

# INFORMATIQUE

2024 - 2025

## Algorithmes sur les Graphes et Combinatoire

Laurent PHILIPPE

SUP<sup>F</sup><sub>C</sub>

---

# Algorithmique sur les Graphes et Combinatoire

MASTERS INFORMATIQUE DVL / I2A / ISL  
1ÈRE ANNÉE

---

## Chapitre III - Exercices - Programmation dynamique

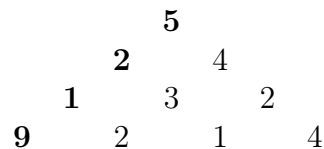


Centre de télé enseignement  
Filière Informatique  
Domaine Universitaire de la Bouloie  
25030 Besançon Cedex (France)

# 1 Construction de solutions de programmation dynamique

## Exercice 1 : Pyramide de nombres

Un problème simple et classique en programmation dynamique est la recherche du meilleur chemin dans une pyramide de nombres telle que celle donnée ci-dessous :



Le chemin part de la pointe de la pyramide et descend étage par étage. A chaque étage le chemin cumule les points des cases par lesquelles il passe et l'objectif est de trouver le chemin qui obtient le meilleur score après avoir traversé toute la pyramide.

Sur la figure le chemin donné en gras obtient le meilleur score (ici 17). Nous pouvons aussi constater que le fait de choisir la plus grande valeur à chaque étape ne permet pas de trouver la solution optimale puisqu'ici cette stratégie ne permet d'obtenir qu'un score de 14 avec le chemin 5-4-3-2.

La solution naïve de résolution de ce problème consiste à explorer, depuis le sommet, l'ensemble des chemins possibles. Il y en a 8 dans notre exemple mais le nombre croît très rapidement avec la hauteur de la pyramide : à chaque nouvel étage chacun des chemins calculés dispose de deux possibilités, donc crée deux nouveaux chemins. Le nombre total de chemin est alors de l'ordre de  $2^n$ . Nous nous proposons donc de trouver une solution de programmation dynamique à ce problème.

Le principe de la programmation dynamique repose sur la décomposition d'une solution en sous-solutions. Il faut alors montrer que la solution optimale peut être calculée à partir des sous-problèmes.

**Question 1.1** : *Donner la structure de la solution optimale en fonction des sous-problèmes.*

**Question 1.2** : *Donner une expression récursive de la solution.*

La solution récursive, si elle est calculé directement, engendre un grand nombre de calcul.

**Question 1.3** : *Comment diminuer le nombre de calculs ?*

**Question 1.4** : *Donner un algorithme de programmation dynamique qui permet de trouver la solution optimale de manière efficace.*

L'algorithme donné précédemment permet de trouver la plus grande valeur de chemin mais ne donne pas le chemin lui-même.

**Question 1.5** : *Comment, à partir des calculs réalisés, retrouver le chemin optimal ? Ce chemin contiendra la liste des indices dans leur ligne des points traversés, en commençant par la base de la pyramide. Dans le cas de l'exemple, le résultat sera 1,1,1,1 puisque le meilleur chemin n'utilise que les premières valeurs de chaque ligne.*

Pour se convaincre de l'intérêt de la programmation dynamique il suffit de tester les temps d'exécution pour les deux algorithmes.

**Question 1.6 :** *Écrire une classe `Pyramid` qui lit un fichier texte contenant une pyramide de nombres et deux méthodes, `recursiveAlgo` et `progdynAlgo`, qui implémentent les algorithmes conçus aux questions précédentes.*

Vous pourrez utiliser le fichier `pyramide1.txt` donné sur moodle pour effectuer vos tests.

**Question 1.7 :** *En utilisant les fichiers `pyramide15.txt` et `pyramide100.txt`, calculer les temps d'exécution pour chacun des algorithmes.*

**Question 1.8 :** *À partir de quelle taille de pyramide le calcul récursif devient-il trop long ?*

## **Exercice 2 : Sac à dos**

Nous avons introduit dans le cours le problème du sac à dos, un classique en programmation dynamique. Dans ce problème nous disposons d'un sac supportant un poids maximum que nous voulons remplir avec des objets caractérisés par un poids et une valeur. Chaque objet n'existe qu'en un seul exemplaire.

Le problème du sac à dos consiste à maximiser la valeur totale des objets emportés en respectant la contrainte de poids liée à la capacité du sac. Ce problème a été montré NP-Complet et, de ce fait, il n'existe pas d'algorithme polynomial capable de trouver la solution optimale. La solution systématique, qui consiste à explorer toutes les combinaisons possibles d'objets est combinatoire. Elle revient à construire un arbre où chaque sommet représente le choix possible d'un objet et possède deux branches, une où l'objet est choisi et l'autre où il ne l'est pas. De ce fait le nombre de solutions est de l'ordre de  $2^n$ .

L'objectif de cet exercice est de réaliser une programmation dynamique permettant de trouver la solution optimale de ce problème.

Pour illustrer le problème posé nous commençons par un exemple où nous disposons d'un sac de 10 kg. Les objets que nous pouvons emporter sont donnés dans la table 3.

article	1	2	3	4
valeur	3	5	8	10
poids	2	3	5	6

TABLE 1 – Tableau des valeurs et des poids des objets

**Question 2.1 :** *Quelles sont les solutions possibles et leur valeur pour cet exemple.*

Nous cherchons maintenant une solution de programmation dynamique au problème général d'un sac à dos de poids maximum  $P$  que nous voulons remplir à partir d'un ensemble  $O$  d'objets  $o_i$ . Un objet est caractérisé par un poids  $p_i$  et une valeur  $v_i$ .

