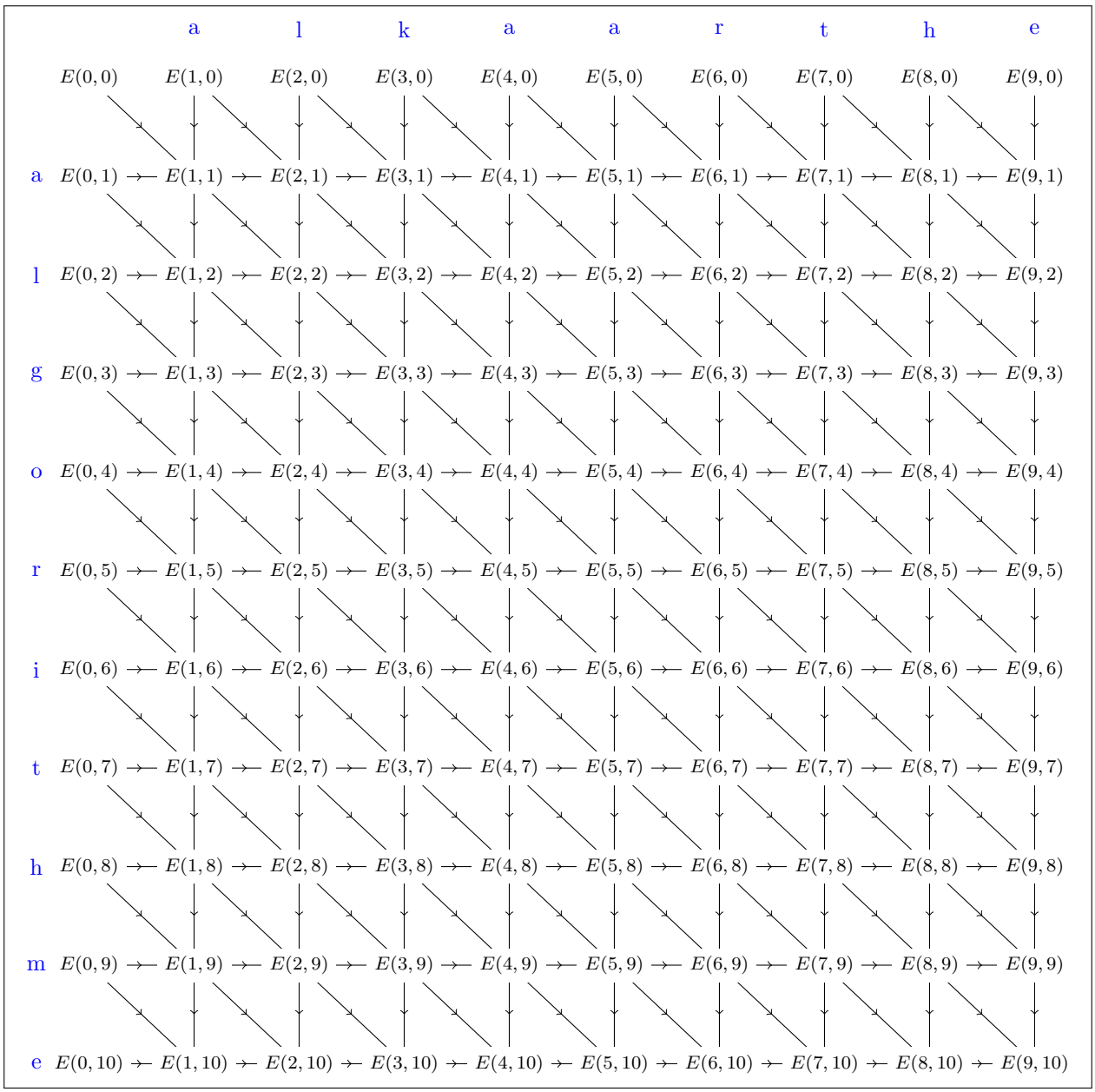
Un	alignement optimal	de $x[1..i]$ et $y[1..j]$		est de la forme :
Cas 1 :	alignement optimal	de $x[1..i-1]$ et $y[1..j]$	$x[i]$ –	
Cas 2 :	alignement optimal	de $x[1..i]$ et $y[1..j-1]$	– $y[j]$	
Cas 3 :	alignement optimal	de $x[1..i-1]$ et $y[1..j-1]$	$x[i]$ $y[j]$	

Proposition 18 (relation de récurrence)

$E(i,0) = i \qquad E(0,j) = j$

$$E(i,j) := \min\Big(E(i-1,j)+1, \quad E(i,j-1)+1, \quad E(i-1,j-1)+\mathsf{diff}(i,j) \Big)$$

où $\mathsf{diff}(i,j) = \begin{cases} 1 & \text{si } x[i] \neq y[1] \\ 0 & \text{sinon} \end{cases}$.



