Chapitre III
Récursivité

To understand what recursion is,
you must first understand recursion.

1. Algorithmes récursifs

1.1 Un premier exemple

Considérons le problème suivant : rédiger une fonction `star1(n)` qui prend pour argument un entier naturel $n \in \mathbb{N}$ et affiche dans la console n étoiles sur la première ligne, $n-1$ sur la seconde ligne, …, et enfin une étoile sur la n^{e} ligne.

```
In [1]: star1(5)
*****
****
***
**
*
```

```
In [2]: star2(5)
*
**
***
****
*****
```

```
In [3]: star3(4)
****
***
**
*
**
***
****
```

FIGURE 1 – Un exemple d'utilisation des fonctions `star1`, `star2` et `star3`.

Il est bien entendu très simple de rédiger une version itérative de cette fonction, mais on peut aussi observer qu'il ne s'agit finalement, lorsque n est strictement positif, que de tracer n étoiles sur la première ligne puis d'appliquer `star1(n-1)`.Traduit en Python, cela donne :

```
def star1(n):
    if n > 0:
        print(n * '*')
        star1(n - 1)
```

Une telle fonction est dite *récursive* car elle est utilisée *au sein même de sa définition*. Pour qu'une telle définition soit valide, il est nécessaire que deux conditions soient vérifiées :

- il doit y avoir une *condition d'arrêt*, c'est-à-dire au moins une valeur de n pour laquelle aucun appel récursif n'est effectué ;
- il ne doit pas y avoir de suite *infinie* d'appels récursifs.

Dans l'exemple de la fonction `star1`, la condition d'arrêt correspond aux entiers négatifs ou nuls (il ne se passe rien), et pour chaque entier $n \geqslant 1$, n appels récursifs sont réalisés (un nombre fini, donc).

EXERCICE 1

a. Rédiger une fonction récursive `star2(n)` qui prend pour argument un entier n et qui affiche une étoile sur la première ligne, deux sur la seconde, … et enfin n sur la n^{e} ligne (voir un exemple figure 1).

b. Rédiger une fonction récursive `star3(n)` sur le modèle de la fonction illustrée figure 1.

1.2 L'algorithme d'exponentiation rapide

Mais au fond, à quoi bon écrire un algorithme récursif ? Somme toute, les trois fonctions `star1`, `star2` et `star3` sont très simples à écrire de manière itérative. Il existe cependant des algorithmes *qui s'expriment naturellement de manière récursive*, et ce sont avant tout ces algorithmes qui motivent l'utilisation de la programmation récursive.

C'est le cas par exemple de l'*algorithme d'exponentiation rapide*. Cet algorithme calcule x^n en exploitant les formules :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x & \text{si } n = 1 \\ (x^2)^p & \text{si } n = 2p \text{ est pair} \\ x(x^2)^p & \text{si } n = 2p+1 \text{ est impair} \end{cases}$$

Compte tenu de sa formulation, il est naturel d'utiliser la programmation récursive pour le traduire :

```
def power(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    y = power(x * x, n // 2)
    if n % 2 == 0:
        return y
    else:
        return x * y
```

Il y a ici deux conditions d'arrêts : $n = 0$ et $n = 1$. Montrons maintenant par récurrence sur n que pour tout $n \in \mathbb{N}$, seul un nombre fini d'appels récursif est réalisé.

- Si $n = 0$ ou $n = 1$ il n'y a pas d'appel récursif.
- Si $n \geqslant 2$, supposons le résultat acquis jusqu'au rang $n-1$. Sachant que $p = \lfloor n/2 \rfloor < n$, on peut applique l'hypothèse de récurrence : le calcul de $(x^2)^p$ avec $p = \lfloor n/2 \rfloor$ n'utilise qu'un nombre fini d'appels récursifs ; il en est donc de même du calcul de x^n. La récurrence se propage.

Remarque. Si $C(n)$ désigne le nombre d'appels récursifs nécessaires pour calculer x^n, nous venons de montrer que :

$$C(0) = C(1) = 0 \qquad \text{et} \qquad C(n) = 1 + C(\lfloor n/2 \rfloor) \quad \text{pour } n \geqslant 2$$

En utilisant la décomposition de n en base 2 : $n = (b_p \cdots b_1 b_0)_2$ on trouve que $C((b_p \cdots b_1 b_0)_2) = 1 + C((b_p \cdots b_1)_2)$ puis aisément : $C((b_p \cdots b_1 b_0)_2) = p + C(1) = p$ donc $C(n) = \lfloor \log_2 n \rfloor$. Sachant qu'à part les appels récursifs, les autres opérations (des produits et des divisions) sont de complexité constante, on en déduit que cet algorithme a une complexité temporelle en $O(\log n)$, bien meilleure donc qu'une complexité linéaire que produirait l'algorithme itératif naïf.

1.3 Les tours de Hanoï

Il n'existe pas de réponse définitive à la question de savoir si un algorithme récursif est préférable à un algorithme itératif. Il a été prouvé que ces deux paradigmes de programmation sont équivalents ; autrement dit, tout algorithme itératif possède une version récursive, et réciproquement. Un algorithme récursif est aussi performant qu'un algorithme itératif pour peu que le programmeur ait évité les quelques écueils qui seront étudiés plus loin dans ce cours et que le compilateur ou l'interprète de commande gère convenablement la récursivité. Le choix du langage peut aussi avoir son importance : un langage fonctionnel tel OCaml est conçu pour exploiter la récursivité et le programmeur est naturellement amené à choisir la version récursive de l'algorithme qu'il souhaite écrire. À l'inverse, Python, même s'il l'autorise, ne favorise pas l'écriture récursive (limitation basse par défaut du nombre d'appels récursifs, pas ou peu d'optimisation des algorithmes récursifs).

