Informatique Tronc Commun

PTSI-PT*

Séquence 4

Programmation dynamique

Thème : Résolution de problèmes d'optimisation discrète

Objectifs

- Réaliser une programmation dynamique ;
- Connaître la propriété de sous-structure optimale ;
- Reconnaître le chevauchement de sous-problèmes ;
- Calcul de bas en haut ou par mémoïsation ;
- Reconstruction d'une solution optimale à partir de l'information calculée.

Table des matières

Cours - Programmation dynamique

1 Problème de l'effet de recouvrement : Un premier exemple

1.1 Un exemple pour bien comprendre

Le calcul de $\binom{n}{p}$ peut être réalisé à l'aide de la formule de Pascal :

$$\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$$

De plus, nous vérifions :

$$\text{Si } p > n, \binom{n}{p} = 0, \qquad \text{si } n = p, \binom{n}{p} = 1 \qquad \text{et si } p = 0, \binom{n}{p} = 1$$

La fonction récursive que l'on peut programmer est donc :

```
1 def binom(n,p):
2     if p==0 or n==p:
3         return 1
4     elif p>n:
5         return 0
6     else:
7         return binom(n-1,p-1)+binom(n-1,p)
```

Cette programmation de fonction est très peu efficace. Illustrons le avec l'arbre d'appels récursifs de $\binom{5}{2}$.

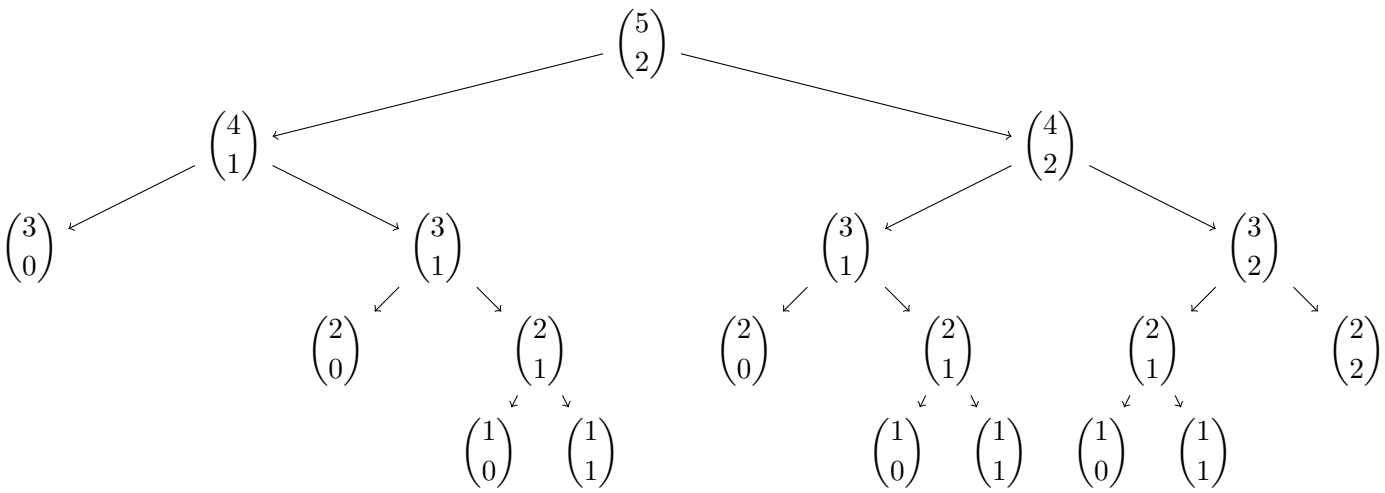


FIGURE 1 – Exemple du calcul de $\binom{5}{2}$

Dans cet exemple le calcul de $\binom{5}{2}$ nécessite de réaliser 3 fois le calcul de $\binom{2}{1}$. Dès que nous passons au calcul de $\binom{30}{15}$, il faut calculer 40 116 600 fois $\binom{2}{1}$. Cette méthode de résolution est donc très peu efficace.

1.2 Complexité temporelle

On va chercher à évaluer la complexité de cette fonction, on note $C(n,p)$ le nombre d'additions réalisées par cette fonction, on dispose des relations :

$$C(n,0) = C(n,p) = 0 \quad \text{et} \quad \forall p \in [\![1, n-1]\!], \quad C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$$

On démontre alors par récurrence sur $n \in \mathbb{N}$ que pour tout $p \in [\![0,n]\!]$, $C(n,p) = \binom{n}{p} - 1$. De plus, la complexité temporelle est maximale lorsque $n \simeq p/2$. Or, la formule de Stirling permet d'établir l'équivalent : $\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi n}}$; le calcul de $\binom{2n}{n}$ par cette fonction est donc de complexité exponentielle.

1.3 Effet de recouvrement

Le problème à résoudre, ici le calcul de $\binom{n}{p}$, se ramène à la résolution de deux sous-problèmes : $\binom{n-1}{p-1}$ et $\binom{n-1}{p}$: **sous-problèmes qui sont en interaction**. Par exemple, on constate sur la FIGURE 1 que le calcul de $\binom{4}{1}$ et le calcul de $\binom{4}{2}$ font tous deux appel au même sous-problème : le calcul de $\binom{3}{1}$. Ainsi, la présence de sous-problèmes en interaction peut faire croitre très rapidement la complexité d'une fonction, au point d'en rendre son usage rédhibitoire.

1.4 Programmation dynamique : démarche de résolution de bas en haut (bottom-up)

L'approche de bas en haut ou bottom-up consiste à résoudre d'abord les problèmes plus simples (instances de taille 1), puis de plus en plus complexes (instances de taille 2), etc. jusqu'à arriver à la résolution du problème de la taille demandée.

Dans notre exemple présenté en FIGURE 1, la résolution par programmation dynamique se fait en suivant le schéma suivant :

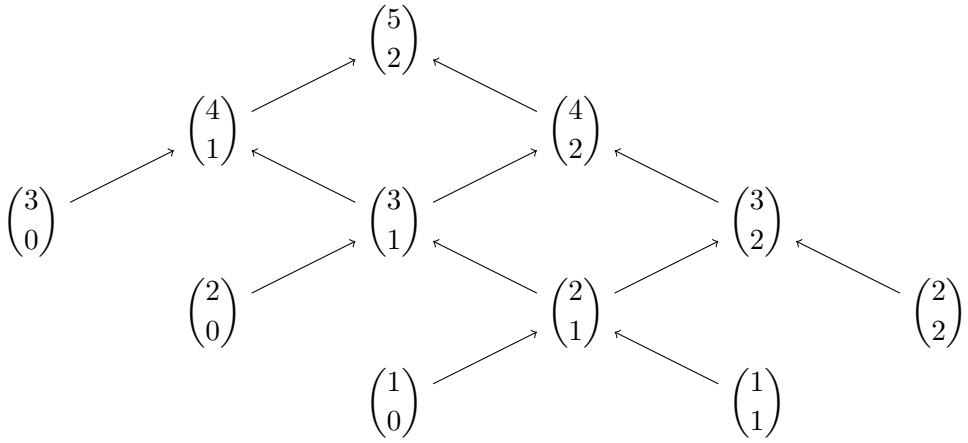


FIGURE 2 – Programmation dynamique de bas en haut : bottom-up

En général, pour réaliser ce type de solution on utilise un tableau, ici un tableau bi-dimensionnel $(n+1)\times(p+1)$ (dont seule la partie pour laquelle $i \geqslant j$ sera utilisée). Ce tableau sera progressivement rempli par les valeurs des coefficients binomaux, en commençant par les plus petits (voir FIGURE 3).

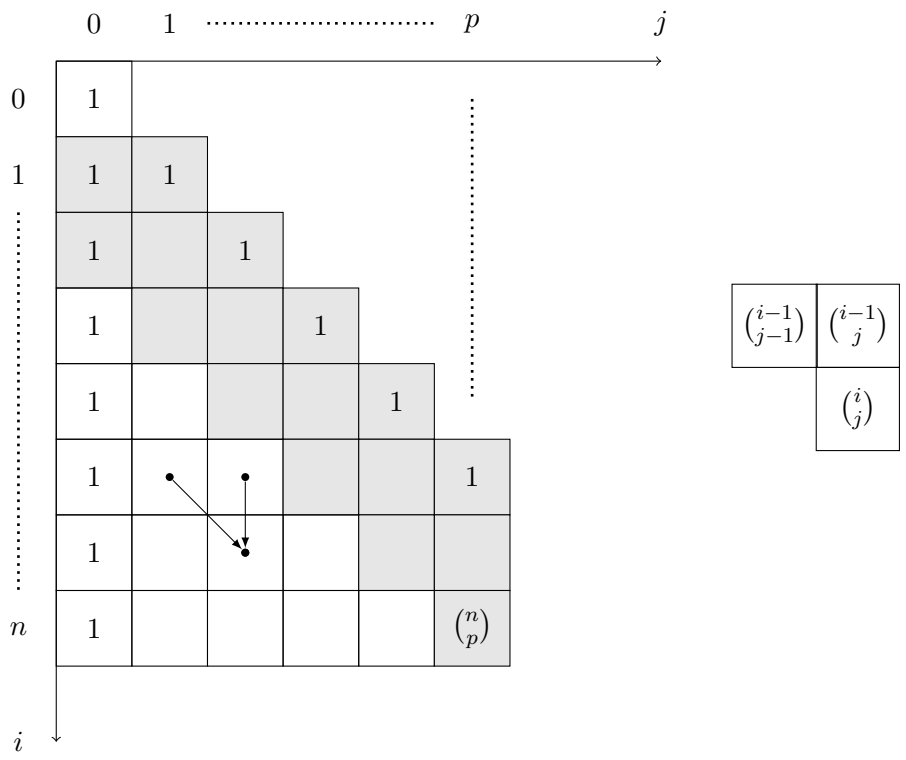


FIGURE 3 – Le schéma de dépendance du calcul de $\binom{n}{p}$

Il faut faire attention à bien respecter la relation de dépendance (modélisée par les flèches sur le schéma ci-dessus) pour remplir les cases de ce tableau : la case destinée à recevoir la valeur de $\binom{i}{j}$ ne peut être remplie qu'après les cases destinées à recevoir $\binom{i-1}{j-1}$ et $\binom{i-1}{j}$. Un code réalisant ce type de programmation est alors :

```
def mini(number1,number2):
    if number1<number2:
        return number1
    else:
        return number2

import numpy as np
def binom_bottom_up(n,p):
    tab=np.zeros((n+1,p+1)) # Création du tableau
    tab[:,0]=1 # 1ere colonne du tableau mise à 1
    for i in range(1,p+1):
        tab[i,i]=1 # Diagonale de (pxp) mise à 1
    for i in range(2,n+1):
        for j in range(1,mini(p,i)+1):
            tab[i,j]=tab[i-1,j-1]+tab[i-1,j] # calcul de la relation de ré
                currence
    return tab[n,p]
```

Au prix d'un coût spatial (la création du tableau) cet algorithme est plus efficace que l'algorithme récursif naïf initial puisque sa complexité temporelle est maintenant en $\mathcal{O}(np)$.

