ITC2

Programmation dynamique

C. MULLAERT

Lycée Saint-Louis

Année 2024-2025

Plan du cours

Introduction

Calcul du terme général de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ définie par

$$F_0 = 0, \quad F_1 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad F_{n+2} = F_n + F_{n+1}$$

Une approche récursive naïve

```
def fibo(n):
    if n <= 1:
        return n
    return fibo(n-1) + fibo(n-2)
```

a une complexité exponentielle qui peut être améliorée en ne calculant pas inutilement plusieurs fois les mêmes termes.

En utilisant le même principe que dans une récurrence double

```
def fibo(n):
    a=0
    b=1
    for k in range(n):
        c=a+b
        a=b
        b=c
    return a
```

on obtient une complexité linéaire.

En utilisant le même principe que dans une récurrence forte

```
def fibo(n):
    if n == 0:
        return 0
    L = [0,1]
    for k in range(n-1):
        L.append(L[-1] + L[-2])
    return L[-1]
```

on obtient une complexité linéaire temporelle et une complexité spatiale linéaire...

L'idée (inutile ici) consiste à stocker les valeurs déjà calculées. Ici on le fait en commençant par les premiers termes de la suite (approche du bas vers le haut).

Une autre technique, la mémoïsation, est une approche de haut en bas.

```
def fibo(n):
    L=[None]*(n+1)
    def aux(k) :
        if L[k]!=None :
            return L[k]
        if k<=1 :
            f=k
        else :
            f=aux(k-1)+aux(k-2)
        L[k]=f
        return f
    return aux(n)
```

On utilise ici le caractère mutable des listes. On peut aussi utiliser une structure de dictionnaire

```
def fibo(n):
    d={}
    def aux(k) :
        if k in d :
            return d[k]
        if k<=1 :
            d[k]=k
            return k
        r=aux(k-1)+aux(k-2)
        d[k]=r
        return r
    return aux(n)
```

Rappels sur les dictionnaires

Le type dictionnaire permet de stocker des éléments mais contrairement à une liste, on n'y accède pas par un indice mais par une clé.

On déclare un dictionnaire à l'aide d'accolades :

```
d={}     #crée un dictionnaire vide
d={"a":4, 8:[7], (2,3):"c"}
         #crée un dictionnaire à 3 éléments.
```

Les clés peuvent être des entiers, des flottants, des chaînes de caractères, des t-uples mais pas des listes.

Si on effectue les instructions

```
for x in d :
    print(x)
```

ou

```
for x in d.keys() :
    print(x)
```

on affiche les clés

Pour accéder à l'ensemble des couples clé-valeur, on peut utiliser la méthode **items** :

```
for x in d.items() :
    print(x)
```

Comme pour les listes, on dispose de **len**.

On peut modifier un élément déjà présent :

```
d["a"] = "modif"
```

ou en rajouter un :

```
d[10] = "nouvellevaleur"
```

ce qui va impliquer les mêmes soucis au niveau des "copies".

Comparez **d2** et **d3** après les instructions suivantes :

```
d2=d
d3=dict(d)
d[0]="+"
```

Pour calculer le terme général de la suite de Fibonacci, on peut aussi réécrire la relation matriciellement :

$$\forall n \in \mathbb{N}, \quad \begin{pmatrix} f_{n+2} \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix}$$

et donc

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

et utiliser l'exponentiation rapide.

```
def fibo(n):
  def puissance(n):
      if n==0 :
          return 1,0,0,1
      if n%2==0 :
          a,b,c,d=puissance(n//2)
          return a*a+b*c, a*b+b*d, c*a+d*c, c*b+d*d
      else :
          a,b,c,d=puissance(n//2)
          A,B,C,D= a*a+b*c, a*b+b*d, c*a+d*c, c*b+d*d
          return C,D,A+C,B+D
  a,b,c,d=puissance(n)
  return b
```

On a alors un nombre d'opérations élémentaires logarithmique.

L'idée de stocker dans une liste ou un dictionnaires les valeurs déjà calculées est plus utile dans le cas où une récurrence forte s'impose.

Calcul du terme général de la suite des nombres de Catalan définie par :

$$C_0 = 1 \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad C_{n+1} = \sum_{k=0}^{n} C_k C_{n-k}$$

Une approche récursive naïve

```
def catalan(n):
    if n == 0:
        return 1
    somme = 0
    for k in range(n):
        somme +=  catalan(k)*catalan(n-1-k)
    return somme
```

a une complexité exponentielle.

Approche de bas en haut itérative :

```
def catalan(n):
    L = [1]
    for p in range(1,n+1):
        somme = 0
        for k in range(p):
            somme += L[k]*L[p-1-k]
        L.append(somme)
    return L[-1]
```

Approche de haut en bas :

```
def catalan(n):
    d={}
    def aux(k) :
        if k in d :
            return d[k]
        if k==0 :
            d[k]=1
            return 1
        s=0
        for i in range(k):
            s+=aux(i)*aux(k-1-i)
        d[k]=s
        return s
    return aux(n)
```

Calcul d'un coefficient binomial à l'aide de la relation de Pascal

Une approche récursive naïve donne

```
def binom(k,n):
    if n == 0:
        return k==0
    return binom(k-1,n-1)+binom(k,n-1)
```

Dans ce cas, le calcul de certains coefficients est inutilement long ($k = 0$, $k > n$, $k = n$).

