Informatique : Programmation dynamique

PSI : Lycée Rabelais

Pré-requis

Cours d'informatique de première année : algorithmes dichotomiques, fonctions récursives, algorithmes gloutons.

Objectifs

Situer la programmation dynamique parmi d'autres stratégies de résolution d'un problème.

Acquérir des notions de programmation dynamique : sous-structure optimale, chevauchement de sous-problèmes.

Mettre en œuvre les principes de la programmation dynamique : déterminer des sous-problèmes, approche descendante avec récursivité ou ascendante sans récursivité, utilisation d'un dictionnaire pour mémoriser les résultats intermédiaires (mémoïsation)

1 Notions de programmation dynamique

1.1 Stratégies de résolution d'un problème

La résolution d'un problème donné (tri d'une liste, recherche d'un élément dans une liste ou une chaîne de caractère, parcours d'un graphe...) peut s'effectuer de différentes manières :

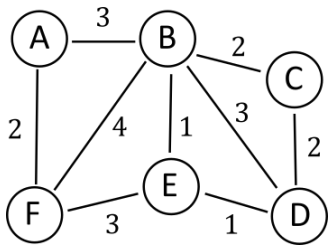
- la **force brute** : l'ensemble des solutions possibles du problème sont établies avant de choisir la meilleure, ce qui engendre des problèmes de coût temporel et de coût en mémoire ;
- la **stratégie gloutonne** : le problème est divisé en sous-problèmes du même type, et l'on ne retient que la meilleure solution à chaque sous-problème traité successivement. La solution finale est la combinaison de ces solutions, mais elle n'est pas toujours optimale ;
- la **stratégie "diviser pour régner"** : le problème est divisé en sous-problèmes nécessairement indépendants. Ils sont traités séparément, puis leurs solutions combinées pour fournir la solution au problème posé (exemple : tri d'une liste par dichotomie).

Ces stratégies de résolution permettent souvent de répondre au problème posé, mais pas toujours de manière optimale. *Optimiser un problème* revient à maximiser ou minimiser une fonction "objectifs" tout en respectant un ensemble de "contraintes". Par exemple, la recherche du plus court chemin dans un graphe est un problème d'optimisation.

La programmation dynamique est une autre stratégie de résolution située entre la stratégie gloutonne et la force brute : elle consiste à séparer le problème en plusieurs sous-problèmes, à chercher la solution optimale à chaque problème, et ainsi de suite jusqu'à construire la réponse optimale. À chaque étape de la résolution, l'ensemble des solutions au sous-problème est établi, et la solution finale est celle qui remplit les contraintes fixées.

1.2 Un exemple : plus court chemin d'un graphe

On cherche le plus court chemin entre les sommets *A* et *D* du graphe représenté ci-dessous :



Réponse :

Mais quelle stratégie de résolution adoptée pour obtenir cette réponse ?

Stratégie de la force brute : elle consiste à examiner l'ensemble des chemins possibles (A-B-C-D, A-B-E-D, A-F-B-E-D etc.), puis à déterminer lequel est le plus court. Pour un graphe de petite taille, cette stratégie est applicable. Elle cesse cependant d'être envisageable dès que le nombre de sommets et d'arêtes du graphe augmente.

Stratégie gloutonne : elle consiste à démarrer de A, puis à déterminer quel chemin est le plus court pour aller au point suivant. Il s'agit ici du chemin A-F, qui vaut 2. On recommence depuis F, et ainsi de suite : le chemin est construit au fur et à mesure de l'exploration du graphe, en décomposant le problème principal en plusieurs sous-problèmes dont on fournit la solution optimale (chemin le plus court). À la fin, on obtient le chemin A-F-E-D, de longueur 6... qui n'est pas le chemin le plus court pour aller de A à D ! **On a optimisé chaque sous-problème, mais pas le problème global.**

Programmation dynamique : on reprend l'idée de la stratégie gloutonne : on examine les chemins possibles depuis un sommet, mais au lieu de déterminer lequel est le plus court, on explore les chemins suivants jusqu'à résoudre le problème. Le principe de base est le suivant : si un chemin optimal de A à D passe par un sommet X alors le chemin emprunté de A à X est optimal, et le chemin emprunté de X à D aussi. Dans ce cas, on peut soit démarrer du début (sommet A) ou de la fin (sommet D). Si l'on démarre de A, voici les étapes successives du déroulement de l'algorithme :

