Programmation Dynamique

Cours de CPGE PSI, 4 novembre 2026

Adeline Pierrot

Complexité des fonctions récursives

On veut calculer les termes de la suite définie par $u_n = \frac{1}{2}(u_{n-1} + \frac{3}{u_{n-1}})$ et $u_0 = 2$.

Quelle est la complexité de la fonction suivante?

```
def u(n):
    if n == 0:
        return 2
    else :
        return 1/2*(u(n-1) + 3/u(n-1))
```

Complexité des fonctions récursives

On veut calculer les termes de la suite définie par $u_n = \frac{1}{2}(u_{n-1} + \frac{3}{u_{n-1}})$ et $u_0 = 2$.

Quelle est la complexité de la fonction suivante?

```
def u(n):
    if n == 0:
        return 2
    else :
        return 1/2*(u(n-1) + 3/u(n-1))
```

Complexité exponentielle: $\mathcal{O}(2^n)$ à cause des deux appels récursifs sur $n-1$.

Quelle est la complexité de la fonction suivante?

```
def u(n):
    if n == 0:
        return 2
    else :
        x = u(n-1)
        return 1/2*(x+3/x)
```

Quelle est la complexité de la fonction suivante?

```
def u(n):
    if n == 0:
        return 2
    else :
        x = u(n-1)
        return 1/2*(x+3/x)
```

Complexité linéaire: $\mathcal{O}(n)$ car un seul appel récursif sur $n-1$.

Quelle est la complexité de la fonction suivante?

```
def u(n):
    if n == 0:
        return 2
    else :
        x = u(n-1)
        return 1/2*(x+3/x)
```

Complexité linéaire: $\mathcal{O}(n)$ car un seul appel récursif sur $n-1$.

Parfois résoudre le problème de complexité d'une fonction récursive est simple...

... parfois c'est plus compliqué.

On veut calculer le coefficient binomial $\binom{n}{p}$ à l'aide de la formule du triangle de Pascal.

Quelle est la complexité de la fonction suivante?

```
def binome(n,p):
    if p ==0 or p == n :
        return 1
    else:
        return binome(n-1,p)+ binome(n-1,p-1)
```

... parfois c'est plus compliqué.

On veut calculer le coefficient binomial $\binom{n}{p}$ à l'aide de la formule du triangle de Pascal.

Quelle est la complexité de la fonction suivante?

```
def binome(n,p):
    if p ==0 or p == n :
        return 1
    else:
        return binome(n-1,p)+ binome(n-1,p-1)
```

Complexité exponentielle à cause des deux appels récursifs.

... parfois c'est plus compliqué.

On veut calculer le coefficient binomial $\binom{n}{p}$ à l'aide de la formule du triangle de Pascal.

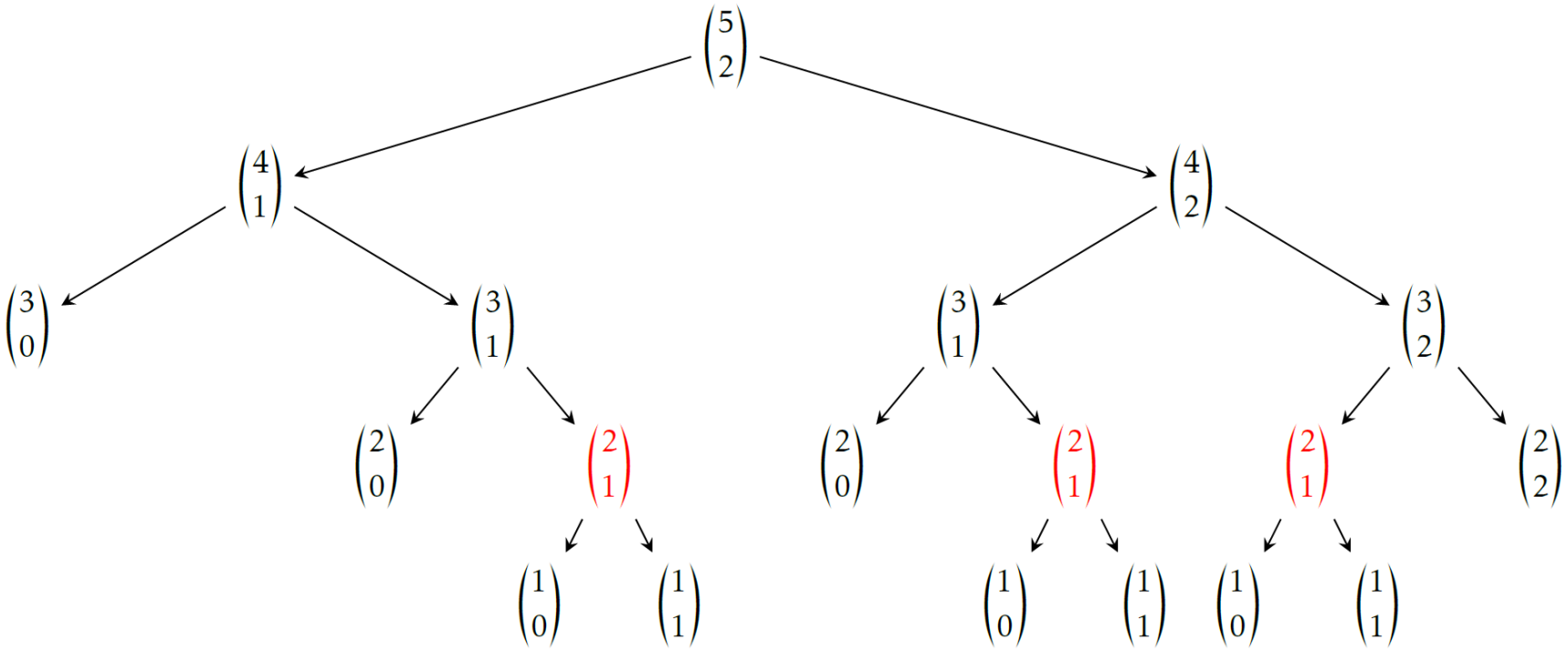
Quelle est la complexité de la fonction suivante?

```
def binome(n,p):
    if p ==0 or p == n :
        return 1
    else:
        return binome(n-1,p)+ binome(n-1,p-1)
```