Remarque 1

Toutes les cases du tableau sont ici complétées. Cela est inutile pour le seul calcul de $\binom{n}{p}$: seules les cases grisées sont nécessaires.

Conclusion

La programmation dynamique de ce calcul permet de réduire drastiquement les coûts de calcul. Un inconvénient non négligeable réside dans la perte de lisibilité de l'algorithme, comparativement à l'algorithme récursif.

1.5 Programmation dynamique : démarche de résolution de haut en bas (top-down)

Dans la section précédente, nous avons vu un moyen plus efficace de calculer $\binom{n}{p}$. Cependant ; la lecture de l'algorithme obtenu n'est pas si simple que cela. L'idéal serait donc de combiner l'élégance de la programmation récursive avec l'efficacité dynamique.

Ce principe s'appuie sur une démarche de haut en bas à l'aide de la **mémoïsation**. On va associer à la fonction un dictionnaire qui va mémoriser le résultat du calcul réalisé. Ainsi, à chaque fois que le programme aura besoin de calculer une valeur, il ira voir dans le dictionnaire si la valeur dont il a besoin a déjà été calculée, et ne réalisera le calcul que dans le cas contraire, en ajoutant ensuite la nouvelle valeur calculée au dictionnaire. Le calcul du coefficient binomial va alors prendre la forme qui suit :

```
1  binom_dict = {} # binom_dict est une variable qui sera utilisée comme variable
       globale dans la fonction binom
2
3  def binom(n, p):
4      if (n, p) not in binom_dict: # Test pour savoir si le tuple (n,p) est une cl
           é du dictionnaire binom_dict
5          if p == 0 or n == p: # Cas de base
6              b=1
7          else:
8              b=binom(n-1,p-1) + binom(n-1,p) # Récursion
9          binom_dict[(n, p)] = b  # Stockage dans le dictionnaire # binom_dict est
               modifiée car un dictionnaire est mutable
10     return binom_dict[(n, p)]
```

On retrouve donc la structure de code récursif vu précédemment. L'ordre des éléments entrés dans le dictionnaire est alors :

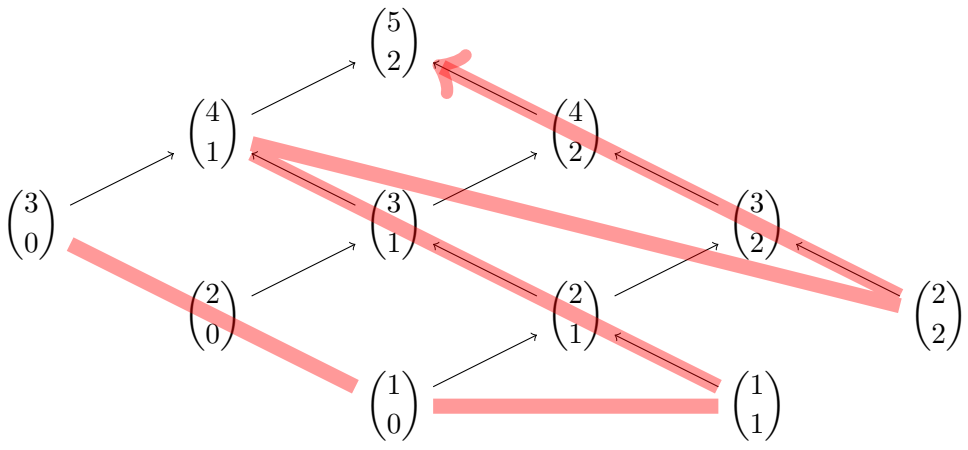


FIGURE 4 – Ordre d'entrée dans le dictionnaire

Conclusion

Cette façon de procéder évite donc d'avoir à recalculer des valeurs déjà calculées. La mémoïsation rend alors l'algorithme récursif bien plus efficient. On a alors l'avantage du gain de temps de calcul et de la simplicité de lecture de l'algorithme

2 Les bases

2.1 La théorie

La programmation dynamique, résout des problèmes en combinant des solutions de sous-problèmes. Le mot « **Programmation** » dans ce contexte, fait référence à l'établissement d'un plan pour résoudre un problème. La méthode de résolution envisagée est tabulaire et ne fait pas référence à l'établissement d'un code informatique (même si celui-ci sera à établir au final). Le fondement de l'utilisation de la programmation dynamique s'appuie sur le fait que l'on peut décomposer un problème en sous-problèmes. La résolution de ces sous-problèmes amènent alors à une solution admissible du problème initial.

La programmation dynamique peut s'appliquer lorsque les sous-problèmes ne sont pas indépendants, c'est-à-dire lorsque des sous-problèmes ont des sous-sous-problèmes communs. On parle de **recouvrement** de sous-problèmes. Un algorithme de programmation dynamique résout chaque sous-sous-problème une seule fois et mémorise sa réponse dans un tableau, évitant ainsi le recalcul de la solution chaque fois que le sous-sous-problème est rencontré.

La programmation dynamique est, en général, appliquée aux **problèmes d'optimisation**. C'est-à-dire des problèmes où l'on recherche la combinaison de paramètres permettant d'obtenir l'optimum d'une fonction objectif.

Exemple 1

Le problème du rendu de monnaie est un problème d'optimisation. Les paramètres sont les devises possibles à rendre (1€, 2€, 5€, etc.). La fonction objectif est la quantité de devises rendues. Il y a une contrainte : on ne doit pas rendre plus ou moins d'argent que nécessaire. On cherche donc à rendre le moins de devises possibles dans le rendu de monnaie tout en s'assurant qu'on rende bien la bonne quantité d'argent.

Le développement d'un algorithme de programmation dynamique peut être découpé en quatre étapes :

1. Caractériser la structure d'une solution optimale : **c'est le point clé** ;
2. Définir récursivement la valeur d'une solution optimale ;
3. Calculer la valeur d'une solution optimale de manière ascendante (bottom-up) ou descendante (top-down) ;
4. Construire une solution optimale à partir des informations calculées.

Les étapes 1–3 forment la base d'une résolution de problème à la mode de la programmation dynamique. On peut omettre l'étape 4 si l'on n'a besoin que de la valeur d'une solution optimale. Lorsqu'on effectue l'étape 4, on gère parfois des informations supplémentaires pendant le calcul de l'étape 3 pour faciliter la construction d'une solution optimale.

2.2 Un exemple pour mieux comprendre

2.2.1 Présentation du problème

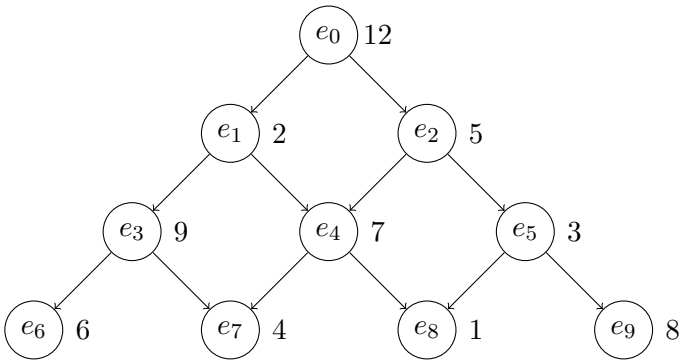


FIGURE 5 – Exemple de pyramide de nombres

En FIGURE 5, on considère la pyramide de nombres entiers. Celle-ci est représentée par un graphe orienté, dont chacun des sommets étiquetés de e_0 à e_9 porte la valeur figurant à sa droite. On cherche le chemin allant de e_0 (sommet de la pyramide) à l'un quelconque des éléments de la base (e_6, e_7, e_8 ou e_9) traversant des nombres dont la somme est maximale.

2.2.2 Caractérisation de la structure d'une solution optimale

Le calcul du chemin de valeur maximale associé à l'élément e_i est noté $s_{opt}(e_i)$.

Exemple d'application 1

Q1. Donnez l'expression de $s_{opt}(e_7)$ en fonction des sommets auxquels est reliés e_7.

Q2. Donnez l'expression de $s_{opt}(e_8)$ en fonction des sommets auxquels est reliés e_8.

Q3. Que constatez-vous ?

Q4. Observez-vous le même phénomène constaté pour le calcul de $s_{opt}(e_3)$, $s_{opt}(e_4)$ ou $s_{opt}(e_5)$?

Q5. Que pouvez-vous en conclure ?

D'après ce que nous venons de réaliser, nous avons exhibé
mais nous n'avons pas exhibé ! C'est-à-dire que pour l'instant nous n'avons pas démontré que la solution optimale du problème peut s'obtenir à partir des solutions optimales de sous-problèmes. Mais nous l'avons presque fait !

On note $s_{sol} = \max(s_{opt}(e_6), s_{opt}(e_7), s_{opt}(e_8), s_{opt}(e_9))$ la solution du problème recherché. D'après les réponses aux questions 1 et 2, pour déterminer le calcul du chemin de valeur maximale associé aux éléments e_6, e_7, e_8 et e_9, il est nécessaire de calculer le chemin de valeur maximale associé aux éléments e_3, e_4, e_5 etc. **De ce fait, la solution s_{sol} répond à ce critère de solution à un problème à sous-structure optimale. De plus, le fait que certains sous-problèmes soient communs nous fait penser que la programmation dynamique est parfaitement adapté pour trouver la réponse au problème donné.**

2.2.3 Définition récursive du problème à résoudre

Soit la pyramide de hauteur h ayant $n = \left(\sum_{k=1}^{h+1} k\right)$ éléments (voir FIGURE 6). Appelons $v(i)$ la valeur de l'élément e_i, $prg(i)$ et $prd(i)$ les numéros des (au plus) deux prédécesseurs possibles (gauche et droite) de e_i sur tout chemin de e_0 à e_i. La valeur $s_{opt}(e_i)$ dépend en général de celles de ses deux prédécesseurs et le cas des éléments n'ayant qu'un ou pas de prédécesseur doit être traité séparément.