• *Etape 1* : deux choix possibles : A-B (longueur 3) et A-F (longueur 2). Contrairement à la stratégie gloutonne, l'algorithme va explorer ces deux voies, et pas uniquement la plus courte ;

• *Etape 2* : depuis B, quatre chemins sont possibles. Le chemin A-B-F (longueur 7) permet de rejoindre F depuis A, *mais il est plus long que le chemin A-F déjà exploré* : le chemin A-B-F ne permet pas de lier A à F de manière optimale, ce début de solution est donc rejeté. De même, pour rejoindre B depuis A, le chemin A-B est à privilégier, et non le chemin A-F-B. On remarque de plus que deux voies sont possibles pour atteindre E, mais la plus courte est A-B-E : le chemin A-F-E ne sera donc pas retenu.

• *Etape 3* : le sommet D a été atteint à l'étape 2 (chemin A-B-D de longueur 6), mais rien n'indique que ce chemin est le plus court. Après élimination des chemins A-B-F, A-F-B et A-F-E, il reste à explorer A-B-C et A-B-E : A-B-C-D est de longueur 7, A-B-E-D de longueur 5. Par comparaison, le chemin A-B-E-D est le plus court de l'ensemble des chemins retenus comme solutions intermédiaires : *A-B-E-D est le chemin le plus court du graphe permettant de lier A à D*.

En cherchant le plus court chemin entre A et D, on a obtenu les plus courts chemins entre A et n'importe quel autre sommet. Pour répondre au problème initial, nous avons résolu un problème plus général que nous avons décomposé en plusieurs sous-problèmes.

1.3 Quand choisir la programmation dynamique pour résoudre un problème ?

La programmation dynamique est envisagée si :

• le problème peut-être résolu à partir de sous-problèmes similaires mais plus petits. La solution optimale au problème posé est obtenue à partir des solutions optimales des sous-problèmes : **la programmation dynamique possède la propriété de sous-structure optimale** ;

• ces sous-problèmes ne sont cependant pas indépendants les uns des autres : ils sont répétés lors de la résolution, ou un appel récursif les résout plusieurs fois au lieu de générer de nouveaux sous-problèmes. **La programmation dynamique présente un chevauchement des sous-problèmes**. ;

• l'ensemble des sous-problèmes traités est discret, c'est-à-dire qu'ils peuvent être indexés et leurs résultats mémorisés.

Deux stratégies sont alors envisageables pour finalement résoudre le problème :

• l'approche de haut en bas (ou descendante) : on effectue **des appels récursifs** jusqu'à parvenir à des cas de base dont la solution est connue. Ces appels peuvent nécessiter de traiter plusieurs fois le même sous-problème. Pour diminuer la complexité temporelle, on mémorise les résultats de chaque sous-problème traité dans un objet (dictionnaire, liste...) : c'est la **mémoïsation** ;

• l'approche de bas en haut (ou ascendante) : on part des petits sous-problèmes et on **itère les calculs** en stockant les résultats intermédiaires dans une liste (structure de boucle). Cette stratégie entraîne un coût en espace mais une possibilité de gain si on ne doit pas tout mémoriser.

Conclusion : stratégie de résolution de problème en programmation dynamique

La méthode générale pour résoudre un problème en programmation dynamique est la suivante :

• définir les sous-problèmes (exemple précédent : déterminer les plus courts chemins entre sommets) ;

• évaluer les différentes possibilités ;

• établir des liens entre les solutions des sous-problèmes ;

• mémoriser les résultats intermédiaires afin de les comparer entre eux : on peut utiliser, dans une approche descendante, la récursivité avec mémoïsation pour la résolution des sous-problèmes ou utiliser, dans une approche ascendante, une boucle avec construction d'une table pour stocker les résultats.

2 Construction descendante ou ascendante : exemple de la suite de Fibonacci

Nous allons chercher à calculer le n-ième terme de la suite de Fibonacci de manière optimale, soit par une approche ascendante, soit par une approche descendante.

Suite de Fibonacci

Le n-ième terme de la suite de Fibonacci est calculé par la relation de récurrence $f_n = f_{n-2} + f_{n-1}$, avec $f_0 = 0$ et $f_1 = 1$.