Comme nous l'avons vu dans le cours une approche consiste à décomposer le problème initial en considérant qu'un des objets  $o_i$  est mis ou non dans le sac. Dans ce cas la valeur optimale du sac à dos est calculée à partir :

1. du cas où l'objet fait partie de la solution optimale, donc du même problème de sac à dos mais où on considère que l'objet est pris et il nous reste à résoudre le

sous-cas avec un poids maximal  $P = P - p_i$  diminué du poids de l'objet  $o_i$  et un ensemble d'objets sans  $o_i$  :  $O = O - o_i$ ,

2. du cas où l'objet ne fait pas partie de la solution optimale, donc du même problème de sac à dos où le poids maximal reste à  $P$  mais l'ensemble ne contient plus l'objet  $o_i$  :  $O = O - o_i$ .

**Question 2.2 :** Caractériser la structure d'une solution optimale.

**Question 2.3 :** Donner la relation de récurrence permettant d'ajouter un objet dans le sac à dos compte tenu des objets déjà rangés dans le sac.

**Question 2.4 :** Dans quelle structure est-il possible de stocker les résultats intermédiaires afin de ne jamais calculer deux fois la même chose pour cette résolution ascendante ? Comment la remplir ?

**Question 2.5 :** Écrire l'algorithme de résolution du problème entier grâce à la programmation dynamique.

**Question 2.6 :** Donner la complexité de cet algorithme.

**Question 2.7 :** Programmer l'algorithme récursif et l'algorithme de programmation dynamique en Java pour l'exemple initial.

**Question 2.8 :** A partir de la table des sous-problèmes, comment retrouver le contenu du sac à dos ?

### Exercice 3 : Chaîne de montage

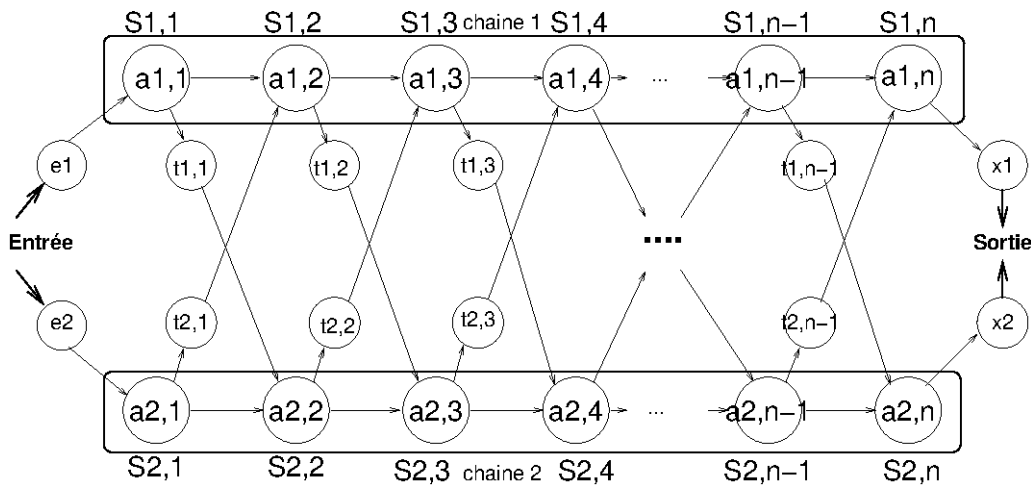


FIGURE 1 – L'atelier de fabrication

Une firme fabrique des automobiles dans un atelier qui a deux chaînes de montage (voir la figure 1). Un châssis arrive sur l'une des chaînes, puis passe par un certain nombre de postes où on lui ajoute des pièces; une fois terminée, l'auto sort par l'autre extrémité de la chaîne. Chaque chaîne de montage a  $n$  stations, numérotées  $j = 1, 2, \dots, n$ . On représente le  $j^{ieme}$  poste de la chaîne  $i$  (avec  $i = 1$  ou  $i = 2$ ) par  $S_{i,j}$ . Le  $j^{ieme}$  ( $S_{1,j}$ ) poste de la chaîne 1 fait le même travail que le  $j^{ieme}$  ( $S_{2,j}$ ) poste de la chaîne 2.

Les postes ont été installés à des époques différentes et avec des technologies différentes ;

ainsi, le temps de montage varie d'un poste à l'autre, même quand il s'agit de postes au fonctionnement identique mais situés sur des chaînes différentes.

Le temps de montage au poste  $S_{i,j}$  est  $a_{i,j}$ . Comme le montre la figure 1, un châssis arrive au poste 1 de l'une des chaînes, puis passe de poste en poste. On a de plus un temps d'arrivée  $e_i$  pour le châssis qui entre sur la chaîne  $i$ , et un temps de sortie  $x_i$  pour l'auto achevée qui sort de la chaîne  $i$ .

