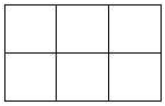
Programmation dynamique

Introduction

On dispose de la grille 2×3 ci-dessous.



Question : Combien de chemins mènent du coin supérieur gauche au coin inférieur droit, en se déplaçant uniquement le long des traits horizontaux vers la droite et le long des traits verticaux vers le bas ? Et pour une grille 10×10 ?

Principe de la programmation dynamique

La **programmation dynamique** est une technique dûe à Richard Bellman dans les années 1950. À l'origine, cette méthode algorithmique était utilisée pour résoudre des problèmes d'optimisation. L'idée générale est de déterminer un résultat sur la base de calculs précédents.

Plus précisément, la programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits au plus grands **en stockant les résultats intermédiaires**.

> Le terme *programmation* désigne la *planification*, et n'a pas de rapport avec les langages de programmation.

Nous allons illustrer le principe de la programmation dynamique sur le calcul du n-ième terme de la suite de Fibonacci.

La suite de Fibonacci

Rappels sur la suite de Fibonacci

On a déjà abordé cette suite lorsque nous avons parlé de la programmation récursive (Thème 4, Chapitre 1). Mais voilà quelques rappels :

La suite de Fibonnacci est une suite de nombres dont chacun est la somme des deux précédents. Le premier et le second nombres sont égaux à 0 et 1 respectivement. On obtient la suite de nombres : 0 - 1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - ...

Mathématiquement, cette suite notée (F_n) est donc définie par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \text{ pour tout entier } n \geqslant 2 \end{cases}$$

Version récursive naïve (et inefficace)

Vous avez déjà programmé une version récursive qui renvoie le terme de rang n de cette suite.

In [1]:
```
def fibo(n):
    """Version récursive naïve"""
    if n <= 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Par exemple, voici l'arbre des appels récursifs si on lance `fibo(5)` .

```
                        fib(5)
                  /               \
              fib(4)               fib(3)
            /       \             /      \
        fib(3)      fib(2)       fib(2)   fib(1)
       /    \       /    \       /     \
  fib(2)   fib(1)  fib(1) fib(0) fib(1) fib(0)
  /     \
fib(1) fib(0)
```

On se rend compte qu'il y a beaucoup d'appels redondants : `fib(1)` a été lancé 5 fois, `fib(2)` a été lancé 3 fois, etc.

Ces redondances entraînent un nombre d'appels récursifs qui explose rapidement dès que n est élevé. Par conséquent, les temps de calcul deviennent vite très élevés. Pire, dès que n est trop grand, l'algorithme ne donnera jamais la réponse.

In [2]:
```
%time fibo(5)
```

```
Wall time: 0 ns
```

Out[2]:
```
5
```

In [3]:
```
%time fibo(30)
```

```
Wall time: 367 ms
```

Out[3]:
```
832040
```

In [4]:
```
%time fibo(37)
```