Complexité exponentielle à cause des deux appels récursifs.

Cette fois les deux appels récursifs ne sont pas exactement les mêmes, on ne va pas s'en sortir avec une simple variable.

Chevauchement des sous-problèmes



Pour calculer $\binom{5}{2}$, `binome(5,2)` appelle trois fois `binome(2,1)`.

Mémoïsation

Pour éviter de re-calculer plusieurs fois les mêmes termes, on va mémoriser les termes déjà calculés $\rightarrow$ Principe de *mémoïsation*.

Mémoïsation

Pour éviter de re-calculer plusieurs fois les mêmes termes, on va mémoriser les termes déjà calculés $\rightarrow$ Principe de *mémoïsation*.

On peut utiliser un tableau à deux dimensions (ex: liste de listes) et stocker le résultat de `binome(i,j)` dans la case d'indice `(i,j)`.

Mémoïsation

Pour éviter de re-calculer plusieurs fois les mêmes termes, on va mémoriser les termes déjà calculés $\rightarrow$ Principe de *mémoïsation*.

On peut utiliser un tableau à deux dimensions (ex: liste de listes) et stocker le résultat de `binome(i,j)` dans la case d'indice `(i,j)`.

Pour utiliser moins de place mémoire, on peut utiliser un dictionnaire, en associant à la clé `(i,j)` la valeur de `binome(i,j)`.

Mémoïsation

Pour éviter de re-calculer plusieurs fois les mêmes termes, on va mémoriser les termes déjà calculés → Principe de *mémoïsation*.

On peut utiliser un tableau à deux dimensions (ex: liste de listes) et stocker le résultat de `binome(i,j)` dans la case d'indice `(i,j)`.

Pour utiliser moins de place mémoire, on peut utiliser un dictionnaire, en associant à la clé `(i,j)` la valeur de `binome(i,j)`.

```
binom_dict = {}
def binome(n, p):
    if (n, p) not in binom_dict:
        if p == 0 or p == n:
            b = 1
        else:
            b = binome(n-1,p-1) + binome(n-1,p)
        binom_dict[(n, p)] = b
    return binom_dict[(n, p)]
```

Attention, le dictionnaire ne doit pas être une variable locale à la fonction récursive, car il doit être le même pour tous les appels récursifs.

Attention, le dictionnaire ne doit pas être une variable locale à la fonction récursive, car il doit être le même pour tous les appels récursifs.
On peut le définir comme une variable globale (voir page précédente), ou le mettre en paramètre des fonctions, ou mettre la fonction récursive à l'intérieur d'une fonction dans laquelle est définie le dictionnaire, ce qui permet d'éviter les variables globales sans rajouter de paramètre:

Attention, le dictionnaire ne doit pas être une variable locale à la fonction récursive, car il doit être le même pour tous les appels récursifs.
On peut le définir comme une variable globale (voir page précédente), ou le mettre en paramètre des fonctions, ou mettre la fonction récursive à l'intérieur d'une fonction dans laquelle est définie le dictionnaire, ce qui permet d'éviter les variables globales sans rajouter de paramètre:

```
def binome(n, p):
    binom_dict = {}
    def bin_rec(n,p):
        if (n, p) not in binom_dict:
            if p == 0 or p == n:
                b = 1
            else:
                b = bin_rec(n-1,p-1) + bin_rec(n-1,p)
            binom_dict[(n, p)] = b
        return binom_dict[(n, p)]
    return bin_rec(n,p)
```

Calcul de bas en haut

Une autre solution au problème de complexité des fonctions récursives est de trouver une fonction itérative équivalente à la fonction récursive.