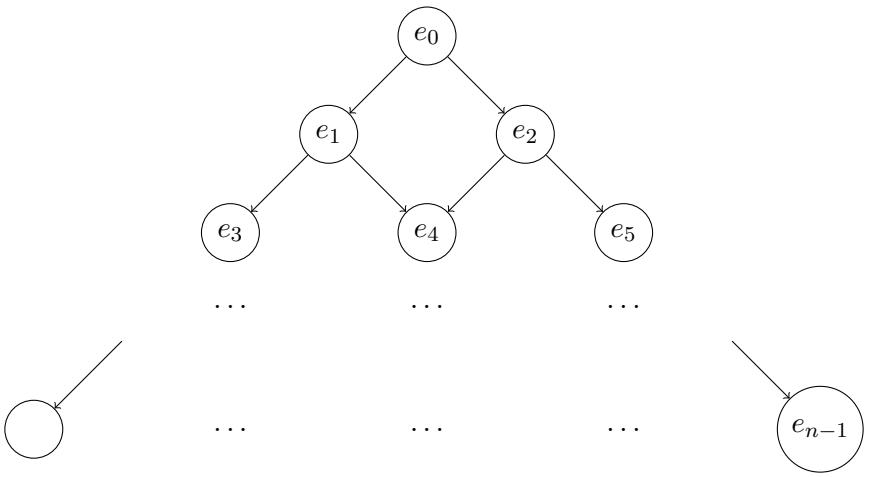


FIGURE 6 – Une pyramide de nombres

Exemple d'application 2

On pose I_d (respectivement I_g) désigne l'ensemble des indices de la branche la plus à droite (respectivement gauche).

Q6. Déterminer les relations de récurrence à mettre en place pour résoudre le problème.

2.2.4 Résolution tabulaire du problème

Dans cette partie, on va résoudre à l'aide d'un tableau le problème de la pyramide de nombres présentées à la FIGURE 5. Dans chaque case du tableau, on va indiquer la valeur maximale permettant d'accéder au sommet e_i. On indiquera également le prédécesseur de e_i permettant d'accéder à cette valeur maximale (ça ne vous rappelle rien ?).

Exemple d'application 3

Q7. Compléter le tableau permettant de résoudre le problème présenté en FIGURE 5 par une démarche bottom-up.

12 $\emptyset$									
e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9

On constate que la démarche bottom-up n'est pas spécifiquement difficile à mettre en place sur cet exemple dès que l'on connait facilement les prédécesseurs d'un élément. Une approche top-down est également possible bien sûr mais il faut retenir (ou noter) où l'on se situe dans la pile d'exécution.

2.2.5 Préparation du traitement informatique du problème

On considère que la pyramide de nombres est renseignée sous forme de dictionnaire. Les clés de ces dictionnaires sont les chaînes de caractères `ei` correspondant à l'élément e_i. La valeur associée à chaque clé est une liste contenant la valeur de l'élément e_i ainsi que les successeurs. Si un sommet ne dispose pas de successeurs, alors on ne renseignera dans la valeur que la valeur du sommet.

Exemple d'application 4

Q8. Renseigner alors le dictionnaire représentatif de la pyramide de nombre de la FIGURE 5

Que ce soit avec une approche bottom-up ou top-down (mémoïsation), il va être plus facile de traiter le problème en connaissant le/les prédécesseurs d'un sommet.

Q9. Établir une fonction `dico_predecesseur(G:dict)->dict` prenant en argument un graphe orienté G et renvoyant un dictionnaire des prédécesseurs des éléments. Le prédécesseur de e_0 sera indiqué comme étant égal à `None`.

2.2.6 Approche bottom-up

Voici un code proposant une approche bottom-up traitant le problème de la pyramide de nombres.

```
1  def position_tableau(sommet):
2      return int(sommet[1:]) # Un sommet est noté ei, pour récupérer la valeur de
           i, on fait donc int(sommet[1:]) afin de ne pas tenir compte du e et d'
           avoir la valeur numérique de i
3
4  def bottom_up(G):
5      predecesseur_graphe=dico_predecesseur(G) # Dictionnaire contenant pour
           chaque sommet le/les prédécesseurs de celui-ci.
6      valeur_ei=[0 for i in range(len(G))] # Initialisation du tableau à compléter
            et qui contiendra la valeur maximale permettant d'accéder au sommet e_i
           .
7      predecesseur_ei=[None for i in range(len(G))] # Initialisation du tableau
           contenant le prédecesseur de e_i qui amène à la valeur maximale.
8      valeur_ei[0]=G['e0'][0] # On renseigne la seule valeur connue initialement :
            celle du noeud e0 qui n'a aucun prédécesseur.
9      n=len(G) # Ordre du graphe.
10     for i in range (1,n): # Parcours de chacun des noeuds.
11         sommet_courant='e'+str(i) # sommet courant actuellement traité
12         if len(predecesseur_graphe[sommet_courant])==1: # Si ce sommet n'a qu'un
               seul prédécesseur
13             valeur_ei[i]=valeur_ei[position_tableau(predecesseur_graphe[
                   sommet_courant][0])]+G[sommet_courant][0] # On récupère la
                   valeur du chemin parcouru jusqu'au prédécesseur et on y ajoute
                   celle du sommet courant
14             predecesseur_ei[i]=predecesseur_graphe[sommet_courant][0] # Le pré
                   decesseur du sommet courant est nécessairement le seul répertori
                   é.
15         elif len(predecesseur_graphe[sommet_courant])==2: # Si le sommet a deux
               prédécesseurs
16             valeur_predecesseur_1=valeur_ei[position_tableau(predecesseur_graphe
                   [sommet_courant][0])] # Récupération de la valeur maximale
                   obtenue jusqu'au premier prédécesseur
17             valeur_predecesseur_2=valeur_ei[position_tableau(predecesseur_graphe
                   [sommet_courant][1])] # Récupération de la valeur maximale
                   obtenue jusqu'au second prédécesseur
18             maxi=max(valeur_predecesseur_1,valeur_predecesseur_2) # On récupère
                   le max de ces deux nombres.
19             valeur_ei[i]=maxi+G[sommet_courant][0] # On assigne alors la valeur
                   maximale permettant d'accéder au noeud ei
20             if maxi==valeur_predecesseur_1: # Le prédécesseur permettant d'accé
                   der au noeud ei avec la plus grande valeur est stocké dans le
                   tableau predecesseur_ei
21                 predecesseur_ei[i]=predecesseur_graphe[sommet_courant][0]
22             else:
23                 predecesseur_ei[i]=predecesseur_graphe[sommet_courant][1]
24     return valeur_ei,predecesseur_ei
```

Conclusion

L'approche bottom-up fournit donc bien un algorithme itératif mais celui-ci n'est pas forcément toujours évident à décrypter.

2.2.7 Approche par mémoïsation

L'approche par mémoïsation que l'on propose de mettre en place doit notamment permettre de déterminer la valeur des sommets n'ayant pas de successeurs. En effet, le chemin de longueur maximale aboutira forcément à l'un d'entre eux.

Exemple d'application 5

Q10. Créer une fonction `sommets_sans_successeur(G:dict)->list`, retournant la liste des sommets qui n'ont pas de successeurs. Ce sont ces sommets qui nous permettront d'initialiser notre calcul.

Q11. Voici un code (un peu naïf) que vous devez compléter et qui utilise une approche par mémoïsation.

```
1  def top_down(G):
2      predecesseur_graphe=dico_predecesseur(G) # Dictionnaire contenant pour
           chaque sommet le/les prédécesseurs de celui-ci.
3      dico_valeur_ei={} # Initialisation du tableau à compléter et qui contiendra
           la valeur maximale permettant d'accéder au sommet e_i.
4      dico_predecesseur_ei={} # Initialisation du tableau contenant le pré
           decesseur de e_i qui amène à la valeur maximale.
5      sommet_a_traiter=sommets_sans_successeur(G) # Récupération des sommets sans
           successeurs
6      for sommet in sommet_a_traiter: # Itération
7          memoisation(sommet,predecesseur_graphe,dico_valeur_ei,
               dico_predecesseur_ei)
8      return dico_valeur_ei,dico_predecesseur_ei
9
10 def memoisation(sommet,predecesseur_graphe,dico_valeur_ei,dico_predecesseur_ei)
       :
11     pred=predecesseur_graphe[sommet] # Récupération du/des prédécesseur(s) du
           sommet
12     ## Cas de base
13     if sommet=='e0':
14         dico_valeur_ei[sommet]=G[sommet][0]
15         dico_predecesseur_ei[sommet]=None
16     ## Cas où le sommet se trouve sur la branche droite ou gauche
17     elif len(pred)==1:
18         if sommet not in dico_valeur_ei: # Si la valeur_ei du chemin de longueur
               maximale aboutissant à ei n'a pas été enregistrée...
19             nombre=memoisation(.....................................)
20             dico_valeur_ei[sommet]=.........+........ # Calcul de valeur_ei du
                   sommet courant
21             dico_predecesseur_ei[sommet]=..............# Récupération du prédé
                   cesseur de ei aboutissant au chemin de plus grande valeur
22     ## Cas où le sommet dispose de 2 prédécesseurs
23     else:
24         if sommet not in dico_valeur_ei:
25             nombre1=..................................... # ... on effectue
                   le calcul
26             nombre2=.....................................
27             maxi=max(nombre1,nombre2) # Maximum de la valeur entre les deux
28             dico_valeur_ei[sommet]=.....................
29             if maxi==nombre1: # Le prédécesseur permettant d'accéder au noeud ei
                   avec la plus grande valeur est stockée dans le tableau
```

```
                predecesseur_ei
30              dico_predecesseur_ei[sommet]=pred[0]
31          else:
32              dico_predecesseur_ei[sommet]=pred[1]
33  return dico_valeur_ei[sommet]
```

2.2.8 Traitement des résultats

Après utilisation de la fonction `top_down`, on récupère deux dictionnaires contenant la valeur maximale associée au chemin aboutissant au nœud e_i pour $\in [\![0, n-1]\!]$. Le second correspond au prédécesseur du nœud e_i du chemin de valeur maximale aboutissant à e_i.
Après utilisation de la fonction `bottom_up`, on récupère deux listes. La première contenant la valeur maximale associée au chemin aboutissant au nœud e_i pour $\in [\![0, n-1]\!]$. La seconde correspond au prédécesseur du nœud e_i du chemin de valeur maximale aboutissant à e_i.

Exemple d'application 6

Q12. Établir le code permettant d'obtenir la valeur du chemin de valeur maximale, ainsi que celui-ci sous forme de listes contenant les nœds à parcourir. Sur l'exemple traité en FIGURE 5, le résultat obtenu est : `(29,['e0','e1','e3','e6'])`.

3 Application de la programmation dynamique

Il existe deux grandes caractéristiques que doit posséder un problème d'optimisation pour que la programmation dynamique soit applicable : sous-structure optimale et chevauchement des sous-problèmes.