```
Wall time: 12.9 s
```

Out[4]:
```
24157817
```

> Il est possible de faire mieux, en évitant de refaire les calculs déjà effectués. Pour cela, il faut **stocker les résultats intermédiaires** !

Version récursive avec *mémoïsation*

Une première approche est d'adapter l'algorithme récursif en stockant les résultats calculés dans un tableau ou un dictionnaire. Lors d'un appel, on commence par vérifier si on ne connaît pas déjà la réponse, auquel cas on la renvoie directement, ce qui évite d'effectuer des calculs redondants.

Cela donne la fonction `fibo_memo` suivante qui prend en paramètres un entier `n` et un dictionnaire `memo` que l'on met à jour en stockant les résultats intermédiaires au fur et à mesure.

In [5]:
```
def fibo_memo(n, memo):
    if n in memo:      # si calcul déjà effectué
        return memo[n] # on renvoie directement sa valeur
    elif n <= 1:
        memo[n] = n
        return memo[n]
    else:
        memo[n] = fibo_memo(n-1, memo) + fibo_memo(n-2, memo)
        return memo[n]
```

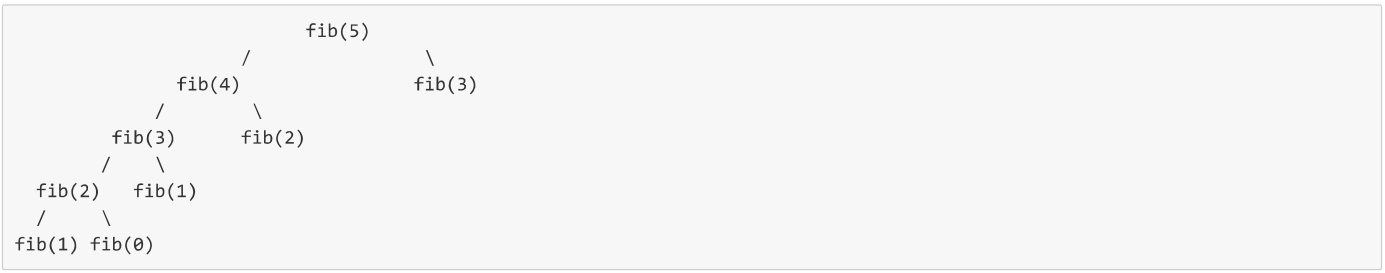
Explications :

- Lignes 2 et 3 : si la valeur `n` est déjà dans le dictionnaire, c'est qu'on a déjà calculé F_n et il suffit alors de renvoyer sa valeur `mem[n]` (la valeur associée à `n`).
- Lignes 4 à 9 : quasiment identiques à la version récursive naïve à ceci près que l'on mémorise la valeur dans le dictionnaire `memo` avant de la renvoyer
- De cette façon, dès qu'une valeur F_n a été calculée, elle est ajoutée dans le dictionnaire comme la valeur associée à n, ce qui permet de la réutiliser directement dès qu'on en a besoin.

Il n'y a plus qu'à lancer le premier appel avec un dictionnaire vide, c'est ce que fait la fonction `fibo` suivante.

In [6]:
```python
def fibo(n):
    """Version récursive avec mémoïsation"""
    F = {}
    return fibo_memo(n, F)
```

Avec ce procédé de mémoïsation, l'arbre des appels est considérablement réduit puisqu'il n'y a plus aucun appel redondant. Par exemple, l'arbre des appels récursifs en lançant `fibo(5)` se réduit à :



On constate alors qu'avec cette version, les valeurs F_n sont calculées quasiment instantanément et que l'on peut obtenir les valeurs F_n pour des grandes valeurs de n.

In [7]:
```
%time fibo(5)
```

```
Wall time: 0 ns
```

Out[7]:
```
5
```

In [8]:
```
%time fibo(30)
```

```
Wall time: 0 ns
```

Out[8]:
```
832040
```

In [9]:
```
%time fibo(37)
```

```
Wall time: 0 ns
```

Out[9]:
```
24157817
```

In [10]:
```
%time fibo(850)
```

```
Wall time: 2.04 ms
```

Out[10]:
```
1949885951587339044875793733356219760673772926586260591210358470405525665390243100575540781157408285819450131557173898 14
```

Méthode *descendante*

La version récursive avec mémoïsation correspond à une approche *descendante*, aussi appelée *haut-bas* (ou *top-down* en anglais). En effet, pour connaître F_n on lance l'appel `fibo(n)` qui déclenche la descente d'appels récursifs jusqu'aux cas de base pour lesquels on mémorise les résultats. Dans un second temps, on remonte les appels tout en mémorisant leurs résultats pour ne pas résoudre plusieurs fois le même problème.

Finalement, avec cette méthode, c'est lors de la remontée des appels que leurs résultats sont mémorisés puis réutilisés sur les problèmes plus grands. On peut alors se demander si on ne peut pas procéder directement du plus petit sous-problème au plus grand (celui que l'on veut résoudre). La réponse est oui ! Et on explique cela de suite.

Version itérative *ascendante*

> On parle aussi de méthode *bas-haut*, ou *bottom-up* en anglais.

Une autre manière de résoudre le problème est d'utiliser une approche *ascendante*.

Il s'agit d'une méthode **itérative** dans laquelle on commence par calculer des solutions pour les sous-problèmes les plus petis puis, de proche en proche, on arrivera à la taille voulue. Comme précédemment, on utilise le principe de la mémoïsation pour stocker les résultats partiels.

Le calcul du terme F_n de la suite de la Fibonacci n'est pas un problème d'optimisation, ainsi le calcul d'une solution d'un problème à partir des solutions connues des sous-problèmes est simple puisqu'il n'y a aucun choix à faire.

De manière générale, on utilise un tableau pour stocker les résultats au fur et à mesure. Voici les étapes habituelles :

1. **Création et initialisation du tableau** :
   - On a besoin d'un tableau `F` de taille $n+1$ qui va contenir les valeurs F_0, F_1, ..., F_n dans cet ordre
   - Pour cela on crée le tableau `F` avec $n+1$ zéros initialement
   - On peut stocker les valeurs déjà connues (F_0 et F_1 dans notre cas)
2. **Utilisation de la formule de récurrence pour remplir le reste du tableau** :
   - La formule de récurrence donne la solution d'un sous-problème à partir de celles de sous-problèmes plus petits et donc déjà traités ! Ici on a pour $2 \leq i \leq n$: $F_i = F_{i-1} + F_{i-2}$
   - On peut donc remplir le tableau `F` en parcourant les indices **par ordre croissant** : on va mettre dans `F[i]` la valeur `F[i-1] + F[i-2]` que l'on connaît puisque ces deux valeurs ont été calculés précédemment.
3. **Le résultat est dans la dernier case du tableau** : on la renvoie !

Cela donne la fonction suivante.

In [11]:
```python
def fibo(n):
    """Version itérative ascendante"""
    F = [0] * (n+1)
    F[0] = 0  # pas indispensable car déjà initialisé à 0
    F[1] = 1
    for i in range(2, n + 1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