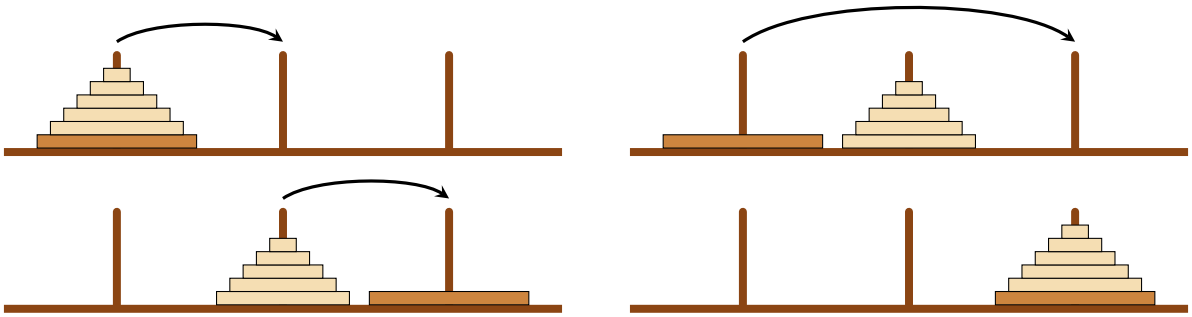
Figure 2 – Le puzzle dans sa configuration initiale.

Parmi les problèmes dont la résolution récursive est beaucoup plus simple que la résolution itérative, l'un des plus emblématiques est sans conteste le problème des tours de Hanoï inventé par le mathématicien français

Édouard Lucas. Ce jeu mathématique est constitué de trois tiges sur lesquelles sont enfilés n disques de diamètres différents. Au début du jeu, ces disques sont tous positionnés sur la première tige (du plus grand au plus petit) et l'objectif est de déplacer tous ces disques sur la troisième tige, en respectant les règles suivantes :

- un seul disque peut être déplacé à la fois ;
- on ne peut jamais poser un disque sur un disque de diamètre inférieur.

Maintenant, raisonnons par récurrence : pour pouvoir déplacer le dernier disque, il est nécessaire de déplacer les $n-1$ disques qui le couvrent sur la tige centrale. Une fois ces déplacements effectués, nous pouvons déplacer le dernier disque sur la troisième tige. Il reste alors à déplacer les $n-1$ autres disques vers la troisième tige.



Tout est dit : pour pouvoir déplacer n disques de la tige 1 vers la tige 3 il suffit de savoir déplacer $n-1$ disques de la tige 1 vers la tige 2 puis de la tige 2 vers la tige 3. Autrement dit, il suffit de généraliser le problème de manière à décrire le déplacement de n disques de la tige i à la tige k en utilisant la tige j comme pivot. Ceci conduit à la définition suivante :

```
def hanoi(n, i=1, j=2, k=3):
    if n == 0:
        return None
    hanoi(n-1, i, k, j)
    print("Déplacer le disque {} de la tige {} vers la tige {}.".format(n, i, k))
    hanoi(n-1, j, i, k)
```

On trouvera figure 3 un exemple de résolution pour $n = 4$.

```
In [1]: hanoi(4)
Déplacer le disque 1 de la tige 1 vers la tige 2.
Déplacer le disque 2 de la tige 1 vers la tige 3.
Déplacer le disque 1 de la tige 2 vers la tige 3.
Déplacer le disque 3 de la tige 1 vers la tige 2.
Déplacer le disque 1 de la tige 3 vers la tige 1.
Déplacer le disque 2 de la tige 3 vers la tige 2.
Déplacer le disque 1 de la tige 1 vers la tige 2.
Déplacer le disque 4 de la tige 1 vers la tige 3.
Déplacer le disque 1 de la tige 2 vers la tige 3.
Déplacer le disque 2 de la tige 2 vers la tige 1.
Déplacer le disque 1 de la tige 3 vers la tige 1.
Déplacer le disque 3 de la tige 2 vers la tige 3.
Déplacer le disque 1 de la tige 1 vers la tige 2.
Déplacer le disque 2 de la tige 1 vers la tige 3.
Déplacer le disque 1 de la tige 2 vers la tige 3.
```

Figure 3 – Une solution du problème des tours de Hanoï avec quatre disques.

Bien qu'il soit tentant de se demander suivant quelle logique les disques de tailles inférieures se déplacent (autrement dit, que se passe-t'il lorsqu'on déroule les différents appels récursifs), on notera bien que ce n'est pas nécessaire. Notre unique tâche est de réduire le problème à un ou plusieurs sous-problèmes identiques et à veiller à respecter les deux conditions pour qu'un algorithme récursif soit valide : existence d'une condition

d'arrêt (ici le cas $n = 0$) et nombre fini d'appels récursifs (il est ici facile de constater que le nombre d'appels récursifs est égal à $2^{n+1} - 2$).

Exercice 2

Résoudre le problème des tours de Hanoï en s'imposant une contrainte supplémentaire : tout mouvement entre les tiges 1 et 3 est interdit (tous les mouvements doivent donc aboutir ou débuter par la tige 2).
Déterminer le nombre de mouvements nécessaires dans ce cas.

