Chapitre 2

Récursivité

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

1. Algorithmes récursifs

1.1 La multiplication du paysan russe

La *méthode du paysan russe* est un très vieil algorithme de multiplication de deux nombres entiers déjà décrit (sous une forme légèrement différente) sur un papyrus égyptien rédigé vers 1650 av. J.-C. Il s'agissait de la principale méthode de calcul en Europe avant l'introduction des chiffres arabes, et les premiers ordinateurs l'ont utilisé avant que la multiplication ne soit directement intégrée dans le processeur sous forme de circuit électronique.
Sous une forme moderne, il peut être décrit ainsi :

```
function MULTIPLY(x, y)
    p ← 0
    while x > 0 do
        if x est impair then
            p ← p + y
        x ← ⌊x/2⌋
        y ← y + y
    return p
```

x	y	p
105	253	0
52	506	253
26	1012	253
13	2024	253
6	4048	2277
3	8096	2277
1	16192	10373
0	32384	26565

FIGURE 1 – Calcul de 105×253 par la méthode du paysan russe.

Cependant, il ne saute pas aux yeux que cet algorithme calcule effectivement le produit de x par y ; pour le montrer il faut commencer par observer qu'il réalise l'itération de trois suites $(p_n)_{n\in\mathbb{N}}$, $(x_n)_{n\in\mathbb{N}}$ et $(y_n)_{n\in\mathbb{N}}$ définies par les conditions initiales :

$$p_0 = 0, \quad x_0 = x, \quad y_0 = y$$

et les relations de récurrence :

$$p_{n+1} = \begin{cases} p_n + y_n & \text{si } x_n \text{ est impair} \\ p_n & \text{sinon} \end{cases}, \qquad x_{n+1} = \lfloor x_n/2 \rfloor, \qquad y_{n+1} = 2y_n.$$

Prouver la *terminaison* de l'algorithme, c'est montrer l'existence d'un rang N pour lequel $x_N \leqslant 0$. Prouver sa *validité* c'est montrer que pour ce rang N on a $p_N = xy$.
Pour justifier soigneusement ceci on utilise l'écriture en base 2 de x : posons $x = (b_k b_{k-1} \cdots b_1 b_0)_2$ avec $b_k = 1$. Il devient dès lors très facile de constater que $x_n = (b_k b_{k-1} \cdots b_n)_2$ et que $y_n = 2^n y$. Par ailleurs, la relation de récurrence vérifiée par la suite $(p_n)_{n\in\mathbb{N}}$ peut aussi s'écrire $p_{n+1} = p_n + y_n \times (x_n \bmod 2) = p_n + b_n y_n$.
Nous avons donc $x_k = b_k = 1 \neq 0$ et $x_{k+1} = 0$, ce qui prouve la terminaison de l'algorithme, et :

$$p_{k+1} = p_0 + \sum_{n=0}^{k} b_k (2^k y) = \left(\sum_{n=0}^{k} b_k 2^k \right) y = xy$$

ce qui prouve sa validité.

Cependant, cette preuve, bien que rigoureuse, n'est pas forcément très éclairante, et il est sans doute plus intéressant d'observer que cet algorithme repose sur les relations :

$$x \times y = \begin{cases} 0 & \text{si } x = 0 \\ \lfloor x/2 \rfloor \times (y+y) & \text{si } x \text{ est pair} \\ \lfloor x/2 \rfloor \times (y+y) + y & \text{si } x \text{ est impair} \end{cases}$$

Autrement dit, on ramène le problème du calcul du produit de x par y au produit de $\lfloor x/2 \rfloor$ et de $2y$.
La plus-part des langages de programmation actuels permettent de mettre en œuvre directement cette réduction du problème, en autorisant une fonction à s'appeler elle-même : on parle alors de fonction *récursive*. On trouvera figure 2 les deux versions, itérative et récursive, de la méthode du paysan russe.

```python
def multiply(x, y):
    p = 0
    while x > 0:
        if x % 2 == 1:
            p += y
        x = x // 2
        y = y + y
    return p
```

```python
def multiply(x, y):
    if x <= 0:
        return 0
    elif x % 2 == 0:
        return multiply(x//2, y+y)
    else:
        return multiply(x//2, y+y) + y
```

FIGURE 2 – Les versions itérative et récursive de la méthode du paysan russe.

Nous verrons plus loin comment l'interprète de commande gère les différents appels récursifs ; pour l'instant nous allons mettre en évidence deux éléments indispensables à la terminaison d'une fonction récursive :

- il est nécessaire qu'il y ait une *condition d'arrêt* ;
- il ne doit pas y avoir de suite *infinie* d'appels récursifs.

Dans le cas de la multiplication paysanne, la condition d'arrêt correspond aux couples $\left\{(x,y) \;\middle|\; x \leqslant 0\right\}$ et le nombre d'appels récursifs est fini puisque lorsque $x > 0$ la suite définie par $x_0 = x$ et la relation $x_{n+1} = \lfloor x_n/2 \rfloor$ est une suite qui stationne en 0. On peut même calculer le nombre exact d'appels récursifs : il y en a $\lfloor \log x \rfloor + 1$.