3.1 Sous-structure optimale

La première étape de la résolution d'un problème d'optimisation via la programmation dynamique est de caractériser la structure d'une solution optimale. Retenons qu'un problème exhibe une sous-structure optimale si une solution optimale au problème contient en elle des solutions optimales de

sous-problèmes. : **C'est le principe de Bellman**. Chaque fois qu'un problème exhibe une sous-structure optimale, c'est un bon indice de l'utilisation de la programmation dynamique. (Cela peut aussi signifier qu'une stratégie gloutonne est applicable). Avec la programmation dynamique, on construit une solution optimale du problème à partir de solutions optimales de sous-problèmes. Par conséquent, on doit penser à vérifier que la gamme des sous-problèmes que l'on considère inclut les sous-problèmes utilisés dans une solution optimale.

Exemple 2

Dans l'exemple de la pyramide de nombres précédemment traité, nous avons observé que la valeur de chemin maximal jusqu'à un sommet situé à la hauteur i du graphe est atteint à partir du maximum du/des sommets prédécesseurs à hauteur $i-1$.

Exemple 3

Prenons l'exemple du plus court chemin entre deux sommets quelconques d'un graphe orienté G. Il est facile de montrer par l'absurde qu'un chemin est de longueur minimale si et seulement si ses sous-chemins sont de longueur minimale. Soit en effet un graphe orienté G et un chemin de a à d de longueur minimale, qui passe par b et c (arcs en trait plein sur la FIGURE 7*). La longueur de ce chemin est la somme des longueurs des sous-chemins de a à b, de b à c et de c à d. Supposons qu'il existe dans le graphe G un chemin de b à c de coût moindre (en pointillé sur la figure* FIGURE 7*) que le chemin de b à c choisi ; il existe alors un chemin plus court de a à d, ce qui est contraire au fait que le chemin de a à d est de longueur minimale.*

FIGURE 7 – Application du principe d'optimalité à la recherche du plus court chemin dans un graphe

Exemple 4

Prenons maintenant un autre exemple, la recherche du plus long chemin sans circuit entre deux sommets quelconques d'un graphe orienté. Soit le graphe de la FIGURE 8*. Le plus long chemin sans circuit de a à c est de longueur 2 $\langle a, b, c\rangle$, le plus long chemin sans circuit de a à b est de longueur 2 $\langle a, c, b\rangle$ et le plus long chemin sans circuit de b à c est de longueur 1. Le plus long chemin sans circuit de a à c ne s'obtient donc pas par composition de chemins sans circuit, ce qui montre que le principe d'optimalité n'est pas vérifié dans ce cas.*

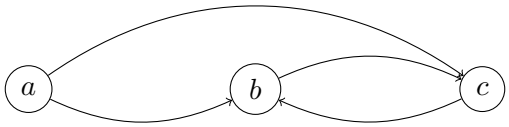


FIGURE 8 – Un (contre-)exemple où le principe d'optimalité ne s'applique pas.

La découverte de la sous-structure optimale obéit au schéma général suivant :

1. Vous montrez qu'une solution du problème consiste à faire un choix, par exemple à choisir un sommet prédécesseur dans le problème de la pyramide de nombre.
2. Vous supposez que, pour un problème donné, on vous donne le choix qui conduit à une solution optimale. Pour l'instant, vous ne vous souciez pas de la façon dont on détermine ce choix. Vous faites comme si on « vous le donnez tout cuit ».
3. À partir de ce choix, vous déterminez quels sont les sous-problèmes qui en découlent et comment caractériser au mieux l'espace des sous-problèmes résultant.
4. Vous montrez que les solutions des sous-problèmes employées par la solution optimale du problème doivent elles-mêmes être optimales, et ce en utilisant la technique du « couper-coller » : vous supposez que chacune des solutions de sous-problème n'est pas optimale et vous en déduisez une contradiction. En particulier, en « coupant » une solution de sous-problème non optimale et en la

« collant » dans la solution optimale, vous montrez que vous obtenez une meilleure solution pour le problème initial, ce qui contredit l'hypothèse que vous avez déjà une solution optimale. S'il y a plusieurs sous-problèmes, ils sont généralement similaires, de sorte que l'argument couper-coller utilisé pour l'un peut resservir pour les autres, moyennant une petite adaptation.

Conclusion

Dans les faits, le résumé de tout cela est de réussir à établir une récurrence dont le terme correspond à la grandeur optimale recherchée. En effet, l'établissement de la récurrence prouve de facto que le principe d'optimalité s'applique.

3.2 Chevauchement des sous-problèmes

La seconde caractéristique que doit avoir un problème d'optimisation pour que la programmation dynamique lui soit applicable est la suivante : l'espace des sous-problèmes doit être « réduit », au sens où un algorithme récursif pour le problème résout constamment les mêmes sous-problèmes au lieu d'en engendrer toujours de nouveaux. En général, le nombre total de sous-problèmes distincts est polynomial par rapport à la taille de l'entrée. Quand un algorithme récursif repasse sur le même problème constamment, on dit que le problème d'optimisation contient des sous-problèmes qui se chevauchent.

Exemple 5

Nous avons très clairement illustré ce principe de chevauchement des sous-problèmes lors du calcul de $\binom{n}{p}$ *où certains calculs* $\binom{i}{j}$ *devaient être effectués plusieurs fois.*

Le principe de mémoïsation permet assez simplement d'éviter à avoir à recalculer des sous-problèmes déjà traités.

Exemple d'application 7

Q13. Est-ce que le principe de mémoïsation est intéressant lors du tri fusion ?

Conclusion

Dans le cadre d'un problème d'optimisation, l'utilisation de la programmation dynamique est fortement recommandé afin de résoudre celui-ci. En général, c'est une bonne façon de traiter le problème même si la garantie de trouver une solution (en effet, la solution n'est pas forcément unique) n'est pas vérifiée. La programmation dynamique est préconisée lorsque :

- le principe de Bellman est applicable. **Pour rappel, ne chercher pas forcément à vérifier *a priori* ce principe mais établir directement la relation de récurrence** ;
- Il existe un chevauchement des sous-problèmes.

4 Programmation dynamique et algorithme glouton

4.1 Introduction

En première année, nous avons vu un genre d'algorithme permettant de résoudre certains problèmes d'optimisation : **les algorithmes gloutons**. Ceux-ci sont utilisés lorsque l'on cherche à construire la solution optimale du problème comme une succession de choix suivant une règle heuristique appelé le **choix glouton**. Pour être efficace, un algorithme glouton doit être utilisé dans le cadre d'un problème d'optimisation où **le principe de Bellman** est applicable.

Le choix glouton correspond à un choix qui est la solution optimale du sous-problème considéré. On construit alors la solution complète du problème par concaténation des différentes solutions des sous-problèmes étudiés. Autrement dit : quand on considère le choix à faire, on fait le choix qui paraît le meilleur pour le sous-problème courant, sans tenir compte des résultats des sous-problèmes précédents (approche bottom-up) ou à venir (approche top-down). C'est en cela que les algorithmes gloutons diffèrent de la programmation dynamique. En programmation dynamique, on fait un choix à chaque étape, mais ce choix dépend généralement de la solution des sous-problèmes.

Conclusion

La programmation dynamique se caractérise par la résolution par taille croissante de tous les problèmes locaux, la stratégie gloutonne consiste à choisir à partir du problème global un seul type de problème local et un seul en suivant une heuristique, c'est-à-dire une stratégie permettant de faire un choix rapide : le choix glouton

4.2 Un problème bien connu : le problème du sac à dos

4.2.1 Présentation

Le problème du sac à dos est un problème fort célèbre. Il est connu de par le fait qu'il possède la propriété d'être NP-complet [1]. La formulation de ce problème est très simple mais sa résolution est difficile dans un temps raisonnable.

Il existe deux variantes de ce problème :

- On souhaite remplir un sac à dos d'objets **fractionnables** ayant la plus grande valeur possible sans dépasser un certain poids. Chaque objet a une valeur et une masse.
- On souhaite remplir un sac à dos d'objets **non fractionnables** ayant la plus grande valeur possible sans dépasser un certain poids. Chaque objet a une valeur et une masse.

La nuance entre ces deux problèmes est donc que les objets à emporter dans le sac peuvent être fractionnables ou non.

4.2.2 Version fractionnable

Formalisation du problème

Dans sa version fractionnable le problème se formalise de la façon suivante : Soient des objets définis par des 2-uplets (valeur,masse). L'ensemble des n objets à disposition sont donc rassemblés dans l'ensemble $O = ((v_1, m_1), (v_2, m_2), \cdots, (v_n, m_n))$. On a donc v_i et m_i qui sont respectivement la valeur et la masse de l'objet i. La masse à ne pas dépasser dans le sac à dos est notée M.

Dans cette version du problème, les objets sont fractionnables, ce qui implique que la solution optimale est une solution où le sac est rempli (en supposant bien sûr que $\sum_{i=1}^{n} m_i \geq M$, ce qui est le cas sinon, le problème ne se pose pas). On précise que lorsqu'on fractionne un objet, sa valeur est également fractionnée. On note x_i la fraction choisie de l'objet i.

1. Voir la chaîne youtube « ScienceEtonnante » https ://www.youtube.com/watch ?v=AgtOCNCejQ8 qui traite de l'importance de ce genre de problème.

Q14. Écrire mathématiquement le problème d'optimisation à résoudre.

Traitement d'un exemple

Un promeneur souhaite transporter dans son sac à dos le fruit de sa cueillette. La cueillette est belle, mais trop importante pour être entièrement transportée dans le sac à dos. Des choix doivent être faits (on peut appeler cela du gâchis et le gâchis, c'est mal.). Il faut que la masse totale des fruits choisis ne dépasse pas la capacité maximale du sac à dos. Les fruits cueillis ont des valeurs différentes, et le promeneur souhaite que son chargement soit de la plus grande valeur possible (cupide en plus...).

Les informations nécessaires sont :

Fruits cueillis	Quantité ramassée	Valeur totale des fruits cueillis
Framboises	1 kg	15€
Myrtilles	3 kg	48€
Fraises	4 kg	50€

La masse totale que peut transporter le sac est de 5 kg.

Q15. Résoudre "à la main" le problème du sac à dos qui vous ai présenté. Quelle grandeur non renseignée directement dans le tableau ci-dessous avez-vous dû calculer pour trouver la solution optimale ?

Q16. Décrire l'algorithme glouton qui permet de résoudre le problème du sac à dos en version fractionnable.