1.4 Le tri fusion

Le tri fusion (*merge sort*) est un des premiers algorithmes inventés pour trier un tableau car (selon Donald Knuth) il aurait été proposé par John von Neuman dès 1945 ; il constitue un parfait exemple d'algorithme naturellement récursif.
Son fonctionnement est le suivant :

- diviser le tableau à trier en deux parties sensiblement égales ;
- trier récursivement chacune de ces deux parties ;
- fusionner les deux parties triées dans un seul tableau trié.

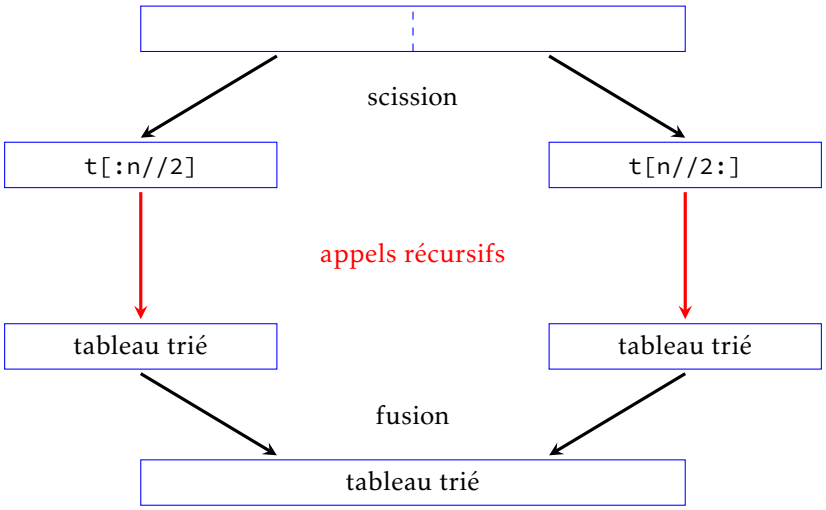


Figure 4 – Une illustration du tri fusion.

Dans la mise en œuvre figure 5, la fonction `merge` prend pour arguments deux tableaux supposés triés par ordre croissant, et renvoie un nouveau tableau, résultat de la fusion des deux tableaux passés en argument. Cette fonction est séparée du corps principal de l'algorithme pour accroitre la lisibilité de la structure récursive.

Quelle est la complexité temporelle de cette fonction ? La fonction `merge` est à l'évidence de coût linéaire vis-à-vis de la somme des longueurs des deux tableaux passés en paramètre. Si $C(n)$ désigne la complexité temporelle du tri d'un tableau de longueur n, on dispose donc de la relation :

$$C(n) = C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + O(n).$$

Ce type de relation de récurrence est typique des méthodes dites « diviser pour régner » ; dans le cas présent on prouve que cette relation implique que $C(n) = O(n \log n)$. Nous apprendrons dans le chapitre suivant qu'il n'est pas possible de faire mieux dans le cadre des algorithmes de tri par comparaison.

1.5 Récursivité et pile d'exécution d'un programme

Un programme n'étant qu'un flux d'instructions exécutées séquentiellement, son exécution peut être représentée par le parcours d'un chemin ayant une origine et une extrémité. Lors de l'appel d'une fonction, ce flux est

```python
def merge(a, b):
    p, q = len(a), len(b)
    c = [None] * (p + q)
    i = j = 0
    for k in range(p+q):
        if j >= q:
            c[k] = a[i]
            i += 1
        elif i >= p:
            c[k] = b[j]
            j += 1
        elif a[i] < b[j]:
            c[k] = a[i]
            i += 1
        else:
            c[k] = b[j]
            j += 1
    return c
```

```python
def mergesort(t):
    n = len(t)
    if n < 2:
        return t
    a = mergesort(t[:n//2])
    b = mergesort(t[n//2:])
    return merge(a, b)
```

FIGURE 5 – Implémentation du tri fusion.

interrompu le temps de l'exécution de cette fonction, avant de reprendre à l'endroit où le programme s'est arrêté (voir figure 6).
Au moment où débute cette bifurcation, il est nécessaire que le processeur sauvegarde un certain nombre d'informations : adresse de retour, état des paramètres et des variables, etc. Toutes ces données forment ce qu'on appelle le *contexte* du programme, et elles sont stockées dans une pile qu'on appelle la *pile d'exécution* [1]. À la fin de l'exécution de la fonction, le contexte est sorti de la pile pour être rétabli et permettre la poursuite de l'exécution du programme.
L'utilisation d"une pile se comprend : lorsque plusieurs appels à des fonctions différentes sont emboités, c'est le contexte de la dernière fonction à avoir été appelée qui doit être rétabli en premier ; c'est bien là le principe des piles LIFO (*Last In, First Out*).