Les performances sont semblables à la version récursive avec mémoïsation

In [12]:
```python
%time fibo(850)
```

```
Wall time: 0 ns
```

Out[12]:
```
1949885951587339044875793733356219760673772926586260591210358470405525665390243100575540781157408285819450131557173898 14
```

Autres problèmes

Il existe de nombreux problèmes pouvant être résolus avec le paradigme de la programmation dynamique, dont beaucoup de problèmes d'optimisation :

- Problème du rendu de monnaie (traité en exercice)
- Problème du sac-à-dos (version gloutonne abordée en classe de Première)
- Alignement de séquences (traité en exercice)
- Problème du plus court chemin (Algorithme de Bellman-Ford utilisé par le protocole RIP)
- Problèmes d'ordonnancement d'intervalles pondérés (voir ici)
- Toutes sortes de problème d'affectation des ressources
- etc.

> Passez aux exercices !

Le programme officiel cite les exemples du rendu de monnaie et de l'alignement de séquences comme des exemples pouvant être présentés. Il sont détaillés en exercices mais voici tout de même un résumé condensé qui pourra servir de révisions.

Rendu de monnaie

On a vu en Première que l'on peut résoudre ce problème avec un algorithme glouton mais que ce dernier ne fournit pas nécessairement une solution optimale (et parfois fausse selon le système monétaire utilisé).

Une autre approche est d'utiliser la force brute en testant toutes les combinaisons possibles. On peut le faire de manière récursive mais le nombre de combinaisons à tester devient vite trop important pour que cette solution soit satisfaisante.

Relation de récurrence

On peut chercher à exprimer le problème (rendre une somme n) à partir de sous-problèmes plus petits (rendre une somme plus petite que n). On aboutit à la relation de récurrence suivante sur le nombre de pièces optimal pour rendre une somme n :

$$\text{nb\_pieces}(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + \min\limits_{p \leq n}(\text{nb\_pieces}(n-p)) & \text{sinon} \end{cases}$$

En effet, l'idée est la suivante : pour rendre la somme n de façon optimale il faut rendre une pièce p est la somme $n-p$ de façon optimale. Pour cela, il faut tester toutes les pièces p possibles et prendre celle qui minimise le nombre de pièces pour rendre pour la somme $n-p$ (d'où la recherche du minimum)

Version récursive avec mémoïsation

L'algorithme récursif classique a une efficacité catastrophique car le nombre d'appels explose car il y a beaucoup de redondances. On peut améliorer cette version récursive en utilisant la programmation dynamique : pour cela on stocke les résultats connus dans un tableau `memo` et on renvoie directement les résultats connus pour éviter les appels récursifs redondants.