2.1 Préliminaire : résolution par fonction récursive sans mémorisation

Construction de la fonction : une manière simple de répondre à ce problème est de définir une fonction récursive comme la suivante :

```
def fibo_rec(n):
    if ................: # cas de base
    ................
    else:
        ............................... # calcul du terme d'ordre n par
     recursivite
    return f
```

Étudions les étapes de calcul lorqu' on appelle cette fonction pour, par exemple, $n = 4$:

• `fibo_rec(4)` : $n = 4$ est différent de 0 ou 1 : on doit calculer `fibo_rec(3)` et `fibo_rec(2)` ;

• calcul de `fibo_rec(2)` : il faut calculer `fibo_rec(1)` et `fibo_rec(0)`. Ces valeurs sont connues : ce sont les cas de base de la fonction. Le programme dépile et renvoie la valeur de `fibo_rec(2)`, soit $0+1=1$;

• calcul de `fibo_rec(3)` : il faut calculer `fibo_rec(2)` et `fibo_rec(1)`. Ce dernier est connu (cas de base). `fibo_rec(2)` est calculée à partir de `fibo_rec(1)` et `fibo_rec(0)`.

• une fois l'ensemble des calculs effectués, et les cas de base retrouvés, le programme dépile pour donner `fibo_rec(2)`$= 0+1=1$ et `fibo_rec(3)` $= (0+1)+1=2$, soit $f = 3$.

Complexité en temps de `fibo_rec(n)` : si $\mathscr{C}(n)$ est le coût pour calculer f_n, alors $\mathscr{C}(n) = \mathscr{C}(n-2)+\mathscr{C}(n-1)+1$ (le $+1$ provient de l'addition). Cela revient à écrire $\mathscr{C}(n) \geq 2\mathscr{C}(n-2)$: la complexité est exponentielle !

À titre d'exemple, le calcul de f_4 demande à calculer deux fois `fibo_rec(2)` , malgré la connaissance du résultat dès le premier calcul. Pour des termes d'ordre plus élevés, cette redondance est inacceptable car très coûteuse en temps de calcul. Ainsi, il dévient déjà difficile de calculer f_8 par cette méthode car il faut calculer 13 fois f_2 ! Pour éviter ce problème, on peut enregistrer les résultats de ces calculs intermédiaires et les utiliser dès que nécessaire : c'est le principe de la mémoïsation.

2.2 Approche descendante : calcul de f_n par récursivité et mémoïsation

Pour retenir les résultats intermédiaires, nous allons utiliser un dictionnaire noté `memo` et initialement vide : `memo = {}`. Ce dictionnaire peut être défini à l'extérieur de la fonction (variable globale) :

```
memo={}

def fibo_memo(n):
# on commence par verifier si la cle "n", dont la valeur associee est fibo_memo(n), est deja presente dans le dictionnaire
    if ..... :
        return memo[n]
    if ..... : # cas de base
        .....
    else:
        ..... # calcul du terme d'ordre n par recursivite
        ..... # le dictionnaire memo contient la valeur fibo_memo(n) associee a la cle n.
    return f
```

Essais : • `fibo_memo(4)` renvoie : .....

• `fibo_mem(50)` renvoie : .....

Le temps de calcul est réduit : `fibo_memo(50)` donne un résultat de manière presque instantané, contrairement à `fibo_rec(50)`. L'explication tient au fait que les appels à `memo` sont de complexité $\mathcal{O}(1)$, donc obtenir f_n est de complexité $\mathcal{O}(n)$, et non plus exponentielle. Si la clé n n'est pas présente dans le dictionnaire, elle est rajoutée à la fin de celui-ci avec la valeur f_n associée.

Dans cette approche : programmation dynamique = récursion + mémoïsation.

Remarque : certains auteurs définissent le dictionnaire `memo` comme paramètre par défaut dans la fonction : `def fibo_memo(n,memo={}):`. Dans ce cas, `memo` est modifié à chaque appel de la fonction, même dès le premier appel, et reste en mémoire dans le code de la fonction lors des appels suivants. Cette ecriture est à manipuler avec beaucoup de précaution lorsque le paramètre par défaut est mutable.