entrée : deux mots x et y
sortie : la distance d'édition entre x et y

fonction distanceEdition($x[1..n], y[1..m]$)
  pour $i = 1, 2, \ldots, n$ **faire**
    $E(i, 0) = i$

  pour $j = 1, 2, \ldots, m$ **faire**
    $E(j, 0) = j$

  pour $i = 1, 2, \ldots, n$ **faire**
    pour $j = 1, 2, \ldots, m$ **faire**
      $E(i,j) := \min\Big(E(i-1,j)+1, \quad E(i,j-1)+1, \quad E(i-1,-1)+\mathsf{diff}(i,j) \Big)$

  retourner $E(n, m)$

Proposition 19 Complexité temporelle $\mathcal{O}(nm)$.

Proposition 20 Complexité spatiale $\mathcal{O}(nm)$.

3 Plus courts chemins dans un graphe acyclique

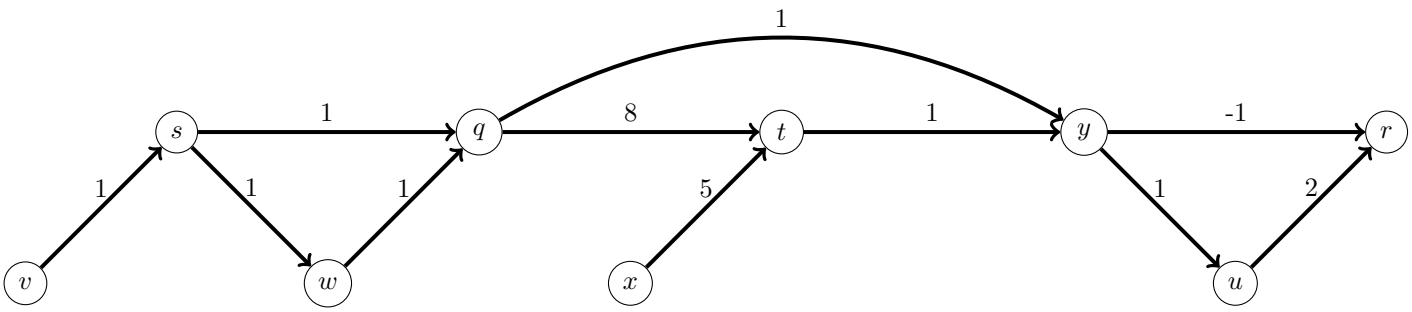
Définition 21 (Problème des plus courts chemins depuis une source dans un graphe acyclique)
— entrée : un graphe orienté acylique pondéré $G = (S, A, poids)$, un sommet source $s \in S$;
— sortie : les distances depuis s, i.e. les poids des plus courts chemins depuis s.

Définition 22 (Sous-problèmes) $d(u)$ = distance de s à u.

Proposition 23 (Relation de récurrence) $d(v) := \min(d(u) + poids(u, v) \mid u \to v)$.

```
pre : Un graphe G orienté pondéré acyclique, une source s
post : la distance de s à t pour tout t ∈ S
fonction distances(G, s)
    initialiser d[·] avec des ∞
    d[s] := 0
    tri topologique sur G
    pour v sommet dans un ordre linéaire faire
        d[v] := min(d[u] + poids(u, v) | u → v)
    retourner d[·]
```

Exemple 24



Proposition 25 Complexité temporelle en $\mathcal{O}(S + A)$.

Proposition 26 Complexité spatiale en $\mathcal{O}(S + A)$.

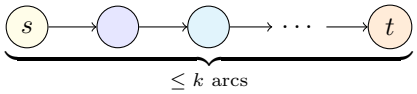
4 Algorithme de Bellman-Ford

Proposition 27 Soit G un graphe connexe pondéré (poids négatifs autorisés) sans cycles négatifs et s, t deux sommets. Il existe un plus court chemin de s à t.

Démonstration. $L_l = \{l(c), c \text{ chemin élémentaire de } s \text{ à } t\}$ est fini donc admet un minimum. ■

Définition 28 (problème des plus courts chemins dans un graphe depuis une source)
- Entrée : un graphe $G = (S, A, poids)$ orienté pondéré (poids négatifs autorisés), sans cycles négatifs ; un sommet s de G ;
- Sortie : des plus court chemins depuis la source s vers tous les autres sommets.

Définition 29 (sous-problèmes) $\delta(k, t)$ le poids d'un plus court chemin de s à t qui a au plus k arcs.