In [13]:
```python
def rendu_monnaie(s, pieces):
    """Version récursive avec mémoïsation"""
    memo = [None] * (s + 1) # intialisation du tableau avec des None
    return rendu_monnaie_memo(s, pieces, memo)

def rendu_monnaie_memo(s, pieces, memo):
    if memo[s] is not None: # si on a déjà calculé le nombre optimal de pièces pour rendre la somme s
        return memo[s]      # on le renvoie directement
    elif s == 0:
        memo[s] = 0
        return 0
    else:
        nb_pieces = s     # nb_pieces = 1 + 1 + ... + 1 dans le pire des cas
        for p in pieces:
            if p <= s:   # inutile de tester une pièce dont la valeur dépasse la somme s à rendre
                nb_pieces_bis = 1 + rendu_monnaie_memo(s - p, pieces, memo)
                nb_pieces = min(nb_pieces, nb_pieces_bis)
        memo[s] = nb_pieces
        return nb_pieces

# ESSAI
rendu_monnaie(100, [1, 2])
```

Out[13]: 50

Version itérative ascendante

On procède classiquement :

1. **Création et initialisation du tableau** :
   - On a besoin d'un tableau `nb` de taille $s+1$ qui va permettre de stocker les valeurs $\mathrm{nb\_pieces}(0)$, $\mathrm{nb\_pieces}(1)$, ..., $\mathrm{nb\_pieces}(s)$ dans cet ordre.
   - Pour cela on crée le tableau `nb` avec $n+1$ zéros initialement
   - La valeur connue `nb[0]` est alors déjà correctement initialisée
2. **Utilisation de la formule de récurrence pour remplir le reste du tableau** :
   - Formule de récurrence :

$$\mathrm{nb}[n] = 1 + \min_{p \leq n} (\mathrm{nb}[n-p])$$

   - On peut donc remplir le tableau `nb` en parcourant les indices **par ordre croissant** (on fait varier `n` de `1` à `s`) en utilisant la formule de récurrence
3. **Le résultat est dans la dernier case du tableau** : on la renvoie !

Cela donne la fonction suivante.

In [14]:
```python
def rendu_monnaie(s, pieces):
    """Version itérative ascendante"""

    # ÉTAPE 1 : création et initialisation du tableau
    nb = [0] * (s+1)    # nb[0] est ainsi bien initialisé

    # ÉTAPE 2 : remplissage du reste du tableau par indice croissant
    for n in range(1, s + 1):
        nb[n] = n       # nb[n] = 1 + 1 + ... + dans le pire des cas
        for p in pieces:
            if p <= n:
                nb_bis = 1 + nb[n-p]
                nb[n] = min(nb[n], nb_bis)

    # ÉTAPE 3 : le résultat est dans la dernière case
    return nb[n]

# ESSAI
rendu_monnaie(100, [1, 2])
```

Out[14]:
```
50
```

Alignement de séquences

Décontextualisation : On peut représenter deux séquences comme deux chaînes de caractères T_1 et T_2 et l'objectif est d'étudier leur degré de similarité (ou dissimilarité). Pour cela on peut chercher à *minimiser* la *distance* $d(T_1, T_2)$ entre les deux chaînes.

Pour passer d'une chaîne à l'autre on dispose de 3 opérations chacune ayant un coût égal à 1 :

- insertion (d'un caractère c dans T_1)
- suppression (d'un caractère c dans T_1)
- substitution (d'un caractère c par un autre caractère c' dans T_1)

On appelle **distance d'édition**, notée dE, le nombre minimal de caractères qu'il faut insérer, supprimer, ou substituer pour passer d'une chaîne à l'autre, c'est-à-dire celle qui minimise le coût des alignements possibles. La distance d'édition permet donc d'obtenir un aligment optimal.

Exemple : Un alignement optimal des chaînes SUCCES et ECHECS est

```
S U C C E - S
- E C H E C S
```

dont le coût est égal à 4, donc $\mathrm{dE}(\mathrm{SUCCES}, \mathrm{ECHECS}) = 4$

Relation de récurrence

On peut dégager une relation de récurrence car le calcul de la distance d'édition entre deux chaînes peut se faire à partir de celles de chaînes plus petites.

On a la relation suivante :

$$\mathrm{dE}(T_1[1..i], T_2[1..j]) = \min \begin{cases} \mathrm{dE}(T_1[1..i-1], T_2[1..j]) + 1 \\ \mathrm{dE}(T_1[1..i], T_2[1..j-1]) + 1 \\ \mathrm{dE}(T_1[1..i-1], T_2[1..j-1]) + \mathrm{sub}(T_1[i], T_2[j]) \end{cases}$$

où $\mathrm{sub}(a, b) = \begin{cases} 0 \text{ si } a = b \text{ (correspondance)} \\ 1 \text{ si } a \neq b \text{ (substitution)} \end{cases}$

En effet, pour aligner de manière optimale $T_1[1..i]$ et $T_2[1..j]$, il y a trois cas de figure :