2.3 Approche ascendante par itération (boucle) et liste

On supprime la récursivité pour la remplacer par une boucle. Les sous-problèmes sont traités en premier, la solution du problème s'obtient par combinaison des solutions aux sous-problèmes. Les résultats peuvent être stockés dans un dictionnaire ou dans une liste.

```
def fibo_asc(n):
  L=[]
    for ..... in .....: # la boucle balaie tous les indices k jusqu'a n
        if .....:
            f=..... # cas de base
        else:
            f=L[k-2]+L[k-1] # calcul du terme d'ordre n a partir des deux valeurs de L precedemment calculees
        ..... # la liste L contient la valeur fibo_asc(n) a l'indice n.
    return f
```

On peut remarquer que la structure de la fonction est en tout point similaire à celle de l'approche descendante : condition `if...else...`, calcul de `f`. La seule exception est la définition de la liste `L` : c'est une variable locale et non globale (i.e., définie en dehors de la fonction), le processus d'itération le permettant, contrairement à la récursivité.

Conclusion : stratégie de résolution du problème "Calcul de f_n"

- définir les sous-problèmes : calcul de f_k pour k allant de 0 à n ;
- établir des liens entre les solutions des sous-problèmes : $f_k = f_{k-2} + f_{k-1}$;
- enregistrer les résultats des sous-problèmes, soit en utilisant la récursivité avec mémoïsation (approche descendante), soit en utilisant une boucle avec construction d'une table pour stocker les résultats (approche ascendante).

3 Dernier exemple : problème du rendu de monnaie

3.1 Résolution par un algorithme glouton

Les algorithmes gloutons ont été abordés en 1^{re} année. Un exemple classique pour illustrer cette approche est celui du rendu de monnaie dont nous rappelons l'énoncé ci-dessous.

Problème du rendu de monnaie

On suppose donné un système monétaire où les valeurs faciales des pièces (ou des billets) sont rangées en ordre décroissant. On cherche à payer la somme s indiquée en utilisant un nombre minimal de pièces. On suppose que l'on possède autant de pièces de chaque valeur que nécessaire.

Exemple : système monétaire `euros` : en ne prenant pas en compte les centimes, et en utilisant les valeurs les plus courantes (billet de 50€ en valeur maximale), le système monétaire `euros` s'écrit : `euros =[50,20,10,5,2,1]`. Pour payer $s = 48$€, on pourrait donner 48 pièces de 1€, ou 24 pièces de 2€... La contrainte imposée ici est de rendre le moins de pièces possible : le programme doit donc renvoyer "2 × 20€, 1 × 5€, 1 × 2€ et 1 × 1€", soit 5 valeurs en tout.

Construction de l'algorithme glouton : une manière de répondre à la question posée est d'optimiser chacun des sous-problèmes suivants : "combien de pièces de la valeur maximale je peux donner pour m'approcher de la somme s demandée ?" Si `v_max` est la valeur maximale en question, et `N` le nombre de pièces, alors la réponse est "`N` × `v_max` tant que `N` × `v_max< s`". Une fois ce sous-problème résolu, on réitère pour la somme restante, à savoir `s - N` × `v_max`.

En appliquant au système monétaire `euros`, avec $s = 48$:

- on commence avec `v_max` $= 20 < 48$: 48-2× 20 =8 < `v_max` : on doit donc rendre 2 billets de 20€, la somme restante est 8 ;
- `v_max` $= 10 > 8$: aucun billet de 10€ ;
- `v_max` $= 5 < 8$: 8-1 × 5 =3 < `v_max` : on doit rendre 1 billet de 5€, la somme restante est 3 ;

• v_max = 2 < 3 : 3-1 × 2 =1 < v_max : on doit rendre 1 pièce de 2€, la somme restante est 1 ;

• v_max = 1 = 1 : 1-1 × 1 =0 : on doit rendre 1 pièce de 1€

La somme finale est nulle, le problème est résolu. Le résultat est bien 5 espèces à rendre, comme analyser dans l'exemple. L'algorithme glouton utilisé ici a optimisé les résultats de chaque sous-problème, et le résultat final est lui-même optimisé, ce qui n'était pas initialement demandé.

Écriture de la fonction `monnaie_glouton` : la fonction `monnaie_glouton` prend en argument la liste des valeurs monétaires possibles du système étudié (`systeme`) et la somme à rendre représentée par l'entier `somme`.

```
euros = [50, 20, 10, 5, 2, 1]
def monnaie_glouton(systeme, somme):
    i = 0 # indice de la piece qu'on va essayer
    p = len(systeme) # nombre de valeurs de pieces disponibles
    monnaie = [] # liste des pieces rendues
    while i<p and somme>0:
        if systeme[i]>somme:
            i = i+1
        else:
            monnaie.append(systeme[i])
            somme = somme - systeme[i]
    if somme==0:
        return monnaie, len(monnaie)
```

Ainsi `monnaie_glouton(euros,48)` renvoie `[20,20,5,2,1],5`.