Proposition 30 (relation de récurrence)
$\delta(0, t) = 0$ si $t = s$ et $+\infty$ sinon.
si $k \geq 1$, $\delta(k, t) = \min\{\delta(k-1, t), \min_{u, u \to t}(\delta(k-1, u) + poids(u, t))$

```
pre : Un graphe G sans cycles négatifs, un sommet s
post : la distance de s à t pour tout t ∈ S
fonction bellmanVersionProvisoire(G, s)
    pour t ∈ S faire d[0,t] := +∞
    d[0,s] := 0
    pour k = 1 à |S| − 1 faire
        pour t ∈ S faire
            d[k,t] := d[k−1,t]
            pour u avec u → t faire
                d[k,t] := min(d[k,t], d[k−1,u] + poids(u,t))
    retourner d[|S| − 1, ·]
```

```
pre : un graphe G sans cycles négatifs, une source s
post : la distance de s à t pour tout t ∈ S
fonction bellman(G, s)
    pour t ∈ S faire d[t] := +∞
    d[s] := 0
    pour k = 1 à |S| − 1 faire
        pour arc u → t faire
            d[t] := min(d[u] + poids(u,t), d[t])
    retourner d
```

Proposition 31 La complexité temporelle de Bellman est $\mathcal{O}(|S| + |A|)$.

Proposition 32 La complexité spatiale est en ~~$\mathcal{O}(S^2)$~~ $\mathcal{O}(S)$.

Théorème 33 (correction) S'il n'y a pas de cycles négatifs, alors $d[t] = \delta(t)$ pour tout $t \in S$.

Démonstration.
On montre les invariants $\delta(t) \leqslant d[t]$ et à la fin du k-ème passage, $d[t] \leqslant \delta(k, t)$.
À la fin, on a donc $\delta(t) \leqslant d[t] \leqslant \delta(|S| - 1, t)$. ■

Théorème 34 (détection de cycles) Il y a un cycle négatif ssi le tableau d est modifié si on exécute une itération supplémentaire des lignes 5 et 6.

Démonstration. $\boxed{\Leftarrow}$ S'il n'y a pas de cycles négatifs, $\delta(t)$ est bien définie. Comme $d[t] = \delta(t)$ et que $\delta(t) \leq d[t]$, d ne peut pas être avoir été modifié.
$\boxed{\Rightarrow}$ Réciproquement, supposons que d ne soit pas modifié. Soit $v_0, \cdots, v_k = v_0$ un cycle. Montrons que le poids du cycle $\sum_{i=0}^{k-1} poids(v_i, v_{i+1})$ est positif.
Comme les valeurs de $d[.]$ ne sont pas modifiées, on a $d[t] \leq d[u] + poids(u, t)$ pour tout arc $u \to t$. Ainsi :

$$\begin{aligned} d[v_0] &\leqslant d[v_{k-1}] + poids(v_{k-1}, v_0) \\ &\leqslant d[v_{k-2}] + poids(v_{k-1}, v_0) + poids(v_{k-2}, v_{k-1}) \\ &\cdots \\ &\leqslant d[v_0] + \sum_{i=0}^{k-1} poids(v_i, v_{i+1}) \end{aligned}$$

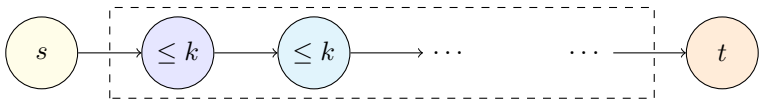
Donc $\sum_{i=0}^{k-1} poids(v_i, v_{i+1}) \geq 0$. ■

5 Algorithme de Floyd-Warshall

Définition 35 (problème des plus courts chemins dans un graphe depuis tout somme vers tout sommet)

— Entrée : un graphe $G = (S, A, poids)$ orienté pondéré (poids négatifs autorisés), sans cycles négatifs ;
— Sortie : les distances entre toute paire de sommets.

Définition 36 (sous-problèmes) On suppose ici que l'ensemble des sommets est $\{1, \dots, n\}$.
$\delta(k, x, y)$ = la longueur d'un plus court chemin de x à y où les sommets intermédiaires sont inférieurs à k.



Proposition 37 (relation de récurrence)

$\delta(0, x, y) = poids(x, y)$ si $x \neq y$ et 0 sinon
si $k \geq 1$, $\delta(k, x, y) = \min\{\delta(k-1, x, y),\ \delta(k-1, x, k) + \delta(k-1, k, y)\}$

pre : Un graphe G sans cycles négatifs post : La distance minimale de s à t pour tout $t \in S$
fonction floyd-Warshall(G)
    pour $x, y \in S^2$ **faire**
        si $x = y$ **alors** $d[x, y] := 0$ **sinon** $d[x, y] := poids(x, y)$

    pour $k = 1$ à n **faire**
        pour $x \in S$ **faire**
            pour $y \in S$ **faire**
                $d[x, y] := \min\{d[x, y], d[x, k] + d[k, y]\}$

    retourner d

Proposition 38 La complexité temporelle de Floyd-Warshall est en $\mathcal{O}(|S|^3)$.