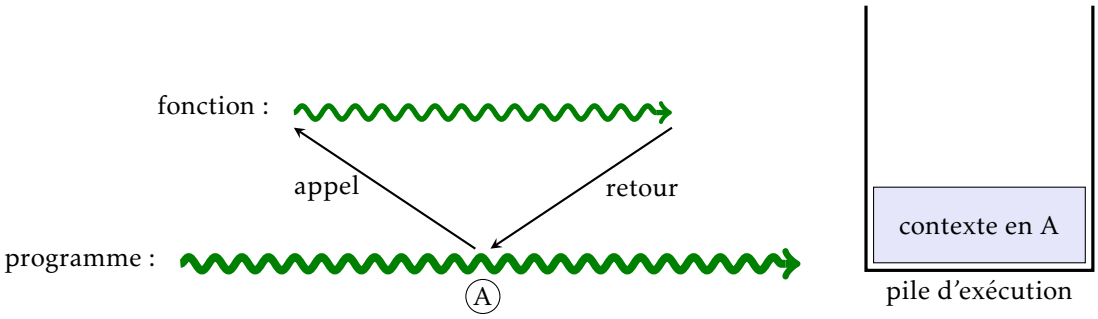


FIGURE 6 – L'exécution d'une fonction au sein d'un programme.

Lors de l'exécution d'une fonction récursive, chaque appel récursif conduit au moment où il se produit à un empilement du contexte dans la pile d'exécution. Lorsqu'au bout de n appels se produit la condition d'arrêt, les différents contextes sont progressivement dépilés pour poursuivre l'exécution de la fonction. La figure 7 illustre les trois premiers appels récursifs de la fonction récursive `power` avec pour paramètres $x = 3$ et $n = 41$ (pour des raisons de lisibilité, seules les valeurs de ces deux paramètres sont présentées dans le contexte).
Il est donc important de prendre conscience qu'une fonction récursive va s'accompagner d'un coût spatial qui va croître avec le nombre d'appels récursifs (en général linéairement, mais ce n'est pas une règle générale, tout dépend du contenu du contexte) ; ce coût ne doit pas être oublié lorsqu'on fait le bilan de la complexité spatiale d'une fonction récursive.

1. Suivant les langages et leurs implémentations, il peut y avoir une pile d'exécution par fonction ou une seule pile globale pour tout le programme.

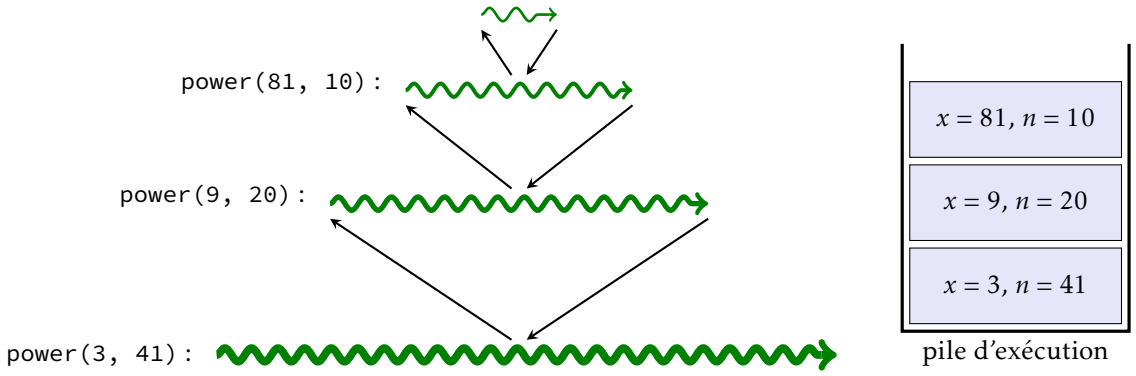


FIGURE 7 – Calcul récursif de 3^{41} par exponentiation rapide.

Remarque. Que ce passe-t-il lorsqu'un algorithme ne se termine pas, soit parce qu'on a oublié la condition d'arrêt, soit parce qu'une entrée génère un nombre infini d'appels récursifs ? La pile d'exécution croît indéfiniment, jusqu'à dépasser la taille limite que peut fournir la mémoire. Ceci pouvant conduire à un comportement erratique de l'ordinateur, Python possède un garde-fou pour empêcher cela : une variable système qui limite le nombre d'appels récursifs (sa valeur dépend du système). C'est pourquoi un algorithme correct peut parfois conduire à une erreur si le nombre maximal d'appels récursifs est dépassé.
Pour illustrer ceci, j'ai délibérément fortement baissé la valeur de cette limite dans l'exemple qui suit :

```
In [1]: star1(10)
**********
*********
********
*******
******
*****
****
RecursionError: maximum recursion depth exceeded while calling a Python object
```

1.6 Trace d'une fonction

Dans les langages de programmation de haut niveau, les spécificités de la pile d'exécution sont cachées au programmeur. Ce dernier a uniquement accès aux appels de fonctions et aux paramètres associés, et non au contenu de la pile elle-même. On peut cependant obtenir une représentation de la pile d'exécution appelée la *trace d'appels* en faisant apparaître les paramètres d'entrées et les valeurs de retour de chaque appel, ce qui peut faciliter entre autre le débogage d'une fonction.

On trouvera figure 8 la trace des appels récursifs de la fonction `power` pour le couple $(x = 3, n = 41)$ et de la fonction `mergesort` pour le tableau $[6, 2, 4, 3, 5, 1]$. Une flèche orientée vers la gauche dénote un appel de la fonction (donc un empilement dans la pile d'exécution) ; une flèche orientée vers la droite un dépilement de celle-ci.