Limite de l'algorithme glouton : considérons l'ancien système monétaire britannique, le système impérial : `imperial = [30,24,12,6,3,1]`. Pour payer 48, l'algorithme glouton va rendre : "1 pièce de 30, 1 pièce de 12, 1 pièce de 6", soit 3 valeurs en tout. Mais il aurait été plus efficace de rendre 2 pièces de 24 ! En voulant optimiser le sous-problème "combien de pièces de 30 je dois rendre pour m'approcher de 48 ?", l'algorithme ne permet pas de vérifier que 24 est plus profitable... C'est ici qu'intervient la programmation dynamique !

3.2 Amélioration de l'algorithme par programmation dynamique

La réponse optimale est celle qui utilise le nombre minimal de pièces. On a vu que l'algorithme glouton échoue à la trouver en général (l'exemple de rendre 48 avec le système impérial suffit à le prouver).

Définition du sous-problème : pour rendre une somme s, on peut rendre une pièce quelconque p et il reste alors à rendre la somme $s-p$. Si l'on sait rendre toute somme strictement inférieure à s de manière optimale, il suffit de tester les différentes possibilités de rendre $s-p$ pour toutes les pièces p du système monétaire et choisir la meilleure, c'est-à-dire celle qui minimise le nombre de pièces à rendre.

Une approche récursive est naturelle pour résoudre le problème : si on appelle $f(s)$ le nombre minimal de pièces qu'il faut utiliser pour payer la somme s, on a :

• $f(0) = 0$: si la somme est nulle, aucune pièce n'est à rendre ;

• $f(s) = \min_{p \leq s}(1 + f(s-p))$: pour une somme s donnée, on calcule le nombre de pièces à rendre pour toute valeur de pièce p inférieure ou égale à s. On recommence avec la somme restante, $s-p$, jusqu'à aboutir au cas de base $f(0) = 0$.

1ère approche : récursivité sans mémoïsation :
Pour comparer les résultats entre eux, et choisir le résultat associé au minimum de pièce à rendre, on utilise la fonction `min(x1,x2)` de `Python` qui renvoie le minimum entre les deux nombres `x1` et `x2`. La variable `mini` est initialisée à `inf`, c'est-à-dire l'infini (en fait, un nombre flottant très grand). C'est nécessaire pour effectuer la première application de `min(...)` dans la boucle `for`.

```
def dynRecursif(systeme, somme):
    if somme == 0:
        return 0
    mini = inf
    for x in systeme:
        if somme >= x:
            mini = min(mini, 1 + dynRecursif(systeme, somme - x))
    return mini
```

Sans mise en mémoire des résultats des sous-problèmes, le programme met trop longtemps à aboutir. On exploite la mémoïsation (approche descendante) ou la mise en mémoire dans une table (approche ascendante) pour diminuer la complexité temporelle.

Amélioration : récursivité avec mémoïsation (approche descendante) :
On enregistre les résultats des sous-problèmes traités dans un dictionnaire. Chaque clé est une somme s, chaque valeur associée le nombre minimal de pièces à rendre. On initialise le dictionnaire par `memo_monnaie={0:0}` : si la somme est nulle (clé $= 0$), le nombre de pièce à rendre est nul aussi (valeur associée $= 0$).

```
memo_monnaie={0:0}
def dynRecursif_memoise(systeme, somme):
    if somme in memo_monnaie:
        return memo_monnaie[somme]
    mini = inf
    for x in systeme:
        if somme >= x:
            mini = min(mini, 1 + dynRecursif_memoise(systeme, somme - x))
            memo_monnaie[somme]=mini
    return mini
```

Autre possibilité : itération (approche ascendante) :
Dans ce cas, la liste `f` contient le nombre minimal de pièces à rendre pour chaque somme intermédiaire à calculer, c'est-à-dire de 0 à `somme` par pas de 1. Elle est définie dans le corps de la fonction.

```
def dynAsc_Memoise(systeme, somme):
    f = [0] * (somme + 1)
    for s in range(1, somme + 1):
        f[s] = inf
        for x in systeme:
            if s >= x:
                f[s] = min(f[s], 1 + f[s - x])
    print(f)
    return f[somme]
```