Normalement, une fois qu'un châssis arrive sur une chaîne de montage, il ne circule que sur cette chaîne et le temps de passage d'un poste à l'autre sur une même chaîne est négligeable. En cas d'urgence, toutefois, il se peut que l'on veuille accélérer le délai de fabrication d'une automobile. En pareil cas, le châssis transite toujours par les  $n$  postes dans l'ordre, mais le chef d'atelier peut faire passer une auto partiellement construite d'une chaîne à l'autre, et ce après chaque poste. Le temps de transfert d'un châssis depuis la chaîne  $i$  et après le poste  $S_{i,j}$  est  $t_{i,j}$ , avec  $i = 1, 2$  et  $j = 1, 2, \dots, n - 1$  (car après le  $n^{\text{ième}}$  poste c'est fini).

Le problème consiste à déterminer quels sont les postes à sélectionner sur la chaîne 1 et sur la chaîne 2 pour minimiser le délai de transit d'une auto à travers l'atelier.

On note  $f_i[j]$  le délai le plus court possible avec lequel le châssis va du point de départ à la sortie du poste  $S_{i,j}$ . On note  $f^*$  le délai le plus court par lequel un châssis traverse tout l'atelier. De plus, on note  $l_i[j]$  le numéro de la chaîne du poste  $j - 1$  appartenant au chemin optimal arrivant en  $S_{i,j}$  et  $l^*$  le numéro de la chaîne par laquelle sortent les voitures ayant empruntées le chemin optimal.

**Question 3.1 :** *Caractérisation d'une solution optimale*

**Question 3.2 :** *Exprimer  $f^*$  en fonction de  $f_1[n]$  et de  $f_2[n]$ .*

**Question 3.3 :** *Donner les valeurs de  $f_1[1]$  et de  $f_2[1]$ .*

**Question 3.4 :** *Caractériser par une relation de récurrence les valeurs de  $f_1[j]$  et de  $f_2[j]$ .*

**Question 3.5 :** *Écriture des algorithmes*

**Question 3.6 :** *Proposer un algorithme récursif et un algorithme itératif qui calcule les valeurs de  $f_1[j]$  et  $f_2[j]$  ainsi que les valeurs de  $l_1[j]$  et  $l_2[j]$  pour  $j = 1, 2, \dots, n$ .*

**Question 3.7 :** *Proposer un algorithme d'affichage du chemin optimal, d'abord dans le sens inverse (algorithme itératif simple) puis dans le bon sens, en utilisant le tableau  $l_i[j]$ .*

**Question 3.8 :** *Programmation des solutions*

**Question 3.9 :** *Écrire un programme C qui permette la mise en œuvre des solutions récursives et itératives présentées dans l'exercice précédent.*

**Question 3.10 :** *Comparer les exécutions des deux solutions. Jusqu'à quelle taille de chaîne de montage êtes vous allé avec la version récursive ? Quel temps cela a pris pour le calcul ? Quelle a été la durée du même calcul avec la version itérative issue de la programmation dynamique ? Conclure.*

## Correction Exercice 1 : Pyramide de nombres

**Solution question 1.1 :** *Donner la structure de la solution optimale en fonction des sous-problèmes.*

Les chemins partent de la pointe de la pyramide et arrivent à la base, le niveau  $n$ , sans retour. La solution optimale est le chemin qui obtient le meilleur score. Le problème se décompose donc en deux parties : trouver le meilleur score des points de la base et trouver le meilleur score en un point de la pyramide.

Pour trouver le meilleur score des points de la base, il suffit de calculer le maximum du score obtenu en chaque point  $p_{i,n}$  (avec  $i = 1, \dots, n$ ) de la base. Si nous notons  $s(p_{i,j})$  le score obtenu en  $p_{i,j}$ , alors le meilleur score  $S^*$  de la pyramide à l'étage  $n$  est :

$$S^* = \max_{i=1, \dots, n} (s(p_{i,n}))$$

Nous devons donc maintenant calculer le score en un point de la base. Le calcul du meilleur score d'un point de la base est identique au calcul du score en un point quelconque de la pyramide.

Pour trouver le meilleur score en un point de la pyramide, il faut évaluer les chemins qui y arrivent. Pour chacun des points  $p_{i,n}$ , ces chemins sont passés par l'étage  $n - 1$ . Plus précisément, ces chemins arrivent soit du point de droite, soit du point de gauche au dessus du point. Par exemple, sur l'étage 3 de la pyramide donnée en exemple, les chemins qui arrivent au point 3 proviennent tous de l'étage 2 et, soit de droite (le point de valeur 4), soit gauche (le point de valeur 2). Ainsi, le meilleur score obtenu en un point  $p_{i,n}$  de la pyramide est le maximum du score du point de l'étage supérieur à gauche et de l'étage supérieur à droite auquel on ajoute la valeur du point courant. Pour exprimer cela nous définissons trois fonctions :  $sg(p)$  qui donne le plus grand score obtenu par le point à gauche du point  $p$  à l'étage  $n - 1$ ,  $sd(p)$  qui donne le plus grand score obtenu par le point à droite du point  $p$  à l'étage  $n - 1$  et  $v(p)$  qui rend la valeur du point  $p$ . Ainsi nous pouvons définir  $s(p_{i,j})$ , le plus grand score obtenu au point  $p_{i,n}$  par :

$$s(p_{i,j}) = v(p_{i,j}) + \max(sg(p_{i,j}), sd(p_{i,j}))$$

À noter que les valeurs  $sg(p_{i,j})$  et  $sd(p_{i,j})$  expriment bien des valeurs de l'étage  $n - 1$ , et pas de l'étage  $n$ . Il s'agit donc bien de sous-problèmes : la dimension de la pyramide considérée est  $n - 1$ .