2. Les écueils de la programmation récursive

Nous allons maintenant aborder les principaux pièges qui guettent le programmeur qui souhaite écrire une fonction récursive, non pas pour vous dissuader d'en écrire, mais pour vous permettre de les éviter.

```
power <- (3, 41)
 power <- (9, 20)
  power <- (81, 10)
   power <- (6561, 5)
    power <- (43046721, 2)
     power <- (1853020188851841, 1)
     power -> 1853020188851841
    power -> 1853020188851841
   power -> 12157665459056928801
  power -> 12157665459056928801
 power -> 12157665459056928801
power -> 36472996377170786403
```

```
mergesort <- [6, 2, 4, 3, 5, 1]
 mergesort <- [6, 2, 4]
  mergesort <- [6]
  mergesort -> [6]
  mergesort <- [2, 4]
   mergesort <- [2]
   mergesort -> [2]
   mergesort <- [4]
   mergesort -> [4]
  mergesort -> [2, 4]
 mergesort -> [2, 4, 6]
 mergesort <- [3, 5, 1]
  mergesort <- [3]
  mergesort -> [3]
  mergesort <- [5, 1]
   mergesort <- [5]
   mergesort -> [5]
   mergesort <- [1]
   mergesort -> [1]
  mergesort -> [1, 5]
 mergesort -> [1, 3, 5]
mergesort -> [1, 2, 3, 4, 5, 6]
```

FIGURE 8 – Un exemple de trace des fonctions `power` et `mergesort`.

2.1 Attention aux coûts cachés

Les principales difficultés sont liées à de mauvaises évaluations des coûts tant temporels que spatiaux. Partons d'un exemple connu, l'algorithme de recherche dichotomique. Étant donné un tableau t de taille n contenant une liste *triée* par ordre croissant d'éléments et un élément x, on cherche à savoir si x se trouve dans t. La démarche est connue, et vous l'avez étudié en première année : il s'agit de comparer x à t_k avec $k = \lfloor n/2 \rfloor$ puis :

- si $x = t_k$, la recherche est terminée ;
- si $x < t_k$ la recherche se poursuit dans $t[0 \dots k-1]$;
- si $x > t_k$ la recherche se poursuit dans $t[k+1 \dots n-1]$.

Cette description se prête à merveille à une programmation de nature récursive, et il est tentant d'écrire le code que l'on trouve figure 9.

```
def dicho(x, t):
    if len(t) == 0:
        return False
    k = len(t) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho(x, t[:k])
    else:
        return dicho(x, t[k+1:])
```

FIGURE 9 – Une première version de la recherche dichotomique dans un tableau.

Bien que particulièrement limpide, ce code se révèle mauvais car le calcul de `t[:k]` et de `t[k+1:]` est de coût linéaire, tant temporel que spatial (car on procède à une recopie de la moitié de tableau dans un autre espace mémoire). La relation de récurrence qui régit la complexité est donc de la forme : $C(n) = C(n/2) + O(n)$, ce qui conduit à $C(n) = O(n)$. Cet algorithme est de complexité linéaire, donc du même ordre que l'algorithme de recherche dans un tableau non trié, et bien loin du coût logarithmique de l'algorithme itératif étudié en première année.

Il faut donc écrire une version récursive de l'algorithme qui ne procède à aucune recopie de tableau, et pour ce faire on doit généraliser le problème en écrivant une fonction qui recherche x dans la partie du tableau $t[i \dots j-1]$ en comparant x à t_k avec $k = \lfloor (i+j)/2 \rfloor$:

- si $x = t_k$, la recherche est terminée ;
- si $x < t_k$ la recherche se poursuit dans $t[i \dots k-1]$;
- si $x > t_k$ la recherche se poursuit dans $t[k+1 \dots j-1]$.

Le code de la version récursive correcte se trouve figure 10.

```
def dicho_aux(x, t, i, j):
    if j <= i:
        return False
    k = (i + j) // 2
    if x == t[k]:
        return True
    elif x < t[k]:
        return dicho_aux(x, t, i, k)
    else:
        return dicho_aux(x, t, k+1, j)

def dicho(x, t):
    return dicho_aux(x, t, 0, len(t))
```

Figure 10 – Une version récursive correcte de la recherche dichotomique.

Cette fois toutes les opérations qui précèdent les appels récursifs sont de coût constant donc la complexité vérifie une relation du type $C(n) = C(n/2) + O(1)$ qui donne $C(n) = O(\log n)$ (coût tant temporel que spatial).

Exercice 3

Un *palindrome* est un mot ou une phrase qui peut se lire indifféremment de la gauche vers la droite, tels *kayak* ou *ressasser*, ou encore, si on ne tiens pas compte des espaces et de la ponctuation, *Tu l'as trop écrasé, César, ce Port-Salut !*.
Rédiger une fonction récursive `palindrome(m)` qui prend en argument un mot m et renvoie `True` si ce mot est un palindrome, `False` sinon.

2.2 Appels récursifs multiples

Une difficulté plus sérieuse se présente lors d'appels récursifs multiples qui sont en interaction. L'exemple emblématique de ce problème concerne le calcul du n^e terme f_n de la suite de Fibonacci. La définition récursive naturelle est la suivante :

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

Malheureusement, les performances de cette fonction se dégradent extrêmement rapidement (voir figure 11) et au lieu d'un coût linéaire attendu on obtient un coût temporel d'apparence exponentiel.