- $T_1[i]$ s'aligne sur « - ». Cela demande à avoir aligné $T_1[1..i-1]$ et $T_2[1..j]$ de manière optimale et supprimé $T_1[i]$.
- « - » s'aligne sur $T_2[j]$. Cela demande à avoir aligné $T_1[1..i]$ et $T_2[1..j-1]$ de manière optimale et inséré $T_2[j]$.
- $T_1[i]$ s'aligne sur $T_2[j]$:
  - si $T_1[i] \neq T_2[j]$, cela demande à avoir aligné $T_1[1..i-1]$ et $T_2[1..j-1]$ de manière optimale et substitué $T_1[i]$ par $T_2[j]$.
  - si $T_1[i] = T_2[j]$. Cela demande d'avoir aligné $T_1[1..i-1]$ et $T_2[1..j-1]$ de manière optimale et c'est tout (sans surcoût)

> La version récursive naïve permettant de calculer la distance d'édition a un coût en temps catastrophique car les appels sont redondants. On peut améliorer cela à l'aide de la technique de mémoïsation comme pour les autres exemples mais on ne présente dans la suite que la version itérative ascendante.

Version itérative ascendante

On note n_1 et n_2 les longueurs respectives de T_1 et T_2.

On procède classiquement :

1. **Création et initialisation du tableau** :
   - On a besoin d'un tableau `D` à deux dimensions de taille $(n1+1) \times (n2+1)$ qui va permettre de stocker en position (i, j) la distance d'édition entre les i premiers caractères de T_1 et les j premiers caractères T_2 (`D[i][j]` $= \mathrm{dE}(T_1[1..i], T_2[1..j])$).
   - Pour cela on crée le tableau `D` avec uniquement des zéros
   - On peut remplir la première ligne et la première colonne facilement
   - À ce stade on obtient le tableau suivant dans lequel on n'a pas représenté les 0 dans les cases vides puisqu'ils seront écrasés au fur et à mesure du remplissage

		0	1	2	3	4	5	6
		ε	E	C	H	E	C	S
0	ε	0	1	2	3	4	5	6
1	S	1						
2	U	2						
3	C	3						
4	C	4						
5	E	5						
6	S	6						

2. **Utilisation de la formule de récurrence pour remplir le reste du tableau** :
   - Formule de récurrence :

$$\mathrm{nb[i][j]} = \min \begin{cases} \mathrm{nb[i\text{-}1][j]} + 1 \text{ (suppression)} \\ \mathrm{nb[i][j\text{-}1]} + 1 \text{ (insertion)} \\ \mathrm{nb[i\text{-}1][j\text{-}1]} + \begin{cases} 0 \text{ si } T_1[i] = T_2[j] \text{ (correspondance)} \\ 1 \text{ si } T_1[i] \neq T_2[j] \text{ (substitution)} \end{cases} \end{cases}$$

   - On peut donc remplir le tableau `nb` avec deux boucles imbriquées qui parcourent les indices **par ordre croissant** (on fait varier i de 1 à n_1 et j de 1 à n_2) en utilisant la formule de récurrence.
3. **Le résultat est dans la dernier case du tableau** : on la renvoie !

Cela donne la fonction suivante.

In [15]:
```python
def dE(t1, t2):
    """Version itérative ascendante"""

    # ÉTAPE 1 : création et initialisation du tableau
    n1 = len(t1)
    n2 = len(t2)
    D = [[0] * (n2 + 1) for i in range(n1 + 1)]
    for i in range(n1 + 1):  # remplissage première colonne
        D[i][0] = i
    for j in range(n2 + 1):  # remplissage première ligne
        D[0][j] = j

    # ÉTAPE 2 : remplissage du reste du tableau par indice croissant
    for i in range(1, n1 + 1):
        for j in range(1, n2 + 1):
            if t1[i-1] == t2[j-1]:  # attention ! le k-ième caractère d'une chaîne est en position k-1
                cout_sub = 0  # correspondance
            else:
                cout_sub = 1  # substitution
            D[i][j] = min(D[i-1][j] + 1, D[i][j-1] + 1, D[i-1][j-1] + cout_sub)

    # ÉTAPE 3 : le résultat est dans la dernière case
    return D[n1][n2], D   # on renvoie aussi le tableau D pour l'observer

# ESSAI
dE('SUCCES', 'ECHECS')
```

Out[15]:
```
(4,
 [[0, 1, 2, 3, 4, 5, 6],
  [1, 1, 2, 3, 4, 5, 5],
  [2, 2, 2, 3, 4, 5, 6],
  [3, 3, 2, 3, 4, 4, 5],
  [4, 4, 3, 3, 4, 4, 5],
  [5, 4, 4, 4, 3, 4, 5],
  [6, 5, 5, 5, 4, 4, 4]])
```

Contrairement à la version récursive naïve, on peut obtenir les distances entre deux chaînes assez longues.

