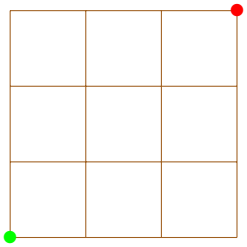
14.1 Introduction

14.1.1 Un exemple pour comprendre !

Approche

On désire relier le point rouge à partir du point vert. On ne peut se déplacer que sur les traits horizontaux vers la droite et le long des traits verticaux vers le haut.



☛ Combien existe-t-il de chemins différents ?

Exercice 14.1©

Créer une fonction python `nb_chemins(n,p)` avec `n` le nombre de ligne et `p` le nombre de colonne, qui renvoie le nombre de chemins possibles pour aller du point (0,0) au point (n,p).
On pourra considérer une matrice de taille (n+1)*(p+1) (intersections)
On pourra en déduire le nombre de chemins différents sur une grille de 10×10 ?
***

☛ De façon générale, la programmation dynamique est une technique qui évite de ne pas calculer la même chose plusieurs fois. Dans l'exemple ci-dessus, on utilise un tableau à deux dimensions pour stocker les nombres de chemins déjà calculés...

14.1.2 Un autre exemple avec la suite de fibonnaci

Il s'agit de la suite de Fibonacci. Rappelons la formule de récurrence définissant cette suite :

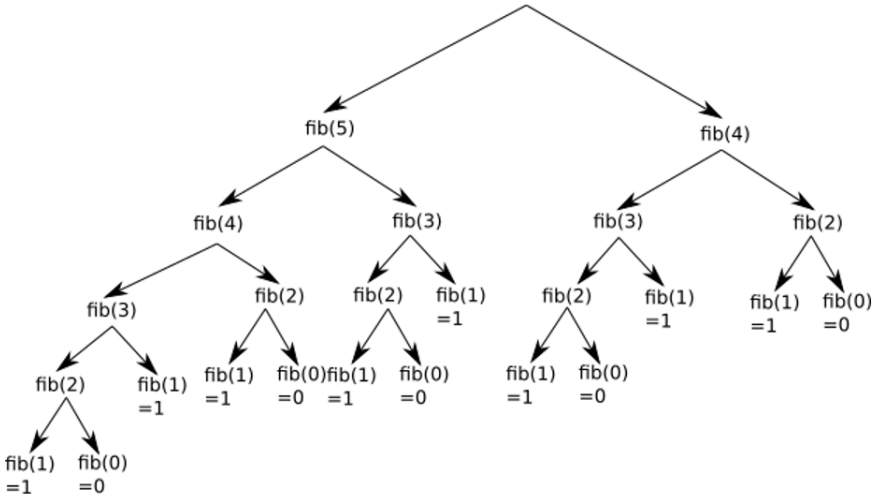
$$u_n = \begin{cases} 0 \; si \; n = 0 \\ 1 \; si \; n = 1 \\ u_{n-1} + u_{n-2} \; si \; n \geq 2 \end{cases}$$

Ses premiers termes sont donc 0, 1, 1, 2, 3, 5, 8, 13, 21, etc.

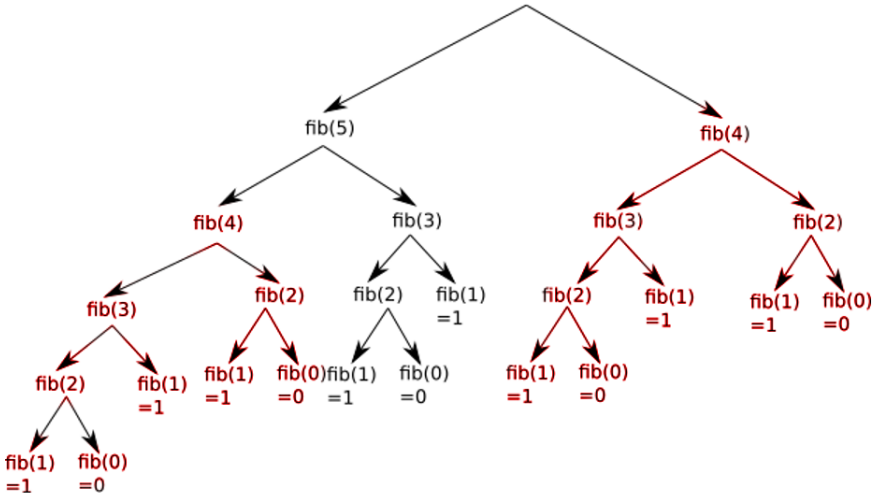
Exercice 14.2

construire l'algorithme récursif d'une fonction calculant le terme d'indice n de la suite. Le traduire en programme python.
***

Pour n=6, il est possible d'illustrer le fonctionnement de ce programme avec le schéma ci-dessous :



On peut alors que constater que plusieurs appels sont réalisés avec une même valeur du paramètre (typique dans les algorithmes récursifs), si on additionne toutes les feuilles de cette structure arborescente (fib(1)=1 et fib(0)=0), on retrouve bien 8.



En observant attentivement le schéma ci-dessus, on remarque que de nombreux calculs sont inutiles, car effectué 2 fois : par exemple on retrouve le calcul de fib(4) à 2 endroits (en haut à droite et un peu plus bas à gauche). On réalise ainsi un grand nombre d'opérations inutiles car redondantes. Celles-ci sont très coûteuses et expliquent la complexité exponentielle de cet algorithme.

On pourrait donc grandement simplifier le calcul en calculant une fois pour toutes fib(4), en "mémorisant" le résultat et en le réutilisant quand nécessaire.

Cela n'est bien sûr possible que si les sous-problèmes ne sont pas indépendants. Cela signifie donc que ces sous-problèmes ont des sous-sous-problèmes communs.
Dans le cas qui nous intéresse, on peut légitimement s'interroger sur le bénéfice de cette opération de "mémorisation", mais pour des valeurs de n beaucoup plus élevées, la question ne se pose même pas, le gain en termes de performance (temps de calcul) est évident. Pour des

valeurs n très élevées, dans le cas du programme récursif "classique" (n'utilisant pas la "mémorisation"), on peut même se retrouver avec un programme qui "plante" à cause du trop grand nombre d'appels récursifs. La méthode que nous venons d'utiliser se nomme "programmation dynamique".

La programmation dynamique, comme la méthode diviser pour régner, résout des problèmes en combinant des solutions de sous-problèmes. Cette méthode a été introduite au début des années 1950 par Richard Bellman.