Nous pouvons maintenant définir la structure de la solution optimale en fonction des sous-problèmes :

$$S^* = \max_{i=1, \dots, n} (v(p_{i,n}) + \max(sg(p_{i,n}), sd(p_{i,n})))$$

**Solution question 1.2 :** *Donner une expression récursive de la solution.*

A partir de cette formulation la définition récursive est relativement simple puisque l'expression précédente inclut déjà une expression de la valeur en  $n$  à partir de la valeur en  $n - 1$ . Il nous suffit donc d'exprimer le premier niveau de la récurrence, celui pour lequel

la pyramide n'a qu'un étage. Lorsque la pyramide n'a qu'un étage, la valeur optimale est celle du seul point existant.

$$S^* = v(p_{1,1})$$

L'algorithme 1 donne une solution récursive à la résolution du problème de la pyramide de nombres. Il est décomposé entre la fonction *plusGrandChemin*( $l, c$ ) qui calcule la valeur maximale qui peut être atteinte en un point (colonne  $c$ , ligne  $l$ ) et le programme principal qui cherche la valeur maximale sur la ligne de base de la pyramide.

---

**Algorithme 1 :** Algorithme récursif pour le problème de la pyramide de nombres

---

**Données :**  $V$  : matrice contenant la liste de nombres

$n$  : nombre d'étages dans la matrice

**Résultat :**  $S^*$  : valeur du plus grand chemin

```

1 fonction plusGrandChemin( $l, c$ )
2 début
3   si  $c = 1 \wedge l = 1$  alors retourner  $V(1, 1)$ 
4   si  $c = 1$  alors retourner  $plusGrandChemin(l - 1, c) + V(l, c)$ 
5   si  $c = l$  alors retourner  $plusGrandChemin(l - 1, c - 1) + V(l, c)$ 
6   retourner
      max( $plusGrandChemin(l - 1, c - 1), plusGrandChemin(l - 1, c)$ ) +  $V(l, c)$ 
7 Programme principal
8 début
9    $S^* \leftarrow 0$ 
10  pour  $c$  de 1 à  $n$  faire
11     $S \leftarrow plusGrandChemin(n, c)$ 
12    si  $S > S^*$  alors  $S^* \leftarrow S$ 
```

---

Pour la réalisation de l'algorithme, un des problèmes auquel nous sommes confrontés est l'utilisation d'une structure de données pour stocker les valeurs contenues dans la pyramide. Nous choisissons de la stocker dans une matrice  $V$  dont nous n'utilisons que le triangle inférieur. Attention, nous supposons ici, pour la lisibilité de l'algorithme, que la matrice est indicée de 1 à  $n$ , cela ne sera pas le cas lorsque nous écrirons le programme.

**Solution question 1.3 :** *Comment diminuer le nombre de calculs ?*

Avec l'expression récursive nous avons une formulation du problème qui, plutôt que de chercher les chemins de manière descendante, les cherche de manière ascendante. Or il est aisé de voir que, pour un point de l'étage au dessus de l'étage courant, il se trouve une fois à droite et une fois à gauche dans les calculs de l'étage. De ce fait, nous pouvons diviser par deux le nombre de calculs à réaliser à chaque étage et obtenir une solution beaucoup plus efficace, à condition de mémoriser les résultats intermédiaires. Nous allons donc créer une nouvelle pyramide, de manière ascendante dans laquelle nous mémorisons, en chaque point la valeur maximale obtenue.

**Solution question 1.4 :** *Donner un algorithme de programmation dynamique qui permet de trouver la solution optimale de manière efficace.*



L'algorithme 2 donne une solution de programmation dynamique efficace.

---

**Algorithme 2 :** Programmation dynamique pour la pyramide de nombres

---

**Données :**  $V$  : matrice contenant la liste de nombres

$n$  : nombre d'étages dans la matrice

**Résultat :**  $S^*$  : valeur du plus grand chemin

```

1   $I$  : matrice contenant les résultats intermédiaires
2  début
3       $I(1, 1) = V(1, 1)$ 
4      pour  $l$  de 2 à  $n$  faire
5          /* Extrémités de la ligne */
6           $I(l, 1) = I(l - 1, 1) + V(l, 1)$ 
7           $I(l, l) = V(l - 1, l - 1) + V(l, l)$ 
8          pour  $c$  de 2 à  $l - 1$  faire
9               $I(l, c) = \max(I(l - 1, c - 1), I(l - 1, c)) + V(l, c)$ 
9       $S^* \leftarrow 0$ 
10     pour  $c$  de 1 à  $n$  faire
11         si  $I(n, c) > S^*$  alors  $S^* \leftarrow I(n, c)$ 

```

---

Comme pour l'algorithme récursif, la matrice  $V$  est utilisée pour stocker les valeurs de la pyramide. Les résultats intermédiaires sont stockés dans une matrice  $I$ , dont nous n'utilisons que le triangle inférieur.