La version :

```
def binom(k,n):
    if k>n :
        return 0
    if k == 0 or k==n :
        return 1
    return binom(k-1,n-1)+binom(k,n-1)
```

est plus efficace mais reste exponentielle pour le calcul des termes "centraux" comme $\binom{n}{2n}$.

Pour éviter les calculs inutiles on va utiliser la programmation dynamique.

Approche de bas en haut :

```
def binom(k,n) :
    T=[(k+1)*[0] for i in range (n+1)]
      # création d'un tableau que l'on va remplir
      # de bas en haut et de gauche à droite
    for m in range(n+1):
        T[m][0]=1
        for l in range(1,k+1):
              T[m][l]=T[m-1][l]+T[m-1][l-1]
    return T[n][k]
```

Chaque coefficient binomial n'est calculé qu'une fois mais l'on calcule beaucoup de coefficients binomiaux inutiles (surtout si $k > n$)...

Une idée d'amélioration ne conserver que les couples (ℓ, m) tels que $m - \ell \leq n - k$

```
def binom(k,n) :
    if k>n :
        return 0
    T=[(k+1)*[0] for i in range (n+1)]
    for m in range(n+1):
        T[m][0]=1
        for l in range(m-n+k,k+1):
            T[m][l]=T[m-1][l]+T[m-1][l-1]
    return T[n][k]
```

malheureusement ce code est faux...

Celui-ci est juste :

```
def binom(k,n) :
    if k>n :
        return 0
    T=[(k+1)*[0] for i in range (n+1)]
    for m in range(n+1):
        T[m][0]=1
        for l in range(max(1,m-n+k),k+1):
            T[m][l]=T[m-1][l]+T[m-1][l-1]
    return T[n][k]
```

L'approche de haut en bas limite la réflexion : seuls les coefficients binomiaux intermédiaires utiles seront calculés

```
def binom(k,n):
    d={}
    def aux(l,m):
        if (l,m) in d :
            return d[(l,m)]
        if l>m :
            b=0
        if l==0 or l==m :
            b=1
        else :
            b=aux(l-1,m-1)+aux(l,m-1)
        d[(l,m)]=b
        return b
    return aux(k,n)
```

En reprenant l'approche de bas en haut (et en réfléchissant), on peut avoir une complexité spatiale linéaire en ne stockant que deux lignes :

```
def binom(k,n) :
    if k>n :
        return 0
    T1=[1]+k*[0]
    T2=[1]+k*[0]
    for m in range(n):
        for l in range(1,k+1):
            T2[l]=T1[l]+T1[l-1]
        T1=T2[:]
    return T1[k]
```

ou une seule :

```
def binom(k,n) :
    if k>n :
        return 0
    T=[1]+k*[0]
    for m in range(n):
        for l in range(k):
            T[k-l]=T[k-l]+T[k-l-1]
    return T[k]
```

Généralités

• Pour trouver une solution (optimale) à un problème on peut se ramener à la recherche de solutions (optimale) de sous-problèmes puis en déduire la solution (optimale). On dit qu'un tel problème à une *propriété de sous-structure optimale*.
C'est le cas dans la recherche dichotomique d'un élément dans une liste triée, le tri fusion, la recherche du plus court chemin d'un sommet à un autre dans un graphe, le nombre de façons de placer des parenthèses dans une expression, le calcul du terme général d'une suite définie par récurrence....

• Lorsque les problèmes sont indépendants (tri fusion, recherche dichotomique,...), l'approche récursive est efficace. Lorsqu'il y a *chevauchement des sous-problèmes*, il est préférable de stocker les valeurs déjà calculées. Ce stockage peut se faire avec une liste ou un dictionnaire.

• On peut utiliser une méthode *ascendante* dite *de bas en haut*, généralement de type itératif, qui consiste à partir des petits sous-problèmes et à monter progressivement en difficulté jusqu'à atteindre la valeur recherchée.

• On peut utiliser une méthode *descendante* dite *de haut en bas*, généralement de type récursif, dans laquelle on cherche d'abord à résoudre les problèmes de grande taille, et que l'on fait pour cela appel à des problèmes de plus petite taille. À chaque fois que l'on trouve la solution à un sous problème on le stocke et à chaque fois que l'on a besoin de la solution à un sous-problème, on vérifie si le calcul n'a pas déjà été fait. On parle de *mémoïsation*.

• Dans ces cas, on gagne en complexité temporelle par rapport à l'algorithme récursif naïf au prix d'une complexité spatiale plus importante.

• La version descendante permet de ne s'intéresser qu'aux sous-problèmes "utiles" sans réflexion par rapport aux problèmes

• La version ascendante peut être modifiée pour gagner en complexité spatiale mais cela demande de bien réfléchir au problème.

• Le plus difficile reste de trouver une relation entre le problème et ses sous-problèmes.