Un début d'explication est donnée lorsqu'on trace cette fonction pour calculer f_6 (voir figure 12).
On constate que f_0 est calculé cinq fois, f_1 huit fois, f_2 cinq fois, f_3 trois fois, f_4 deux fois, f_5 une fois et f_6 une fois, ce qui fait (en tout) 25 appels à la fonction `fib` au lieu des 7 appels attendus.
Par exemple, f_4 est calculé une première fois pour calculer $f_5 = f_4 + f_3$ et une deuxième fois pour calculer $f_6 = f_5 + f_4$, f_3 va être calculé une fois pour chacun des deux calculs de f_4 et une fois pour calculer f_5 dont trois fois en tout, etc.

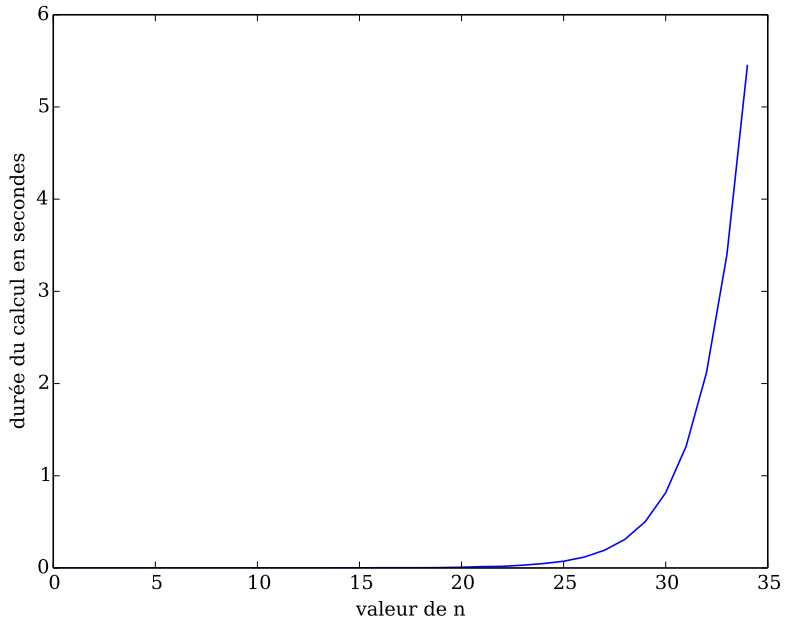


FIGURE 11 – Temps d'exécution en secondes pour calculer f_n par l'algorithme récursif.

Précisons les choses en calculant le nombre a_n d'appels à la fonction `fib` pour calculer f_n. On dispose des relations :

$$a_0 = a_1 = 1 \qquad \text{et} \qquad \forall n \geqslant 2,\ a_n = a_{n-1} + a_{n-2} + 1.$$

Cette suite se résout en $a_n = 2f_{n+1} - 1$. Sachant que $f_n \sim \frac{1}{\sqrt{5}}\phi^n$ (où ϕ est le nombre d'or) on obtient $a_n = O(\phi^n)$; le coût de cette fonction, tant temporel que spatial, est exponentiel !

Une solution : la mémoïsation

Il existe plusieurs solutions pour obtenir un coût plus raisonnable. Celle que nous allons adopter consiste à mémoriser le résultat des calculs une fois qu'ils auront été calculés, de manière à ne pas refaire deux fois le même calcul. La structure de données qui s'impose ici est la structure de *dictionnaire*, qui est constituée de paires associant une clef à une valeur. Dans le cas qui nous intéresse, la clef est l'entier n et la valeur, l'entier f_n.

Nous ne détaillerons pas l'implémentation des dictionnaires qui est complexe, et nous admettrons que le coût de l'ajout comme de la lecture dans un dictionnaire est en moyenne constant.

Nous pouvons maintenant réécrire la fonction `fib`, en la faisant précéder de la création d'un dictionnaire. Ensuite, le calcul de f_n se déroulera de la façon suivante : on commence par regarder si n est déjà présent dans le dictionnaire, et si ce n'est pas le cas (et uniquement dans ce cas) on calcule f_n à l'aide de la formule récursive. Une fois calculée, l'association (n, f_n) sera intégrée au dictionnaire :

```
d_fib = {0: 0, 1: 1}

def fib(n):
    if n not in d_fib:
        d_fib[n] = fib(n-1) + fib(n-2)
    return d_fib[n]
```

Cette solution permet de concilier la simplicité de la solution récursive avec l'efficacité temporelle, au prix d'un coût spatial linéaire (constitué du dictionnaire et de la pile d'exécution) :

```
fib <- 6
 fib <- 5
  fib <- 4
   fib <- 3
    fib <- 2
     fib <- 1
     fib -> 1
     fib <- 0
     fib -> 0
    fib -> 1
    fib <- 1
    fib -> 1
   fib -> 2
   fib <- 2
    fib <- 1
    fib -> 1
    fib <- 0
```

```
     fib -> 0
    fib -> 1
   fib -> 3
   fib <- 3
    fib <- 2
     fib <- 1
     fib -> 1
     fib <- 0
     fib -> 0
    fib -> 1
    fib <- 1
    fib -> 1
   fib -> 2
  fib -> 5
  fib <- 4
   fib <- 3
    fib <- 2
```

```
     fib <- 1
     fib -> 1
     fib <- 0
     fib -> 0
    fib -> 1
    fib <- 1
    fib -> 1
   fib -> 2
   fib <- 2
    fib <- 1
    fib -> 1
    fib <- 0
    fib -> 0
   fib -> 1
  fib -> 3
 fib -> 8
```

FIGURE 12 – La trace de la fonction récursive `fib`.