**Solution question 1.5 :** *Comment, à partir des calculs réalisés, retrouver le chemin optimal ? Ce chemin contiendra la liste des indices dans leur ligne des points traversés, en commençant par la base de la pyramide. Dans le cas de l'exemple, le résultat sera 1,1,1,1 puisque le meilleur chemin n'utilise que les premières valeurs de chaque ligne.*

Le chemin obtenant le meilleur score peut être retrouvé à partir de la matrice des résultats intermédiaires. A partir du point de la base qui obtient le meilleur score, il est possible de remonter dans la matrice  $I$  en choisissant le meilleur des deux voisins (lorsqu'il y a deux voisins) et en mémorisant l'indice correspondant.

L'algorithme 3 permet de retrouver le chemin qui donne le meilleur score.

---

**Algorithme 3** : Meilleur chemin dans la pyramide de nombres

---

**Données** :  $I$  : matrice contenant les valeurs des plus grands chemins

$n$  : nombre d'étages dans la matrice

**Résultat** :  $C^*$  : chemin optimal

```
1 début
2    $best \leftarrow 1$ 
3   pour  $c$  de 2 à  $n$  faire
4     si  $I(n, c) > I(n, best)$  alors  $best \leftarrow c$ 
5    $C^* \leftarrow \{best\}$ 
6   pour  $l$  de  $n$  à 2 faire
7     si  $(best = 0) \vee (I(l-1, best-1) < I(l-1, best))$  alors
8        $C^* \leftarrow C^* \cup best$ 
9     sinon
10       $C^* \leftarrow C^* \cup best - 1$ 
11       $best \leftarrow best - 1$ 
```

---

**Solution question 1.6** : Écrire une classe `Pyramid` qui lit un fichier texte contenant une pyramide de nombres et deux méthodes, `recursiveAlgo` et `progdynAlgo`, qui implémentent les algorithmes conçus aux questions précédentes.

Le code source de la classe `Pyramid` est donné sur moodle.

**Solution question 1.7** : En utilisant les fichiers `pyramide15.txt` et `pyramide100.txt`, calculer les temps d'exécution pour chacun des algorithmes.

Le tableau 2 donne les temps en millisecondes de calcul sur différentes tailles de pyramides.

taille matrice	15	20	25	30	35	40	50
tps récursif	2	18	184	5522	205168	> 1h	> 1h
tps dynamique	0	0	0	0	0	0	0

TABLE 2 – Performance comparée entre la programmation récursive et la programmation dynamique

**Solution question 1.8** : À partir de quelle taille de pyramide le calcul récursif devient-il trop long ?

Sur ma machine de travail, équipée d'un processeur Intel Core i5-10310U CPU à 1.70GHz et avec un JDK Oracle 12, une pyramide avec 35 lignes prend 251847 millisecondes soit 3 minutes et 25 secondes pour exécuter la recherche récursive alors que le temps pris en programmation dynamique n'atteint pas la milliseconde ! Le programme dynamique met moins d'1 milliseconde pour traiter une pyramide de 100 lignes alors qu'il est impossible de réaliser ce traitement en récursif.

### Correction Exercice 2 : Sac à dos

**Solution question 2.1 :** *Quelles sont les solutions possibles et leur valeur pour cet exemple.*

Les solutions possibles sont les combinaisons d'objets qui ne dépassent pas le poids supporté par le sac à dos. Elle sont données dans la table 3.

article	1	2	3	4	poids	valeur
solution 1	x	-	-	-	2	3
solution 2	-	x	-	-	3	5
solution 3	-	-	x	-	5	8
solution 4	-	-	-	x	6	10
solution 5	x	x	-	-	5	8
solution 6	x	-	x	-	7	11
solution 7	x	-	-	x	8	13
solution 8	-	x	x	-	8	13
solution 9	-	x	-	x	9	15
<b>solution 10</b>	x	x	x	-	10	<b>16</b>

TABLE 3 – Tableau des valeurs et des poids des objets

Cette table permet de mettre en évidence la nécessité d'explorer les solutions pour trouver la plus grande valeur possible.

**Solution question 2.2 :** *Caractériser la structure d'une solution optimale.*

Le problème initial peut être décomposé en deux sous problèmes suivant que l'objet  $o_i$  est pris ou non.

Le premier des sous problèmes, celui où l'objet est chargé dans le sac à dos, est caractérisé par un sac à dos le poids maximal  $P - p_i$  et une liste d'objets  $O - o_i$ . Le second, celui où l'objet  $o_i$  n'est pas chargé dans le sac à dos, est caractérisé par un poids maximal  $P$  et une liste d'objets  $O - o_i$ . La sous-structure optimale de ce problème repose donc sur la plus grande des valeurs optimales de ces deux sous-problèmes.

**Solution question 2.3 :** *Donner la relation de récurrence permettant d'ajouter un objet dans le sac à dos compte tenu des objets déjà rangés dans le sac.*

Soit  $V(P, i)$  la valeur optimale d'un chargement de poids maximal  $P$  pour l'ensemble des  $i$  premiers objets (ceux-ci peuvent être sélectionnés ou pas). La résolution du problème entier du sac à dos consiste à résoudre  $V(P, n)$ . La relation de récurrence permettant de savoir si l'objet  $i$  peut être ajouté est la suivante :

$$V(P, i) = \max\{V(P, i - 1), V(P - p_i, i - 1) + v_i\}$$

**Solution question 2.4 :** *Dans quelle structure est il possible de stocker les résultats intermédiaires afin de ne jamais calculer deux fois la même chose pour cette résolution ascendante ? Comment la remplir ?*

L'idée est de conserver les valeurs des chargements optimaux pour chaque poids et pour des séries des objets 1 à  $i$ , avec  $i = 1 \dots n$ . La taille de la table ainsi calculée est  $P \times n$ .