Q17. Compléter le code suivant permettant de résoudre par un algorithme glouton, le problème du sac à dos dans sa version fractionnable.

```
def sac_a_dos(L:[list],capacite:int)->list,int,int:
    '''
    Entrées : L est une liste de listes des éléments pouvant être mis dans le sac à dos sous la forme : ["nom objet",valeur,masse]
              capacite est la la masse maximale que peut transporter le sac à dos
    Sortie : liste sac contenant les éléments de L et correspondant au chargement effectué. La valeur de celui-ci ainsi que sa masse
    '''
    ## 1ere étape : On indique le prix au kilo de chaque objet et on l'indique dans la liste L et on trie la liste L suivant ce critère
    for i in range(len(L)): # On parcourt l'ensemble des éléments à choisir
        L[i].append(.....) # Ajout de l'indicateur prix au kilo
    L=sorted(L,key=lambda elem:elem[3],reverse=True) # La liste L est maintenant triée dans l'ordre décroissant suivant le prix au kilo des objets
    ## 2eme étape : Initialisation des variables
    masse_sac = 0 # masse actuellement dans le sac
    sac=[] # à l'initialisation, le sac est vide
    i=0 # i sert d'indice dans la liste L
    valeur = 0 # valeur du chargement
    ## 3eme étape : Mise en place de l'algorithme glouton
    while ..... and .....:# tant qu'on n'a pas parcouru toute la cueillette (cas où la capacité du sac à dos est suffisante pour prendre toute la cueillette) et que la masse du sac n'a pas atteint sa capacité (cas où la cueillette est trop importante)
        fruit = L[i]
        capacite_restante = capacite - masse_sac
        if ..... : # si la quantité du i-eme fruit est supérieure à la capacité restante du sac
            fruit[2] = ..... # on modifie la quantité de fruit pour n'en prendre que la quantité correspondant à la capacité restante
        sac. .....(fruit) # On met le fruit dans le sac
        valeur = valeur + ..... # On calcule la nouvelle valeur du chargement
        masse_sac = masse_sac + ..... # On calcule la nouvelle masse du sac
        i=i+1 # On incrémente i pour passer au fruit suivant de la cueillette
return sac,valeur,masse_sac
```

On implémente la cueillette par la liste :

```
cueillette = [ ["framboises",15,1], ["myrtilles",48,3], ["fraises",50,4]].
```

Q18. À la suite de la fonction `sac_a_dos(L,capacite)` tapez la ligne de code permettant de l'utiliser pour déterminer les quantités de fruits à choisir pour remplir le sac à dos, à partir de la liste `cueillette`.

4.2.3 Version non fractionnable

Formalisation du problème

Dans sa version non fractionnable du problème du sac à dos, les objets doivent être choisi en plein. On note toujours $O = ((v_1, m_1), (v_2, m_2), \cdots, (v_n, m_n))$ l'ensemble des objets disponibles. On a donc v_i et m_i qui sont respectivement la valeur et la masse de l'objet i. La masse à ne pas dépasser dans le sac à dos est notée M.

Q19. Écrire mathématiquement le problème d'optimisation à résoudre.

Traitement d'un exemple

On suppose maintenant que les éléments à transporter ne sont pas fractionnables. Les fruits parmi lesquels le cueilleur doit choisir sont présentés ci-dessous :

Fruits cueillis	Masse d'un fruit	Quantité disponible	Prix au kilo
Melon de cavaillon	1 kg	1	3€/kg
Melon jaune	2 kg	1	2,5€/kg
Pastèque	3 kg	1	2€/kg

L'objectif est toujours de placer dans le sac à dos le chargement de valeur maximale, de masse totale inférieure à 5 kg. Par contre, les éléments n'étant pas fractionnables, il est possible que les choix successifs mènent à un chargement qui ne remplit pas complètement le sac à dos.

On se propose de tester la méthode gloutonne pour cette nouvelle formulation.

Q20. En utilisant la même heuristique que pour la version fractionnable, quel résultat obtenez-vous ? (Valeur et masse dans le sac à dos). Quel est la solution optimale ?

Conclusion

L'utilisation d'un algorithme glouton, ne permet de résoudre de façon systématique le problème du sac à dos dans sa version non fractionnable avec une heuristique choisie. L'utilisation de la programmation dynamique semble plus appropriée pour ce type de problème.

Programmation dynamique pour résoudre le problème du sac à dos

Pour résoudre ce problème, nous allons noter $f(k, M)$ la valeur maximale qu'il est possible d'atteindre avec k objets potentiels pour un poids total égal à M. Si l'objet d'indice k est dans la solution optimale, alors $m_k \leqslant M$ et $f(k, M) = v_k + f(k-1, M-m_k)$; s'il n'y est pas alors $f(k, M) = f(k-1, M)$. On en déduit :

$$f(k,M) = \begin{cases} \max\left(v_k + f\left(k-1, M-m_k\right), f\left(k-1, M\right)\right) & \text{si } m_k \leqslant M \\ f(k-1, M) & \text{sinon} \end{cases}$$

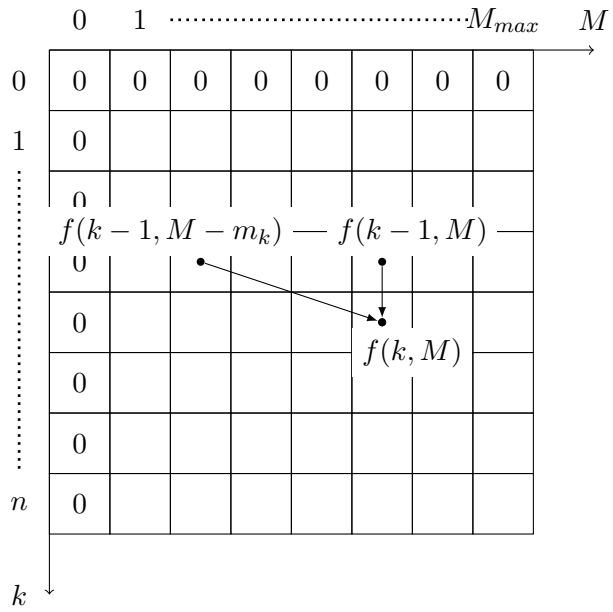


FIGURE 9 – Le schéma de dépendance du sac à dos

Pour calculer cette valeur, nous allons utiliser un tableau bi-dimensionnel de taille $(n+1)\times(M_{max}+1)$ destiné à contenir les valeurs de $f(k,M)$ pour $k \in [\![0,n]\!]$ et $M \in [\![0,M_{max}]\!]$. Avec M_{max} la masse maximale transportable dans le sac. Dans le problème, on considère que la valeur et la masse des objets est sous forme d'entiers (chose que l'on peut toujours faire). Nous prendrons comme valeurs initiales $f(0,M) = f(k,0) = 0$, et notre but est de calculer $f(n,M_{max})$.

Dans un premier temps, on considère que les noms des objets, leur valeur et leur masse sont stockés dans 3 listes différentes dans un ordre qui est le même pour les 3 listes.

```
def sac_a_dos(v:[int],m:[int],Mmax:int)->int:
    '''
    Fonction permettant de calculer f(n,Mmax) par une approche dynamique.
    Entrées :
    - v : liste des valeurs des objets
    - m : liste des masses des objets
    - Mmax : entier, représentant la masse maximale transportable
    Sortie :
    - f(n,Mmax) retourne la valeur maximale transportable dans le sac pour une
      masse totale inférieure ou égale à Mmax
    '''
    n = len(v) # Nombre d'objets potentiellement à transporter
    f = np.zeros((n + 1, Mmax + 1), dtype=int) # Création d'un tableau numpy de
      taille n+1 x Mmax+1 qui sera composé d'entiers
    for k in range(n):
        for W in range(0, Mmax + 1):
            if m[k] <= W:
                f[k + 1, W] = max(v[k] + f[k, W - m[k]], f[k, W])
            else:
                f[k + 1, W] = f[k, W]
    return f[n, Mmax]
```

Exemple d'application 8

Q21. Appliquer la fonction `sac_a_dos` sur le problème de cueillette précédent. En complétant notamment le tableau suivant avec les listes suivantes : `v=[3,5,6]`, `m=[1,2,3]`.

	0	1	2	3	4	5	M
0	0	0	0	0	0	0	
1	0						
2	0						
3	0						
n							

Remarque 2

Cet algorithme calcule la valeur maximale qui peut être emportée dans le sac, mais pas la façon d'y parvenir. Pour la connaître il faut utiliser le tableau calculé par la fonction précédente, et retrouver le chemin qui mène de la case initiale à la case finale.

```python
def objets_a_choisir(v:[int],m:[int],nom:[str],Mmax:int)->[(str,int,int)]:
    '''
    Fonction permettant de calculer la liste des objets remplissant le sac avec la plus grande valeur possible sans dépasser la masse Mmax
    Entrées :
    - v : liste des valeurs des objets
    - m : liste des masses des objets
    - nom : liste des noms des objets
    - Mmax : entier, représentant la masse maximale transportable
    Sortie :
    - Liste des objets avec leur nom, valeur et masse
    '''
    f = sacAdos(v, m, Mmax) # Calcul de f(n,Mmax)
    sac = [] # >Initialisation du sac
    k, W = len(c), Mmax
    while k > 0:
        if f[k, W] > f[k - 1, W]:
            sac.append((nom[k-1],v[k - 1], m[k - 1]))
            W -= w[k - 1]
        k -= 1
    return sac
```

On peut également créer un tableau, de taille $n+1 \times (M_{max}+1)$ qui ne contiendra uniquement les chaînes de caractères des différents objets présents dans le sac.

```python
def sac_a_dos(v:[int],m:[int],nom:[str],Mmax:int)->int,[str]:
    '''
    Fonction permettant de calculer f(n,Mmax) par une approche dynamique.
    Entrées :
    - v : liste des valeurs des objets
    - m : liste des masses des objets
    - nom : liste des noms des objets
    - Mmax : entier, représentant la masse maximale transportable
    Sortie :
    - f(n,Mmax) retourne la valeur maximale transportable dans le sac pour une masse totale inférieure ou égale à Mmax
    - tab_nom(n,Mmax)[1:] retourne les noms des objets de valeur maximale transportable dans le sac pour un masse totale inférieure ou égale à Mmax
    '''
    n = len(v) # Nombre d'objets potentiellement à transporter
    f = np.zeros((n + 1, Mmax + 1), dtype=int) # Création d'un tableau numpy de taille n+1 x Mmax+1 qui sera composé d'entiers
    longueur=max([len(i) for i in nom]) # Longueur maximale des chaînes de caractères dans la liste o
    tab_nom=np.zeros((n + 1, Mmax + 1), dtype='<U'+len(nom)*str(longueur)) # Création d'un tableau numpy de taille n+1 x Mmax+1 qui sera composé de chaînes de caractères : dtype<'Unombre' : permet de construire un tableau de chaîne de caractères chaque case peut contenir une chaîne de caractères de taille nombre.
    for k in range(n):
        for W in range(0, Mmax + 1):
            if m[k] <= W:
                f[k + 1, W] = max(v[k] + f[k, W - m[k]], f[k, W])
                if f[k + 1, W]==v[k] + f[k, W - m[k]]:
                    tab_nom[k+1,W]=tab_nom[k, W - m[k]]+'-'+nom[k]
                else:
                    tab_nom[k+1,W]=tab_nom[k,W]
            else:
                f[k + 1, W] = f[k, W]
                tab_nom[k+1,W]=tab_nom[k,W]
    return f[n,Mmax],tab_nom[n,Mmax][1:]
```