- `d = {c1: v1, c2: v2, c3: v3}` crée un dictionnaire `d` comportant pour l'instant trois paires d'association ;
- `d[c2]` renvoie la valeur (ici v_2) associée à la clef c_2 ou déclenche l'exception `KeyError` si l'association n'existe pas ;
- `d[c4] = v4` permet d'ajouter une nouvelle paire d'association (ou de remplacer la précédente association si c_4 est déjà dans le dictionnaire) ;
- `(c in d)` renvoie le booléen `True` si la clef c est dans d, et False sinon.

Seules restrictions : il n'est pas permis d'avoir plus d'une entrée par clef, et ces dernières ne doivent être des objets immuables.

FIGURE 13 – Un récapitulatif des principales fonctions des dictionnaires.

```
In [1]: fib(10)
Out[1]: 55

In [2]: d_fib
Out[2]: {0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55}
```

Autre exemple classique, le calcul des coefficients binomiaux à l'aide de la formule de PASCAL doit impérativement être mémoïsée sous peine d'obtenir un coût temporel exponentiel.

```
d_binom = {}

def binom(n, p):
    if (n, p) not in d_binom:
        if p == 0 or n == p:
            d_binom[(n, p)] = 1
        else:
            d_binom[(n, p)] = binom(n-1, p-1) + binom(n-1, p)
    return d_binom[(n, p)]
```

Sans mémoïsation, il faut près de 2 minutes pour calculer $\binom{30}{15}$ avec un processeur Intel Core 2 Duo à 2,13 GHz alors que le calcul demande moins d'une milli-seconde avec mémoïsation.

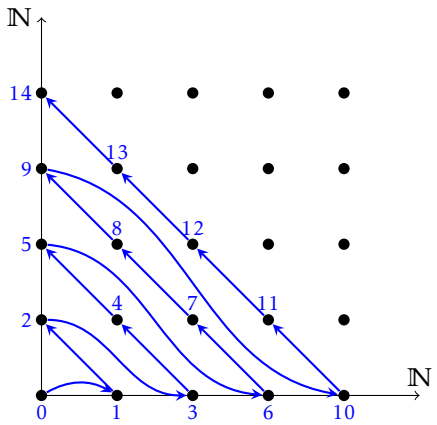
3. Exercices

Exercice 4

Écrire une fonction récursive qui calcule x^n en exploitant la relation : $x^n = x^{\lfloor n/2 \rfloor} \times x^{\lceil n/2 \rceil}$. Combien de multiplications effectue-t-elle ?

Exercice 5

On démontre que l'ensemble $\mathbb{N} \times \mathbb{N}$ est dénombrable en numérotant chaque couple $(x, y) \in \mathbb{N}^2$ suivant le procédé suggéré par la figure ci-dessous :



Rédiger une fonction *récursive* qui retourne le numéro du point de coordonnées (x, y).
Rédiger la fonction réciproque, là encore de façon récursive.

Exercice 6

On suppose donné un tableau $t[0 \cdots n-1]$ (contenant au moins trois éléments) qui possède la propriété suivante : $t_0 \geqslant t_1$ et $t_{n-2} \leqslant t_{n-1}$. Soit $k \in [\![1, n-2]\!]$; on dit que t_k est un *minimum local* lorsque $t_k \leqslant t_{k-1}$ et $t_k \leqslant t_{k+1}$.

a) Justifier l'existence d'un minimum local dans t.

b) Il est facile de déterminer un minimum local en coût linéaire : il suffit de procéder à un parcours du tableau. Mais pourriez-vous trouver un algorithme récursif qui en trouve un en coût logarithmique ?

Exercice 7

On suppose les ensembles représentés par des listes non triées d'éléments deux-à-deux distincts. Rédiger une fonction `subset` qui prend pour argument un ensemble et renvoie l'ensemble de ses sous-ensembles. Par exemple :

```
In [1]: subset([1,2,3])
Out[1]: [[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]
```

Exercice 8

Écrire une fonction `somme(M)` qui prend en paramètre une séquence imbriquée, de profondeur et de structure quelconques, dont tous les composants élémentaires sont des nombres, et calcule la somme de tous ces éléments. Par exemple, `somme([[[1, 2], [3, 4, 5]], 6, [7, 8], 9])` devra renvoyer `45`.
Indication. L'expression booléenne `isinstance(x, numbers.Real)` *permet de tester si x est un nombre.*

```
In [1]: isinstance(1, numbers.Real)
Out[1]: True

In [1]: isinstance([1], numbers.Real)
Out[1]: False
```

Exercice 9

On suppose disposer d'une fonction `circle((x, y), r)` qui trace à l'écran un cercle de centre (x,y) de rayon r. Définir deux fonctions récursives permettant de tracer les dessins présentés figure 14 (chaque cercle est de rayon moitié moindre qu'à la génération précédente).