Il est possible de réduire le nombre de colonnes en divisant tous les encombrements, y compris celui du sac, par le *pgcd* des encombrements des objets.

Comme la récurrence le montre, chaque valeur dépend des valeurs des chargements calculées pour des problèmes de taille plus petite. Le calcul de ces valeurs se fait donc de manière ascendante et non descendante afin de profiter des calculs précédents pour le calcul optimal courant. Le calcul descendant intuitivement déduit de la formule de récurrence introduit trop de combinatoire dans le calcul de  $V(P, i)$ . En effet beaucoup de calculs seraient faits de nombreuses fois. Le principe de la programmation dynamique est par contre de les éviter grâce à la tabulation des valeurs optimales des sous problèmes.

**Solution question 2.5 :** *Écrire l'algorithme de résolution du problème entier grâce à la programmation dynamique.*

L'algorithme 4 utilise une approche ascendante pour le remplissage du tableau des valeurs des chargements optimaux pour tous les sous problèmes.

---

**Algorithme 4 :** Programmation dynamique pour le problème du sac à dos

---

**Données :**  $v_i$  : valeur de l'objet  $i$  avec  $i = 1..n$

$p_i$  : poids de l'objet  $i$  avec  $i = 1..n$

$P$  : poids disponible dans le sac

**Résultat :**  $V$  : valeur optimale du sac à dos avec les  $n$  objets pour un poids  $P$

1 **début**

2      $p$  : poids courant pour lequel on cherche un chargement optimal

3      $i$  : numéro de l'objet courant (initialisé à 1)

4      $V(i, p)$  : valeur optimale pour le poids  $p$  avec des objets 1 à  $i$

      /\* Initialisation de la table

\*/

5     **pour**  $p$  de 0 à  $P$  **faire**  $V(0, p) \leftarrow 0$

6     **pour**  $i$  de 0 à  $n$  **faire**  $V(i, 0) \leftarrow 0$

7     **pour**  $i$  de 1 à  $n$  **faire**

8         **pour**  $p$  de 1 à  $P$  **faire**

9             **si**  $p \geq p_i$  **alors**

10                  $V(i, p) \leftarrow \max(V(i-1, p), V(i-1, p-p_i) + v_i)$

11             **sinon**  $V(i, p) \leftarrow V(i-1, p)$

12      $V \leftarrow V(n, P)$

---

**Solution question 2.6 :** *Donner la complexité de cet algorithme.*

La complexité de l'algorithme de programmation dynamique permettant la résolution de la variante entière du problème du sac à dos est  $O(n \times P)$ . En effet, pour le calcul de la valeur optimale du chargement du sac acceptant un poids  $P$  avec  $n$  objets, il faut remplir, pour tout poids entier de 1 à  $P$ , la valeur optimale du chargement obtenue avec la série des objets 1 à  $i$  pour tout  $i$  de 1 à  $n$ .

Dans le cas initial nous avons choisi un poids  $P$  qui est relativement petit. Le problème considéré nécessite donc une place mémoire, pour la table  $V$ , qui reste raisonnable. Mais si nous considérons un sac avec un poids supportable de l'ordre du millier de kilos avec un grand nombre d'objets alors la taille de la table  $V$  peut rapidement devenir

conséquence. Il est éventuellement possible alors de réduire d'espace des poids visités en divisant les poids par leur *pgcd* et en ne commençant l'exploration que depuis le plus petit des poids. La complexité dans ce cas est de  $O(n \times (\frac{P}{\text{pgcd}(p_{1..n})} - \min(p_{1..n})) \sim O(n \times P)$ .

**NB :** Attention cette complexité ne signifie en rien que la solution soit polynomiale. Le problème initial étant *NP-Complet*, trouver une solution polynomiale reviendrait à dire que  $P = NP$ . La question que nous sommes alors en droit de nous poser est pourquoi cette complexité n'est pas polynomiale? En fait, la théorie de la complexité étudie les problèmes par rapport à la taille des entrées et non par rapport à leur valeur. Ici la complexité est dite *pseudo-polynomiale*. Ceci signifie que le problème n'est pas *NP-Complet* au sens dur, mais au sens faible. En effet, pour des tailles de problème relativement petite, le nombre d'opérations est faible, donc la solution est calculable. Le terme *pseudo* vient du fait que dans la complexité dépend d'une *valeur* codée en binaire et non en unaire. Du point de vue de l'étude de la complexité de ce problème, le poids  $P$  du sac n'est pas linéaire mais exponentiel, car lorsque sa représentation croît linéairement, sa valeur croît de manière exponentielle. Par contre le nombre d'objets est une valeur unitaire dont la représentation est logarithmique. La taille du problème initial n'est pas la valeur des nombres qui composent la complexité, mais la taille de leur représentation. On comprend alors qu'en doublant la taille de la représentation du volume du sac, alors la valeur du nombre représenté croît de manière exponentielle. A l'inverse, lorsque le nombre d'objets croît linéairement, sa représentation en mémoire croît de manière logarithmique. Lorsqu'on considère un problème comme un tri avec une complexité en  $O(n^2)$ , cette complexité est bien polynomiale car ce que représente  $n$  est un nombre de cases (éléments unaires) et doubler la taille du problème revient donc bien à doubler le nombre de cases. Pour le tri,  $n$  n'est pas un nombre en tant que valeur, mais un nombre de cases. Pour plus de détail, il est possible de trouver des compléments d'information sur Wikipedia<sup>1</sup>.