Résolution par approche top-down (mémoïsation)

Nous n'avons pas utilisé ici la technique de mémoïsation pour résoudre le problème. Cette dernière, lorsqu'elle est utilisée, nous permet de moins nous préoccuper de l'ordre de dépendance qui est géré par la récursivité :

```
def sacAdos(v:[int], m:[int], Mmax:int)->int:
    dico = {}
    def f(k, M):
        if (k, M) not in dico: # Si on n'a pas déjà calculé f(k,M)
            if k == 0 or M == 0:  # Cas de base
                x=0
            # On écrit nos 2 relations de récurrence
            elif m[k - 1] <= M:
                x = max(v[k - 1] + f(k - 1, M - m[k - 1]), f(k - 1, M))
            else:
                x = f(k - 1, M)
            dico[(k, M)] = x  #On stocke le résultat dans le dictionnaire de clé
                (k,M)
        return dico[(k, M)]
    return f(len(v), Mmax)
```

4.3 Un autre problème : le rendu de monnaie

4.3.1 Présentation du problème

On souhaite minimiser le nombre de pièces et de billets rendus par un automate qui rend la monnaie tout en garantissant que le total rendu vaut une certaine somme fixée par le problème. La fonction objectif est donc de minimiser la quantité de monnaie rendue tout en respectant la contrainte suivante : la monnaie rendue correspond bien à la quantité d'argent à rendre effectivement.

Hypothèse : On suppose que vous avez un stock illimité de pièces et billets ;

Le système monétaire utilisé est un n-uplet $V = (v_0, v_1, \cdots, v_{n-1})$ où v_i représente la valeur de la $i^{\text{ème}}$ devise. On note $X = (x_0, x_1, \cdots, x_{n-1})$ le n-uplet contenant le nombre de devises rendu. Ainsi, x_i représente le nombre de la $i^{\text{ème}}$ devise rendu. On note S la somme à rendre.

La somme à rendre peut donc être calculée par : $S = \sum_{i=1}^{n} x_i.v_i$.

Q22. Exprimer le problème d'optimisation que l'on cherche à résoudre.

4.3.2 Résolution par algorithme glouton

Dans ce problème, on travaille avec l'ancien système monétaire britannique (avant 1971) s'appuyant sur le penny avec le système monétaire suivant (sans tenir compte des 1/2 pennies)

```
monnaie = [1,3,6,12,24,30,60,240]
```

Q23. À l'aide de l'algorithme glouton, donnez la réponse au problème de rendu de monnaie obtenu pour rendre `somme`= 48 cts ? Quelle est la résolution optimale de ce problème ?

4.3.3 Programmation dynamique [2]

Soit X la somme à rendre, on notera $Nb(X)$ le nombre minimum de pièces à rendre. Nous allons nous poser la question suivante : Si je suis capable de rendre Y avec $Nb(Y)$ pièces, quelle somme suis-je capable de rendre avec $1 + Nb(Y)$ pièces ?

Si j'ai à ma disposition la liste de pièces suivante : v_0, v_1, v_2,$\cdots$, v_{n-1} et que je suis capable de rendre Y cts, je suis donc aussi capable de rendre :

- $Y - v_0$
- $Y - v_1$

2. D'après : https ://pixees.fr/informatiquelycee/n_site/nsi_term_algo_progdyn.html

- $Y - v_2$
- $\cdots$
- $Y - v_{n-1}$

(à condition que v_i (avec $i \in [\![0, n-1]\!]$)) soit inférieure ou égale à la somme restant à rendre.)

Exemple : si je suis capable de rendre 72 cts et que j'ai à ma disposition des pièces de 2 cts, 5 cts, 10 cts, 50 cts et 1 euro, je peux aussi rendre :

- $72 - 2 = 70$ cts
- $72 - 5 = 67$ cts
- $72 - 10 = 62$ cts
- $72 - 50 = 22$ cts
- On ne peut pas utiliser de pièce de 1 euro.

Autrement dit, si $Nb(X - v_i)$ (avec $i \in [\![0, n-1]\!]$) est le nombre minimal de pièces à rendre pour le montant $X - v_i$, alors $Nb(X) = 1 + Nb(X - v_i)$ est le nombre minimal de pièces à rendre pour un montant X. Nous avons donc la formule de récurrence suivante :

$$\begin{array}{l} \text{Si } X = 0 : Nb(X) = 0 \\ \text{Si } X > 0 : Nb(X) = 1 + \min\limits_{0 \leq i \leq n-1 \text{ et } v_i \leq X} Nb(X - v_i) \end{array}$$

4.3.4 Programmation sous Python

```
1  def rendu_monnaie_rec(V,X):
2      if X==0:
3          return 0
4      else:
5          mini = 1000
6      for i in range(len(V)):
7          if V[i]<=X:
8              nb = 1 + rendu_monnaie_rec(V,X-V[i])
9              if nb<mini:
10                 mini = nb
11   return mini
```

Comme vous l'avez sans doute remarqué, pour être sûr de renvoyer le plus petit nombre de pièces, on attribue dans un premier temps la valeur 1000 à la variable `mini` (cette valeur 1000 est arbitraire, il faut juste une valeur suffisamment grande : on peut partir du principe que nous ne rencontrerons jamais de cas où il faudra rendre plus de 1000 pièces), ensuite, à chaque appel récursif, on « sauvegarde » le plus petit nombre de pièces dans cette variable mini. Par contre, on ne sait pas quelles pièces rendre...

TD Info - Programmation dynamique

I - Exercices de mise en jambe

Objectif

L'objectif de ces différents « petits » exercices est de prendre en main la programmation dynamique sur des exemples simples. Les deux orientations de programmation dynamique « bottom-up » et « top-down » sont abordées.

1 Suite récurrente simple

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par $u_n = n \times u_{n-1}$ avec $u_0 = 1$.

Q1. Effectuer une programmation itérative d'une fonction `u_iter` d'argument `n:int` permettant de calculer u_n. Vérifier que vous obtenez $u_{10} = 3\,628\,800$.

Q2. Effectuer une programmation récursive d'une fonction `u_rec` d'argument `n:int` permettant de calculer u_n. Vérifier que vous obtenez $u_{10} = 3\,628\,800$.

Q3. Justifiez que le principe de programmation dynamique ne s'applique pas dans ce cas d'étude (par exemple pas besoin de memoïsation).

2 Suite de Fibonacci

La suite de Fibonacci $(u_n)_{n\in\mathbb{N}}$ est définie de la façon suivante :

$$u_n = u_{n-1} + u_{n-2} \quad \text{avec } n \in \mathbb{N} \text{ et } n > 2 \quad u_0 = 0 \text{ et } u_1 = 1$$

Q4. Effectuer une programmation récursive d'une fonction `fibo` d'argument `n:int` permettant de calculer u_n. Vérifier que vous obtenez $u_{10} = 55$.

Q5. Construire l'arbre d'appels récursifs de `fibo(5)`. Justifier alors que la programmation dynamique est tout à fait indiqué pour résoudre plus efficacement ce problème.

Q6. Effectuer une programmation dynamique `top-down` (principe de mémoïsation) en vous appuyant sur la structure de code proposée. **Dans le cadre d'une mémoïsation, cette structure est à privilégier**. Vérifier que `fibonacci(10)` renvoie bien 55.

```
1  def fibonacci(n:int)->int:
2      tab_fab=[0 for i in range(n+1)] # Initialisation du tableau de mémoïsation
3      tab_fab[1]= ..... # Initialisation utile pour u_1, u_0 est initialisé à 0
4      resultat=memoisation(n,tab_fab) # renvoie le résultat de u_n
5      return resultat
6
7  def memoisation(n,tab_fab):
8      # tab_fab est une liste qui est mise à jour d'appels en appels tout en conservant les valeurs renseignées grâce à l'effet de bord.
9      if n==0:
10         return .....
```

```
    elif n==1:
        return .............
    else :
        if tab_fab[n]!=0:
            return .............
        else:
            resultat= ............. + .............
            tab_fab[n]=.............
            return .............
```

Q7. Reprendre cette question mais en utilisant un dictionnaire pour réaliser la mémoïsation.

Q8. Effectuer maintenant une programmation bottom-up `fibo_bottom_up` afin de calculer u_n. Celle-ci se fera en débutant par le calcul de u_2, puis de u_3 etc. Vous serez attentif à ne pas utiliser un tableau pour stocker les résultats.

II - Découpe de barres d'acier

1 Présentation

Objectif

Un vendeur de matière brute propose la découpe de barre d'acier. Il existe cependant deux conditions :

- *La découpe ne peut se faire que par nombre entier de centimètres ;*
- *Le prix de vente d'un morceau de barre d'acier dépend* ***non linéairement*** *de sa longueur.*

L'objectif du problème étudié est de déterminer le revenu maximum qu'on peut attendre de la vente d'une tige de n centimètres.

Le problème algorithmique posé est le suivant :

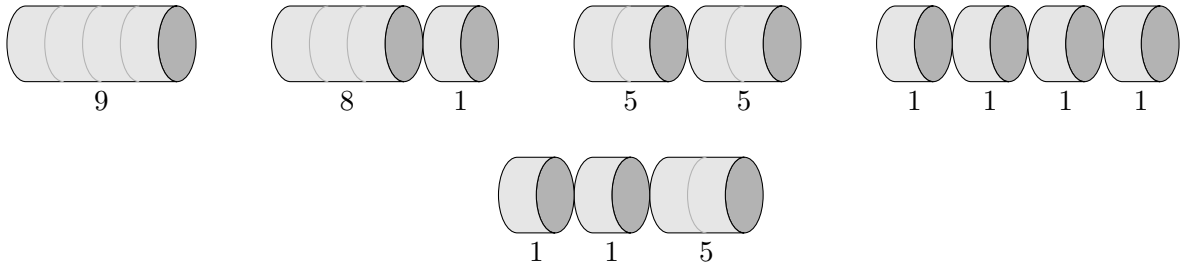
- Entrée : une longueur $n > 0$ et une table de prix P contenant les prix p_i en € d'un morceau de longueur i avec $i \in [\![1, n]\!]$;
- Sortie : Le revenu maximum que l'on peut obtenir pour des tiges de longueur n.