Calcul de bas en haut

Une autre solution au problème de complexité des fonctions récursives est de trouver une fonction itérative équivalente à la fonction récursive.

La récursivité est une approche "haut en bas" (top down): on résout le problème globlal (ex: calcul de f(n)) en appellant f sur une valeur un peu plus petite (typiquement n-1).

Calcul de bas en haut

Une autre solution au problème de complexité des fonctions récursives est de trouver une fonction itérative équivalente à la fonction récursive.

La récursivité est une approche "haut en bas" (top down): on résout le problème globlal (ex: calcul de f(n)) en appellant f sur une valeur un peu plus petite (typiquement n-1).

On peut au contraire essayer de mettre en oeuvre une approche "de bas en haut" (bottom up): commencer le calcul pour les petites valeurs, et construire itérativement le résultat pour des valeurs plus grandes à partir du résultat sur les valeurs les plus petites.

Calcul de bas en haut

Une autre solution au problème de complexité des fonctions récursives est de trouver une fonction itérative équivalente à la fonction récursive.

La récursivité est une approche "haut en bas" (top down): on résout le problème globlal (ex: calcul de f(n)) en appellant f sur une valeur un peu plus petite (typiquement n-1).

On peut au contraire essayer de mettre en oeuvre une approche "de bas en haut" (bottom up): commencer le calcul pour les petites valeurs, et construire itérativement le résultat pour des valeurs plus grandes à partir du résultat sur les valeurs les plus petites.

Pour réaliser ce type de solution on utilise souvent un tableau qui sera progressivement rempli par les valeurs sucessivement calculées.

```
def binome(n, p):
    t = [[0]*(p+1) for _ in range(n+1)]
    for i in range(0, n+1):
        t[i][0] = 1
    for i in range(1, min(n,p)+1):
        t[i][i] = 1
    for i in range(2, n+1):
        for j in range(1, min(p,i)+1):
            t[i][j] = t[i-1][j-1] + t[i-1][j]
    return t[n][p]
```

```
def binome(n, p):
    t = [[0]*(p+1) for _ in range(n+1)]
    for i in range(0, n+1):
        t[i][0] = 1
    for i in range(1, min(n,p)+1):
        t[i][i] = 1
    for i in range(2, n+1):
        for j in range(1, min(p,i)+1):
            t[i][j] = t[i-1][j-1] + t[i-1][j]
    return t[n][p]
```

La programmation dynamique est la mise en oeuvre de cette approche "bas en haut" pour résoudre de façon exacte des problèmes d'optimisation tout en étant efficace.

Conclusion

Une fonction récursive peut poser des problèmes de complexité, qui la rend inutilisable en pratique (temps de calcul trop long).

Bien sûr certaines fonctions récursives ont directement un temps d'exécution linéaire, et pour d'autres il suffit de faire attention en l'écrivant (stocker la valeur dans une variable plutôt que de faire deux fois le même appel).

Conclusion

Une fonction récursive peut poser des problèmes de complexité, qui la rend inutilisable en pratique (temps de calcul trop long).

Bien sûr certaines fonctions récursives ont directement un temps d'exécution linéaire, et pour d'autres il suffit de faire attention en l'écrivant (stocker la valeur dans une variable plutôt que de faire deux fois le même appel).

Dans des cas plus complexes, on peut faire des fonctions récursives avec mémoïsation : éviter de recalculer plusieurs fois les mêmes termes en utilisant un dictionnaire où on stocke les valeurs déjà calculées.

Conclusion

Une fonction récursive peut poser des problèmes de complexité, qui la rend inutilisable en pratique (temps de calcul trop long).

Bien sûr certaines fonctions récursives ont directement un temps d'exécution linéaire, et pour d'autres il suffit de faire attention en l'écrivant (stocker la valeur dans une variable plutôt que de faire deux fois le même appel).

Dans des cas plus complexes, on peut faire des fonctions récursives avec mémoïsation : éviter de recalculer plusieurs fois les mêmes termes en utilisant un dictionnaire où on stocke les valeurs déjà calculées.

On peut aussi remplacer la fonction récursive par une fonction itérative équivalente en mettant en oeuvre une approche "de bas en haut" (bottom up), souvent à l'aide d'un tableau rempli itérativement.