In [16]:
```python
dE('ALGORITHME', 'ALGORIHTME')
```

Out[16]:
```
(2,
 [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
  [1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
  [2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8],
  [3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7],
  [4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6],
  [5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5],
  [6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4],
  [7, 6, 5, 4, 3, 2, 1, 1, 1, 2, 3],
  [8, 7, 6, 5, 4, 3, 2, 1, 2, 2, 3],
  [9, 8, 7, 6, 5, 4, 3, 2, 2, 2, 3],
  [10, 9, 8, 7, 6, 5, 4, 3, 3, 3, 2]])
```

Alignement optimal

Pour trouver un alignement optimal entre T_1 et T_2 il suffit de *remonter* depuis `D[n1][n2]` vers `D[0][0]` (de la case en bas à droite à celle en haut à gauche) en considérant à chaque fois le "meilleur" des 3 voisins, c'est-à-dire celui par lequel on a pu arriver ! (il peut parfois y en avoir plusieurs).

- Si on monte, il s'agit d'une suppression
- Si on va à gauche, il s'agit d'une insertion
- Si on va en diagonale, il s'agit d'une substitution ou d'une correspondance

Par exemple, le tableau `D` correspondant à l'alignement entre SUCCES et ECHECS est :

		0	1	2	3	4	5	6
		ε	E	C	H	E	C	S
0	ε	0	1	2	3	4	5	6
1	S	1	1	2	3	4	5	5
2	U	2	2	2	3	4	5	6
3	C	3	3	2	3	4	4	5
4	C	4	4	3	3	4	4	5
5	E	5	4	4	4	3	4	5
6	S	6	5	5	5	4	4	4

Suppression

Insertion

Substitution/Correspondance

← $\mathrm{dE}(T_1, T_2)$ est ici !

On peut alors construire un alignement optimal (en partant de la fin ou du début) :

```
S U C C E - S
- E C H E C S
```

Conclusion

- La **programmation dynamique** est une technique permettant d'améliorer l'efficacité d'un algorithme en *évitant les calculs redondants*.
- Pour cela, on utilise un tableau (ou un dictionnaire) pour stocker les résultats intermédiaires et pouvoir les réutiliser sans les recalculer.
- Comme la méthode « diviser pour régner », la programmation dynamique permet résoudre un problème à partir des solutions de sous-problèmes. Si ces derniers se « chevauchent » (s'ils sont non indépendants) alors la programmation dynamique permettra d'éviter que les appels récursifs ne soient effectués plusieurs fois. Ainsi, la programmation dynamique permet souvent d'améliorer des algorithmes récursifs.
- Pour utiliser la programmation dynamique, on procède généralement ainsi :
  1. définition des sous-problèmes
  2. identification d'une relation de récurrence les solutions des sous-problèmes
  3. mise en place d'un algorithme récursif avec mémoïsation ou d'un algorithme itératif ascendant
  4. résolution du problème original à partir des solutions des sous-problèmes
- La programmation dynamique permet de résoudre de manière efficace de nombreux problèmes d'optimisation, comme le rendu de monnaie ou l'alignement de séquences, pour lesquels une solution récursive classique est inefficace.

Références :

- Equipe pédagogique DIU EIL, Université de Nantes.
- Articles Wikipédia sur la Programmation dynamique et sur la Distance de Levensthein
- Livre *Spécialité Numérique et sciences informatiques : 24 leçons avec exercices corrigés - Terminale*, éditions Ellipses, T. Balabonski, S. Conchon, J.-C. Filliâtre, K. Nguyen. Site du livre : http://www.nsi-terminale.fr/ pour l'idée de l'exercice 2 notamment.
- Cours de David Roche sur le Pixees : Programmation dynamique
- Un cours sur la programmation dynamique : http://www-desir.lip6.fr/~spanjaard/pmwiki/uploads/ProgrammationDynamique.pdf

---

Germain BECKER, Lycée Mounier, ANGERS

CC BY NC SA