La table des prix P est donnée ci-dessous :

Longueur i	1	2	3	4	5	6	7	8	9	10
Prix p_i en €	1	5	8	9	10	17	17	20	24	30

TABLE 1 – Tableau des prix P

Par exemple, pour une barre de longueur $n = 4$, les **uniques** (donc sans doublons) façons de découper la barre sont :



Ainsi, le meilleur revenu est obtenu en découpant la barre en deux morceaux de taille 2. Le revenu obtenu est 10€.

2 Approche par force brute

L'approche par force brute consiste à tester toutes les possibilités de découpe et de calculer le revenu obtenu pour chacune d'elle. C'est souvent une approche triviale qui peut s'avérer efficace si le problème est de « taille » raisonnable.

Q1. En considérant les doublons possibles, quel est le nombre de possibilités de découpes de la barre de longueur n ?

Q2. Quand bien même il y a des découpes équivalentes possibles, une approche par force brute vous paraît-elle possible pour un nombre n conséquent ?

3 Analyse du problème

Un peu plus formellement, une barre de longueur n peut être découpée en k morceaux. On a alors $n = \ell_1 + \ell_2 + \cdots + \ell_k$ et le revenu associé est $r_n = p_{\ell_1} + p_{\ell_2} + \cdots + p_{\ell_k}$. En dressant une analyse de cas pour différentes longueurs de départ, on obtient les revenus optimaux suivants :

n	r_n	Solution optimale
1	1	1 (pas de découpe)
2	5	2 (pas de découpe)
3	8	3 (pas de découpe)
4	10	$2+2$
5	13	$2+3$
6	17	6 (pas de découpe)
7	18	$1+6$ ou $2+2+3$
8	22	$2+6\cdots$

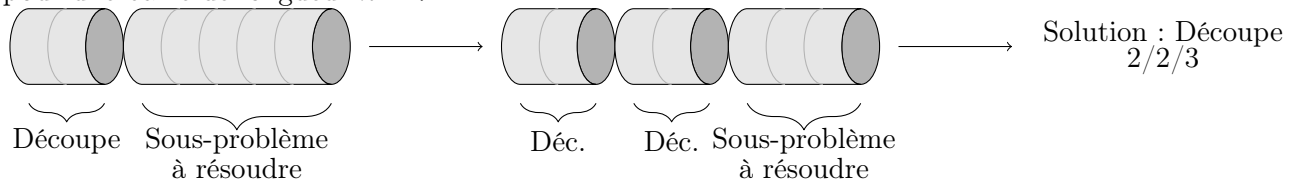
TABLE 2 – Solution du problème pour des tailles n petites

De cette analyse on peut constater qu'il est impossible de poser le choix le plus intéressant au départ pour une solution optimale. Par exemple, dans le cas où $n = 3$, on ne peut pas savoir si le revenu optimal sera obtenu avec une découpe commençant à 1 cm et/ou à 2 cm ou sans découpe. Il sera nécessaire de considérer tous les cas et ne retenir que la solution optimale. On arrive alors à la relation ci-dessous pour une barre de longueur n au départ.

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \cdots, r_{n-1} + r_1\right)$$

On constate que cette version est assez complexe à résoudre car, il faut résoudre pour chaque élément deux problèmes : r_{n-p} et r_p avec $p \in [\![1, n-1]\!]$. On peut simplifier cette approche :

- Toute solution optimale à une découpe la plus à gauche qui ne sera pas redécoupée. Par exemple pour une barre de longueur $n = 7$.



- On peut calculer r, en considérant toutes les tailles pour la première découpe et en combinant avec le découpage optimal pour la partie à droite ;
- Pour chaque cas, on n'a donc qu'à résoudre un seul sous-problème (au lieu de deux), celui du découpage de la partie droite ;

Q3. En supposant que $r_0 = 0$, écrire alors le problème r_n sous sa nouvelle forme.

Q4. On pose la liste `p=[0,1,5,8,9,10,17,17,20,24,30]`, le tableau des prix. Établir la programmation d'une fonction `couper_barre` d'arguments `p` et d'un entier `n`, longueur de la barre avec `n`$\leqslant$`len(p)-1` permettant de résoudre le problème de découpe proposé.

Q5. Modifier votre fonction, pour compter le nombre d'appels récursifs. Tester que votre code fonctionne en vérifiant que vous obtenez le bon résultat de revenu maximal pour des barres de longueur

différentes. Pour $n = 4$ et $n = 10$, combien d'appels sont effectués ? Effectuer une conjecture sur le nombre d'appels à la fonction à réaliser pour obtenir la solution du problème.

Q6. Pour $n = 4$, dessiner l'arbre d'appels récursifs et justifier qu'une programmation dynamique est envisageable.

4 Programmation dynamique top-down : mémoïsation

On va utiliser deux fonctions pour résoudre ce problème. Une fonction principale et une fonction auxiliaire qui fera le calcul et utilisera un tableau `r` pour gérer le processus de mémoïsation.

```
1  p=[0,1,5,8,9,10,17,17,20,24,30]
2  def couper_barre(p:[int],n:int)->int:
3      r=[-1 for i in range(len(p))] # Initialisation du tableau qui contiendra la
           résolution du problème pour r[i]
4      r[0]=0 # Pour r_0=0.
5      return memoisation_coupe_barre(p,n,r)
6
7  def memoisation_coupe_barre(p,n,r):
8      # Si la solution est connue
9      if r[n]>=0:
10         return ..... # A compléter
11 # Cas de base du problème récursif
12     if n==0:
13         return ..... # A compléter
14     # Cas récursif
15     else:
16         rev=..... # A compléter
17         for i in range(1,n+1):
18             rev=..... # A compléter
19     r[n]=..... # A compléter
20     return ..... # A compléter
```

Q7. Compléter le code permettant de résoudre le problème. Que contient la liste `r` ?

Q8. Modifier votre code pour savoir combien d'appels à la fonction `memoisation_coupe_barre`.

5 Programmation dynamique bottom-up

L'approche que l'on va mettre en place est ici différente, on va d'abord résoudre les problèmes les plus simples r_0, r_1, etc. Après la résolution de ces problèmes « élémentaires », on peut construire des solutions plus complexes. La structure de code utilisée est la suivante :

```
1  p=[0,1,5,8,9,10,17,17,20,24,30]
2  def couper_barre(p:[int],n:int)->int:
3      r=[-1 for i in range(len(p))] # Initialisation du tableau qui contiendra la
           résolution du problème pour r[i]
4      r[0]=0 # Pour r_0=0.
5      for j in range(1,n+1):
6          rev=..... # Initialisation
7          for i in range(1,.....): # A compléter
8              rev=..... # A compléter
9          r[j]=..... # A compléter
10     return ..... # A compléter
```

Q9. Compléter le code. Vérifier les résultats obtenus.

III - Le meilleur intervalle

Objectif

Dans cet exercice, on va chercher à mettre en place une relation de récurrence permettant de résoudre un problème simple à comprendre. On va également s'intéresser à l'optimisation du coût mémoire de celui-ci.

On dispose d'un tableau `T` fixe, de taille n. Il ne contient que des valeurs réelles positives. Il existe (au moins) deux indices i et j, définissant l'intervalle $[i, j]$ avec $0 \leqslant i \leqslant j \leqslant n-1$, tel que la valeur `T[j]`$-$`T[i]` (appelé écart) est maximale. Par exemple, si `T=[9,15,10,12,8,18,20,7]`, le meilleur écart est de valeur 12. Il est unique et obtenu pour $i = 4$ et $j = 6$.

Remarque : Si le tableau est monotone décroissant cette valeur est nulle et correspond à n'importe quel intervalle de type $[i, i]$ pour $0 \leqslant i \leqslant n-1$.

Q1. On appelle `vmi(k)` la valeur du (d'un) meilleur écart se terminant *exactement* en position `k`. Quelles sont les valeurs de `vmi(0)`, `vmi(1)`, $\cdots$, `vmi(7)` sur l'exemple précédent ?

Q2. Préciser comment obtenir la valeur du (d'un) meilleur écart du tableau `T` à partir de `vmi(0)`, `vmi(1)`, $\cdots$, `vmi(n-1)`.

Q3. Établir une relation de récurrence permettant le calcul de `vmi(k)`. Pour cela, vous rechercherez notamment le lien entre `vmi(k)` et `vmi(k-1)`.

Q4. Une approche top-down ou bottom-up est-elle nécessaire pour déterminer de manière efficace la valeur de `vmi(k)` pour un k fixé ?

Q5. Une approche top-down ou bottom-up est-elle envisageable afin de réponde au problème initial, à savoir : déterminer le meilleur écart `T[j]`$-$`T[i]`. Proposer celle qui offre la meilleure complexité spatiale et temporelle en justifiant votre réponse.

Q6. Établir alors un programme permettant de déterminer la valeur du meilleur écart sur l'exemple proposé `T=[9,15,10,12,8,18,20,7]`.

Q7. Modifier votre code pour connaître non seulement la valeur du meilleur écart mais également l'intervalle $[i, j]$ correspondant à cette valeur.

IV - Postes de montage industriel

Objectif

Dans une usine automobile, un atelier a deux chaînes de montage (voir FIGURE 10*). Un châssis arrive sur chaque chaîne, 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. L'objectif est de déterminer la façon de faire transiter le châssis de poste en poste et éventuellement en le faisant passer entre les deux chaînes de montage pour réaliser l'assemblage le plus rapide.*

1 Présentation

Chaque chaîne de montage a n stations, numérotées $j = 1, 2, \cdots, n$. On représente le $j^{\text{ème}}$ poste de la chaîne i (avec i valant 1 ou 2) par $S_{i,j}$. Le $j^{\text{ème}}$ poste de la chaîne 1 ($S_{1,j}$) fait le même travail que le $j^{\text{ème}}$ poste de la chaîne 2 ($S_{2,j}$). 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 fonctionnement identiques mais situés sur des chaînes différentes. Le temps de montage au poste $S_{i,j}$ est $a_{i,j}$. Comme le montre la FIGURE 10, un châssis arrive au poste 1 de l'une des chaînes,

puis passe de poste en poste. On a aussi 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. 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 $j = 1, 2, \cdots, n-1$ (car, après le $n^{\text{ème}}$ 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.