Proposition 39 La complexité spatiale est en $\mathcal{O}(|S|^2)$.

6 Sac à dos

Exemple 40 But : remplir un sac de poids en maximisant la valeur et sans dépasser un poids maximal $P = 10$.

Objet i	Poids p_i	Valeur v_i
1	6	30€
2	3	14€
3	4	16€
4	2	9€

Solution avec répétition : objets 1, 4, 4. Solution sans répétition : objets 1 et 3.

Définition 41 (Problème du sac à dos avec répétition (sans répétition))
- Entrée : une collection d'objets $(p_i, v_i)_{i=1..n}$, un poids maximal P ;
- Sortie : la valeur maximale de $\sum_{i=1}^{n} n_i v_i$ avec $\sum_{i=1}^{n} n_i p_i \leq P$ et n_i dans $\mathbb{N}$ (dans $\{0, 1\}$)

6.1 Avec répétition

Définition 42 (sous-problèmes)
$V(p)$ = la valeur maximale atteignable avec un sac à dos de capacité maximale p.

Proposition 43 Pour tout p, $\quad V(p) := \max\{V(p - p_i) + v_i \mid i = 1, \ldots, n \text{ et } p_i \leq p\}$

```
fonction sacados((p_i, v_i)_{i=1..n}, P )
    pour p := 0 à P faire
        V(p) := max{V(p − p_i) + v_i | i = 1, ..., n et p_i ≤ p}
    retourner V(P)
```

Proposition 44 Complexité temporelle en $\mathcal{O}(nP)$.

Proposition 45 Complexité spatiale en $\mathcal{O}(P)$.

Mémoïzation. écrit un algorithme récursif mais en stockant les valeurs déjà calculées.

```
V table de hachage vide
fonction sacadosMemo(p)
    si la valeur V(p) est définie alors
        retourner V(p)
    insérer la valeur max{sacadosMemo(p − p_i) + v_i | i ∈ {1, ..., n} and p_i ≤ p} dans V(p)
    retourner V(p)
```

6.2 Sans répétition

Définition 46 (sous-problèmes)
$V(p, j)$ = la valeur maximale atteignable avec un sac à dos de capacité maximale p et que des objets parmi $1, \ldots, j$.

Définition 47 (relation de récurrence)

$$V(p, 0) = 0$$
$$V(p, j) := \begin{cases} \max(V(p - p_j, j - 1) + v_j, V(p, j - 1)) & \text{si } p_j \leq p \\ V(p, j - 1) & \text{sinon} \end{cases}$$

```
fonction sacadossansrepetition((p_i, v_i)_{i=1..n}, P )
    pour j = 1..n faire
        V(0, j) := 0
    pour p = 0..P faire
        V(p, 0) := 0
    pour j = 1..n faire
        pour p := 0 à P faire
            V(p, j) := { max(V(p − p_j, j − 1) + v_j, V(p, j − 1))   si p_j ≤ p
                       { V(p, j − 1)                                 sinon
    retourner V(P, n)
```

Proposition 48 Complexité temporelle en $\mathcal{O}(nP)$.

Proposition 49 Complexité spatiale en $\mathcal{O}(nP)$.

Notes bibliographiques

Sur la genèse de la programmation dynamique, on peut se rapporter à [Dre02]. La progression

$$\text{glouton} \rightarrow \text{programmation dynamique} \rightarrow \text{flots} \rightarrow \text{programmation linéaire}$$

est présentée dans [KT06]. La programmation dynamique est très utilisée en bioinformatique. L'algorithme de Smith-Waterman [SW+81] un algorithme d'alignement de séquences, qui utilise la même idée que la distance d'édition. Il y a toujours des papiers de recherche avec de la programmation dynamique comme par exemple [KPG+18], dont un des coauteurs est à Lille.

Références

[Dre02] Stuart Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research*, 50(1) :48–51, 2002.

[KPG+18] Anna Kuosmanen, Topi Paavilainen, Travis Gagie, Rayan Chikhi, Alexandru I. Tomescu, and Veli Mäkinen. Using minimum path cover to boost dynamic programming on dags : Co-linear chaining extended. In Benjamin J. Raphael, editor, *Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings*, volume 10812 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2018.

[KT06] Jon Kleinberg and Eva Tardos. *Algorithm design*. Pearson Education India, 2006.

[SW+81] Temple F Smith, Michael S Waterman, et al. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1) :195–197, 1981.