**Solution question 2.7 :** *Programmer l'algorithme récursif et l'algorithme de programmation dynamique en Java pour l'exemple initial.*

Le code source du programme Java est donné dans le fichier KnapSac.java

**Solution question 2.8 :** *A partir de la table des sous-problèmes, comment retrouver le contenu du sac à dos?*

A l'issue de l'exécution de la programmation dynamique la table  $V[i][p]$  a les valeurs données par la table 4.

Le contenu du sac, en fonction de l'ensemble des objets  $o_i$  et pour un sac où  $P = 10$ , est obtenu en parcourant la table 4 à partir de la case (4,10). Si la valeur immédiatement au dessus est la même c'est que l'objet n'a pas été pris et on continue à partir de cette case puisque le poids disponible ne change pas. Si par contre la case immédiatement au dessus est différente c'est que l'objet a été pris et il faut se déplacer dans la ligne pour retrouver le volume diminué du poids de l'objet. On obtient l'ensemble des objets lorsqu'on arrive sur la ligne  $o_i = 0$ .

---

1. [http://en.wikipedia.org/wiki/Pseudo-polynomial\\_time](http://en.wikipedia.org/wiki/Pseudo-polynomial_time)

$\begin{array}{c} P \\ o_i \end{array}$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	5	5	8	8	8	8	8	8
3	0	0	3	5	5	8	8	11	13	13	16
4	0	0	3	5	5	8	10	11	13	15	16

TABLE 4 – Table des valeurs optimales du sac à dos

### Correction Exercice 3 : Chaîne de montage

**Solution question 3.1 :** Exprimer  $f^*$  en fonction de  $f_1[n]$  et de  $f_2[n]$ .

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$$

**Solution question 3.2 :** Donner les valeurs de  $f_1[1]$  et de  $f_2[1]$ .

$$f_i[1] : \begin{cases} f_1[1] &= e_1 + a_{1,1} \\ f_2[1] &= e_2 + a_{2,1} \end{cases}$$

**Solution question 3.3 :** Caractériser par une relation de récurrence les valeurs de  $f_1[j]$  et de  $f_2[j]$ .

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{si } j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{si } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{si } j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{si } j \geq 2 \end{cases}$$

**Solution question 3.4 :** Proposer un algorithme récursif et un algorithme itératif qui calculent les valeurs de  $f_1[j]$  et  $f_2[j]$  ainsi que les valeurs de  $l_1[j]$  et  $l_2[j]$  pour  $j = 1, 2, \dots, n$ .

L'algorithme récursif est divisé en deux parties. Il y a la partie correspondant au lancement du calcul donné par l'algorithme 5. Les fonctions de calcul des valeurs optimales sur chacune des chaînes sont données par l'algorithme 6.

L'algorithme itératif, résultat de la programmation dynamique grâce à une programmation ascendante est donné par l'algorithme 7. Il retrace le cheminement exact de la solution optimale à l'intérieur de la chaîne de montage.

**Solution question 3.5 :** Proposer un algorithme d'affichage du chemin optimal, d'abord dans le sens inverse (algorithme itératif simple) puis dans le bon sens, en utilisant le tableau  $l_i[j]$ .

L'algorithme 8 permet l'affichage d'un cheminement optimal.

**Solution question 3.6 :** Programmation des solutions

**Solution question 3.7 :** Écrire un programme C qui permette la mise en œuvre des

---

**Algorithme 5** : Recherche récursive du plus court cheminement dans les chaînes

---

**Données** :  $n$  : la taille de la chaîne de montage

$a_{i,j}$  : temps de montage à  $S_{i,j}$ ,  $i = 1..2, j = 1..n$

$e_1, e_2$  : temps d'arrivée sur les chaînes 1 et 2

$x_1, x_2$  : temps de sortie des chaînes 1 et 2

$t_{1,j}$  : temps de transfert des chaînes 1 à 2 entre  $S_{1,j}$  et  $S_{2,j+1}$ ,  $j = 1..n - 1$

$t_{2,j}$  : temps de transfert des chaînes 2 à 1 entre  $S_{2,j}$  et  $S_{1,j+1}$ ,  $j = 1..n - 1$

**Résultat** :  $l_{i,j}$  : les numéros de chaînes précédant l'arrivée en  $(i, j)$

$f^*$  : plus court temps de traversée des chaînes de montage

$l^*$  : le numéro de chaîne de sortie du plus court cheminement

```
1 début
2    $f_1 \leftarrow func1(a, e_1, e_2, t, n, l) + x_1$ 
3    $f_2 \leftarrow func2(a, e_1, e_2, t, n, l) + x_2$ 
4   si  $f_1 \leq f_2$  alors
5      $l^* \leftarrow 1$ 
6      $f^* \leftarrow f_1$ 
7   sinon
8      $l^* \leftarrow 2$ 
9      $f^* \leftarrow f_2$ 
```

---

*solutions récursives et itératives présentées dans l'exercice précédent.*

Le programme C permettant d'exécuter la calcul de la solution optimale dans sa version récursive ou itérative est donnée sur la page de téléchargement de cette feuille d'exercices. Il suffit ensuite de compiler ce code avec un compilateur comme `gcc`, puis de lancer le programme. Lancé sans paramètre, le programme affiche les paramètres à utiliser.