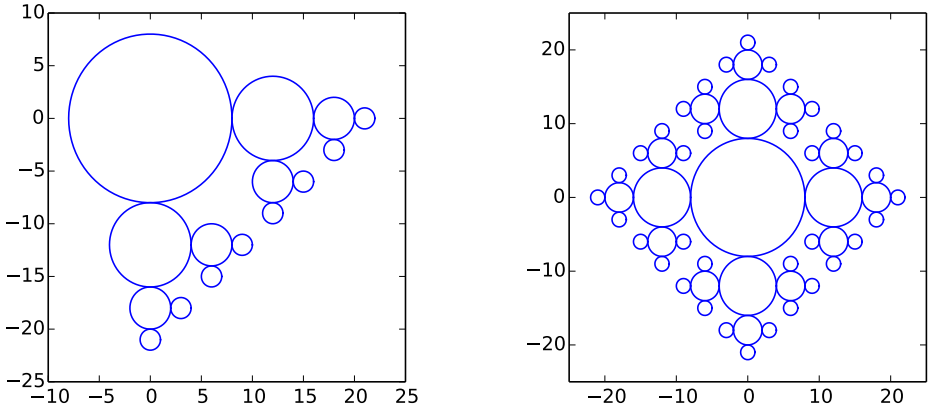


Figure 14 – Le résultat des fonctions `bubble1(4)` et de `bubble2(4)`.

Exercice 10

On suppose disposer d'une fonction `polygon((xa, ya), (xb, yb), (xc, yc))` qui trace le triangle plein dont les sommets ont pour coordonnées (x_a,y_a), (x_b,y_b), (x_c,y_c). Définir une fonction récursive permettant le tracé présenté figure 15 (tous les triangles sont équilatéraux).

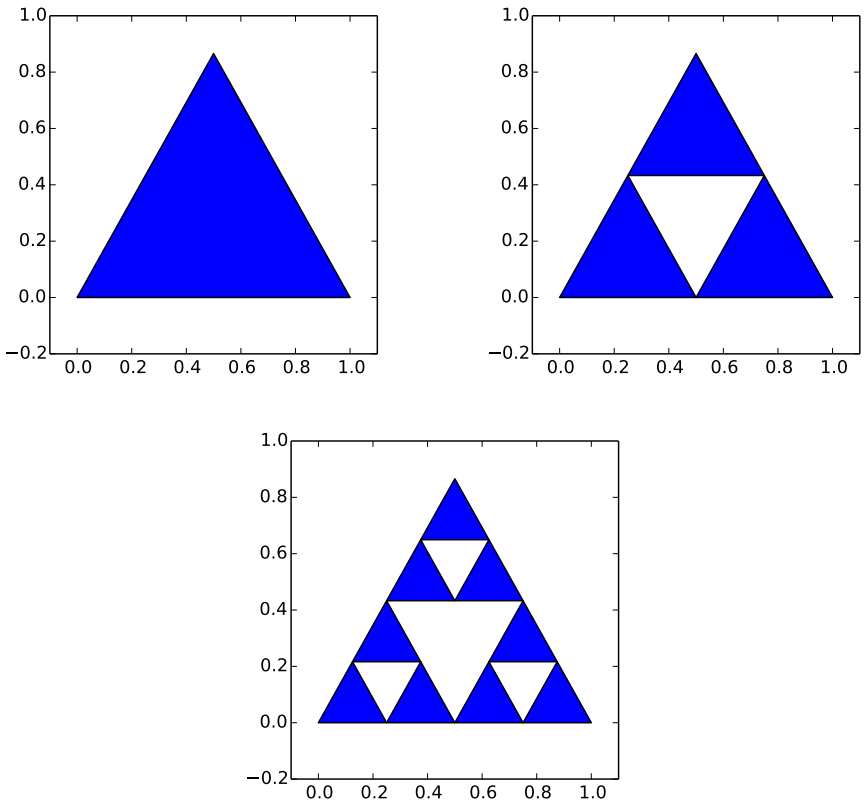


Figure 15 – Le résultat des fonctions `sierpinski(n)` pour $n = 1, 2, 3$.

Exercice 11

Le *problème des n reines* consiste à placer n reines sur un échiquier de taille $n \times n$ de sorte que deux reines quelconques ne puissent jamais s'attaquer (pour ceux d'entre vous qui ne sont pas familiers avec le jeu d'échecs, ceci signifie que deux reines quelconques ne partagent jamais la même ligne, la même colonne ou la même diagonale). De telles positions seront par la suite qualifiées de *valides* (illustration figure 16).
Il est évident que dans chaque colonne doit se trouver exactement une reine ; ainsi il est possible de représenter ce problème par un tableau de n cases $q = [q_0, \ldots, q_{n-1}]$ dans lequel q_j désigne la ligne où est placée la reine de la colonne j.

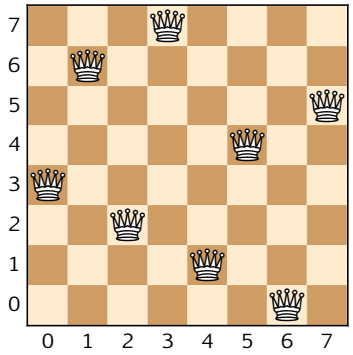


Figure 16 – Une solution, représentée par le tableau $[3, 6, 2, 7, 1, 4, 0, 5]$.

On appelera *solution partielle de rang j* un tableau q de longueur n dont les j premières cases sont remplies par des positions valides pour les reines, les $n - j$ autres cases restant à remplir.

Rédiger une fonction récursive `reine(q, j)` qui prend pour arguments un entier j et une solution partielle q et qui réalise les opérations suivantes :

- si $j = n$ cette fonction se contente d'afficher le tableau q (le problème est résolu) ;
- si $j < n$, cette fonction recherche parmi les n valeurs possibles pour q_j celles qui correspondent à des positions valides et pour chacune d'elles poursuit la recherche au rang $j + 1$.