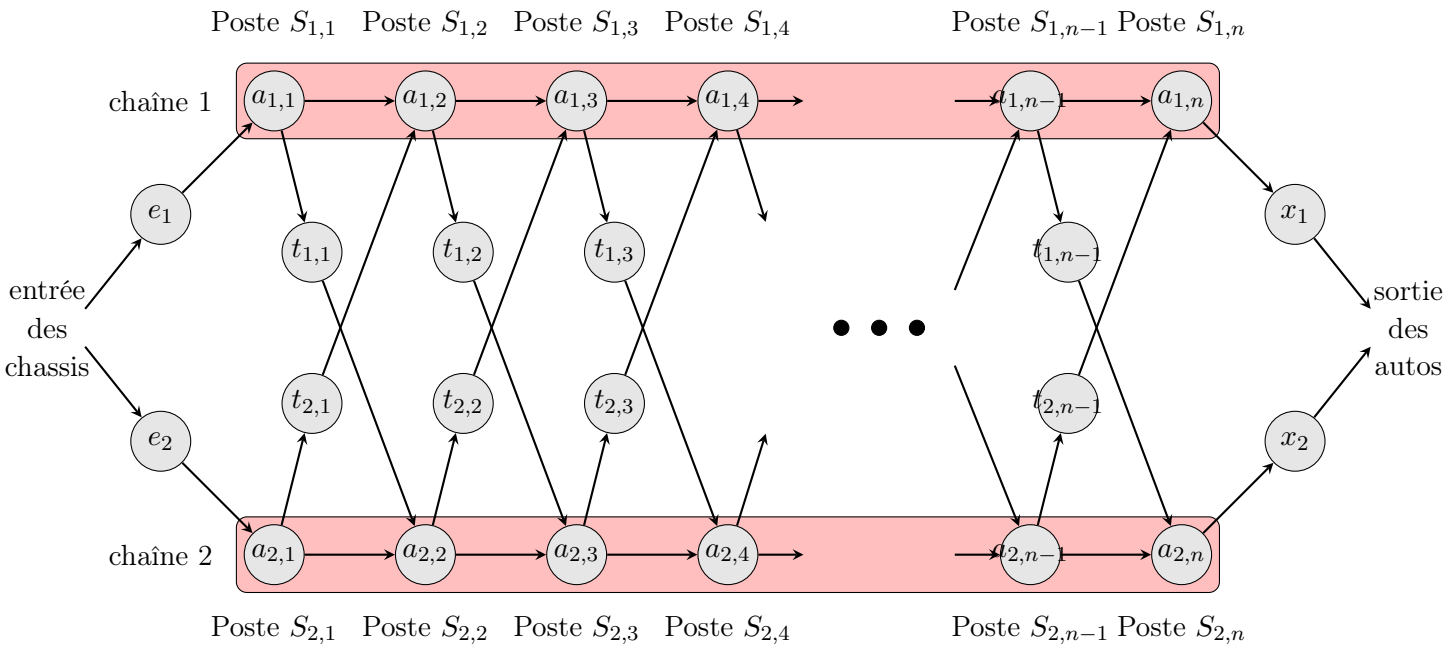


FIGURE 10 – Caractéristiques générales des chaînes de montage

2 Problème à traiter

Le problème traité est le suivant :

Q1. En supposant qu'il y ait n postes de travail sur les chaînes de montage, déterminer le nombre de possibilités de choisir les postes de travail pour réaliser le montage des automobiles. Une approche par force brute est-elle envisageable pour trouver la solution optimale ?

3 Formulation du problème à résoudre

Notre objectif ultime est de déterminer le délai le plus court par lequel un châssis traverse tout l'atelier, délai que nous noterons f^*. Le châssis doit aller au poste n de l'une ou l'autre des chaînes 1 et 2, et de là aller vers la sortie de l'atelier. On note $f_i[j]$ le délai le plus court possible avec lequel le châssis va du point de départ et est traité au poste $S_{i,j}$.

Q2. Exprimer f^* en fonction de $f_1[n]$, $f_2[n]$, x_1 et x_2.

Q3. Exprimer $f_1[1]$ et $f_2[1]$ en fonction respectivement de e_1, $a_{1,1}$ et e_2, $a_{2,1}$.

Voyons maintenant comment calculer $f_i[j]$ pour $j = [\![2, n]\!]$. En nous focalisant sur $f_1[j]$, rappelons-nous que le chemin optimal passant par le poste $S_{1,j}$ est soit le chemin optimal passant par le poste

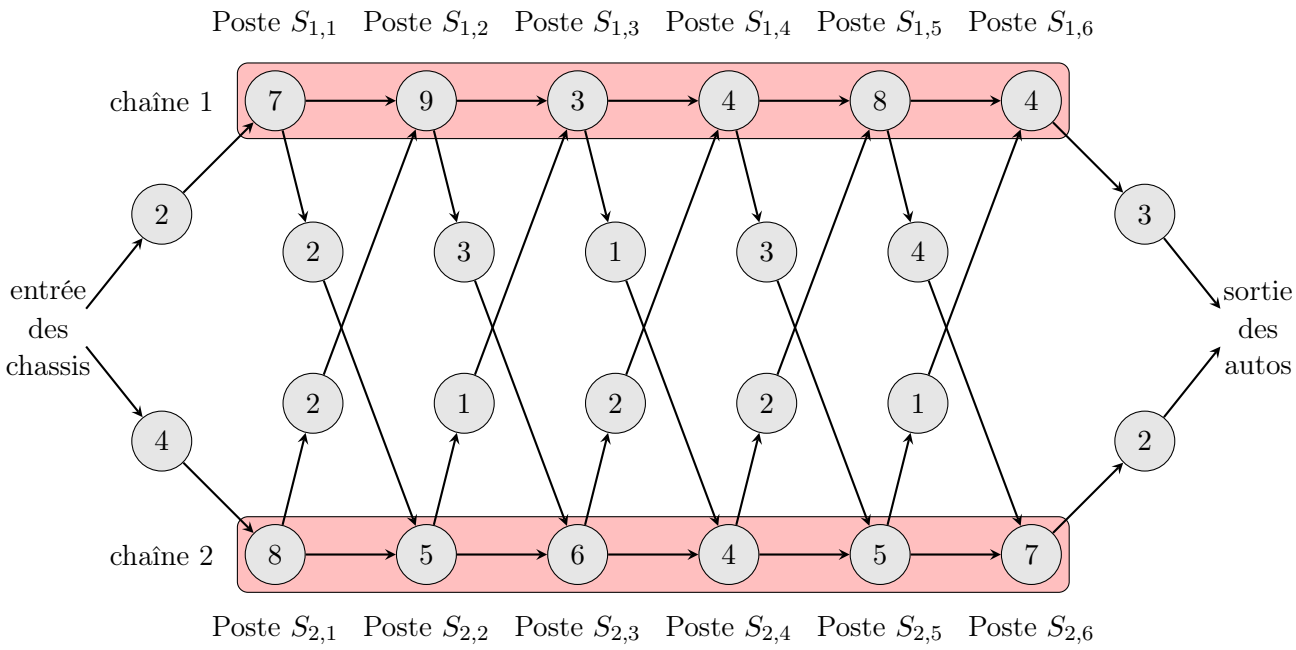


FIGURE 11 – Problème traité

$S_{1,j-1}$ suivi du passage direct au poste $S_{1,j}$, soit le chemin optimal passant par le poste $S_{2,j-1}$ suivi d'un transfert de la chaîne 2 à la chaîne 1 et suivi enfin du passage au poste $S_{1,j}$.

Q4. Exprimer alors la relation entre $f_1[j]$, $f_1[j-1]$, $a_{1,j}$, $f_2[j-1]$, $t_{2,j-1}$. Faire de même pour $f_2[j]$.

Q5. Donner alors les deux relations de récurrence pour $j = [\![1, n]\!]$ traduisant le calcul de $f_1[j]$ et $f_2[j]$.

4 Programmation récursive

La formulation du problème obtenue au paragraphe précédent, permet donc de réaliser une programmation récursive afin de déterminer f^*. Pour réaliser le codage de cette programmation récursive, nous définissons les listes suivantes et les variables suivantes :

```
1 e1,e2,x1,x2=2,4,3,2 # Valeur des coûts d'entrée et de sortie des chaînes de
      montage.
2 a1=[7,9,3,4,8,4] # Temps de passage sur les postes de montage de la chaîne 1.
3 a2=[8,5,6,4,5,7] # Temps de passage sur les postes de montage de la chaîne 2.
4 t1=[2,3,1,3,4] # Temps de transfert des postes de la chaîne 1 vers la chaîne 2.

5 t2=[2,1,2,2,1] # Temps de transfert des postes de la chaîne 2 vers la chaîne 1.
```

Q6. Établir une programmation récursive (en utilisant potentiellement deux fonctions récursives) afin de déterminer $f_1[n]$ et $f_2[n]$.

Q7. Sur l'exemple traité, quelles sont alors les valeurs de $f_1[6]$ et $f_2[6]$?

Q8. Établir alors le code permettant de calculer f^*.

Q9. Mettre en place un compteur (ou deux compteurs) permettant de connaître le nombre d'appels effectués à la (ou les) fonction(s) récursive(s). Que peut-on conjecturer sur le nombre d'appels nécessaires aux fonctions récursives pour résoudre le problème ?

5 Programmation dynamique par mémoïsation : Approche top-down

On constate que le nombre d'appels récursifs est important. Des appels à des calculs identiques se

fait de manière conséquente. Une programmation dynamique est donc toute indiquée pour améliorer l'efficacité du code. On propose ici une approche par mémoïsation.

Q10. Effectuer une programmation dynamique par mémoïsation du problème des postes de montage. Vérifier que vous obtenez bien le même résultat que précédemment pour f^*. N'hésitez pas à utiliser deux fonctions de mémoïsation pour résoudre le problème au besoin (une pour le calcul de f_1 et l'autre pour celui de f_2).

6 Programmation dynamique : approche bottom-up

Q11. Effectuer une programmation dynamique bottom-up pour résoudre ce problème. On précise qu'il n'est pas indispensable de créer un tableau contenant les différentes valeurs de f_1 et f_2.

On va maintenant chercher à reconstruire la solution pour savoir sur quel poste doit passer le châssis automobile pour que le montage soit réalisé le plus rapidement possible. Mais avant cela, cherchons à déterminer manuellement la solution.

On donne les deux tableaux suivants qui seront à compléter :

j	1	2	3	4	5	6
$f_1[j]$	9	18				
$f_2[j]$	12	16				

j	2	3	4	5	6
$\ell_1[j]$	1	2			
$\ell_2[j]$	1	2			

TABLE 3 – Le premier tableau représente les durées les plus courtes pour terminer les opérations d'assemblage sur chaque poste de chaque chaîne. Le second tableau renseigne sur la chaîne du poste précédent au poste j courant.

Q12. Sachant que $\ell_1[j]$ contient le numéro de chaîne précédente pour réaliser $f_1[j]$ et de même pour $\ell_2[j]$. Compléter les deux tableaux. En déduire la solution au problème posé.

Q13. Modifier votre programme de programmation dynamique bottom_up pour construire les deux tableaux ℓ_1 et ℓ_2. Votre fonction renverra la valeur de f^*, la chaîne par laquelle l'automobile sort, les tableaux ℓ_1 et ℓ_2