**Solution question 3.8** : *Comparer les exécutions des deux solutions. Jusqu'à quelle taille de chaîne de montage êtes vous allé avec la version récursive ? Quel temps cela a pris pour le calcul ? Quelle a été la durée du même calcul avec la version itérative issue de la programmation dynamique ? Conclure.*

---

**Algorithme 6** : Fonctions *func1* et *func2*

---

**Données** :  $a_{i,j}$  : temps de montage au poste  $S_{i,j}$ ,  $i = 1..2$ ,  $j = 1..n$

$e_1, e_2$  : temps d'arrivée sur les chaînes 1 et 2

$t_{1,j}$  : temps de transfert des chaînes 1 à 2 entre  $S_{1,j}$  et  $S_{2,j+1}$ ,  $j = 1..n - 1$

$t_{2,j}$  : temps de transfert des chaînes 2 à 1 entre  $S_{2,j}$  et  $S_{1,j+1}$ ,  $j = 1..n - 1$

$j$  : l'étage courant

**Résultat** :  $l_{i,j}$  : chaîne de sortie du sous chemin optimal,  $i = 1..2$ ,  $j = 1..n$

$f^*$  : plus court temps de traversée des chaînes de montage

```
1 fonction func1(a, e1, e2, t, j, l)
2 début
3   si j = 1 alors f* ← e1 + a1,1
4   sinon
5     f1 ← func1(a, e1, e2, t, j - 1, l)
6     f2 ← func2(a, e1, e2, t, j - 1, l)
7     si f1 ≤ f2 + t2,j-1 alors
8       l1,j ← 1
9       f* ← f1 + a1,j
10    sinon
11      l1,j ← 2
12      f* ← f2 + t2,j-1 + a1,j

13 fonction func2(a, e1, e2, t, j, l)
14 début
15   si j = 1 alors f* ← e2 + a2,1
16   sinon
17     f1 ← func1(a, e1, e2, t, j - 1, l)
18     f2 ← func2(a, e1, e2, t, j - 1, l)
19     si f2 ≤ f1 + t1,j-1 alors
20       l2,j ← 2
21       f* ← f2 + a2,j
22    sinon
23      l2,j ← 1
24      f* ← f1 + t1,j-1 + a2,j
```

---



---

**Algorithme 7 : Recherche itérative du plus court cheminement dans les chaînes**

---

**Données :**  $a[1..n]$  : temps de montage au poste  $S_{i,j}$

$e_1, e_2$  : temps d'arrivée sur les chaînes 1 et 2

$x_1, x_2$  : temps de sortie des chaînes 1 et 2

$t_{1,j}$  : temps de transfert des chaînes 1 à 2 entre  $S_{1,j}$  et  $S_{2,j+1}$ ,  $j = 1..n - 1$

$t_{2,j}$  : temps de transfert des chaînes 2 à 1 entre  $S_{2,j}$  et  $S_{1,j+1}$ ,  $j = 1..n - 1$

**Résultat :**  $f^*$  : plus court temps de traversée des chaînes de montage

$l^*$  : le numéro de chaîne de sortie du plus court cheminement

```
1 début
2    $f_1[1] \leftarrow e_1 + a_{1,1}$ 
3    $f_2[1] \leftarrow e_2 + a_{2,1}$ 
4   pour  $j$  de 2 à  $n$  faire
5       si  $f_1[j - 1] + a_{1,j} \leq f_2[j - 1] + t_{2,j-1} + a_{1,j}$  alors
6            $f_1[j] \leftarrow f_1[j - 1] + a_{1,j}$ 
7            $l_1[j] \leftarrow 1$ 
8       sinon
9            $f_1[j] \leftarrow f_2[j - 1] + t_{2,j-1} + a_{1,j}$ 
10           $l_1[j] \leftarrow 2$ 
11       si  $f_2[j - 1] + a_{2,j} \leq f_1[j - 1] + t_{1,j-1} + a_{2,j}$  alors
12            $f_2[j] \leftarrow f_2[j - 1] + a_{2,j}$ 
13            $l_2[j] \leftarrow 2$ 
14       sinon
15            $f_2[j] \leftarrow f_1[j - 1] + t_{1,j-1} + a_{2,j}$ 
16           $l_2[j] \leftarrow 1$ 
17   si  $f_1[n] + x_1 \leq f_2[n] + x_2$  alors
18        $f^* \leftarrow f_1[n] + x_1$ 
19        $l^* \leftarrow 1$ 
20   sinon
21        $f^* \leftarrow f_2[n] + x_2$ 
22        $l^* \leftarrow 2$ 
```

---

---

**Algorithme 8 : Affichage d'un cheminement optimal**

---

**Données :**  $l_i[j = 1..n]$  : le numéro de chaîne de sortie du plus court cheminement à l'étape  $j$  avant la réalisation de la tâche  $S_{i,j}$

```
1 début
2    $i \leftarrow l^*$ 
3   afficher "chaîne"  $i$  "poste"  $n$ 
4   pour  $j$  de  $n$  à 2 faire
5        $i \leftarrow l_i[j]$ 
6       afficher "chaîne"  $i$  "poste"  $j - 1$ 
```

---