Dans la quasi totalité des algorithmes récursifs que nous rencontrerons il ne sera guère difficile de vérifier ces deux conditions, et si jamais nous faisons une erreur l'interprète de commande nous l'indiquera comme par exemple :

```
>>> def f(n):
        return 1+f(n+1)

>>> f(0)
RuntimeError: maximum recursion depth exceeded
```

Comme nous pouvons le constater, l'interprète PYTHON limite arbitrairement le nombre d'appels récursifs (la valeur par défaut est égale à 1000).
Il faut cependant noter qu'il est aussi très facile de définir des fonctions récursives dont la preuve de terminaison est très délicate à établir. Pour autant que je le sache, la preuve de la terminaison de la fonction Q de HOFSTADTER [1] résiste encore à l'analyse malgré son apparente simplicité :

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

Dans le même style, on pourra chercher les exercices 1 et 2.

1.2 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 ou le contraire. 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 CAML est conçu pour exploiter la récursivité et le programmeur est naturellement amené à choisir la version récursive de

1. Cette fonction, ainsi que celle présentée dans l'exercice **??**, proviennent de son remarquable ouvrage : *Gödel, Escher, Bach : Les Brins d'une Guirlande Éternelle*.

l'algorithme qu'il souhaite écrire. À l'inverse, Python, même s'il l'autorise, ne favorise pas l'écriture récursive [2] (limitation basse par défaut du nombre d'appels récursifs, pas d'optimisation pour la récursivité terminale).

Enfin, le choix d'écrire une fonction récursive ou itérative peut dépendre du problème à résoudre : certains problèmes se résolvent particulièrement simplement sous forme récursive, et le plus emblématique de tous 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.

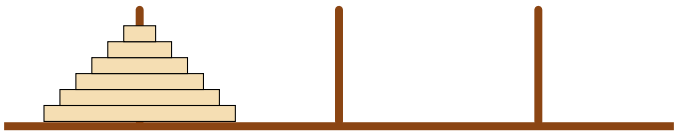
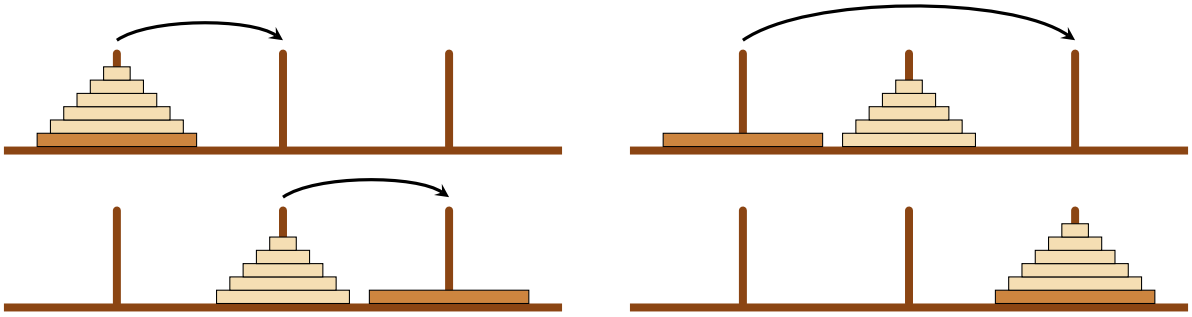


Figure 3 – Le puzzle dans sa configuration initiale.

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 4 un exemple de résolution pour $n = 4$.

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).

2. On peut citer à ce sujet Guido van Rossum, le créateur du langage Python : *I don't believe in recursion as the basis of all programming. This is a fundamental belief of certain computer scientists, especially those who (...) like to teach programming by starting with a "cons" cell and recursion. But to me, seeing recursion as the basis of everything else is just a nice theoretical approach to fundamental mathematics (...), not a day-to-day tool.*

```
>>> 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 4 – Une solution du problème des tours de Hanoï avec quatre disques.

1.3 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.

```
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:]
            break
        elif i >= p:
            c[k:] = b[j:]
            break
        elif a[i] < b[j]:
            c[k] = a[i]
            i += 1
        else:
            c[k] = b[j]
            j += 1
    return c
```

```
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.

La fonction `merge` fusionne deux tableaux triés par ordre croissant dans un troisième tableau trié lui aussi par ordre croissant (cette fonction est séparée du corps principal de l'algorithme pour accroitre la lisibilité de la structure récursive).

Quel est le coût temporel 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 le coût du tri d'un tableau de longueur n, on dispose de la relation :

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

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

1.4 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 un parcours d'un chemin ayant une origine et une extrémité. Lors de l'appel d'une fonction, ce flux est interrompu le temps de l'exécution de cette fonction, avant de reprendre à l'endroit où le programme s'est arrêté.

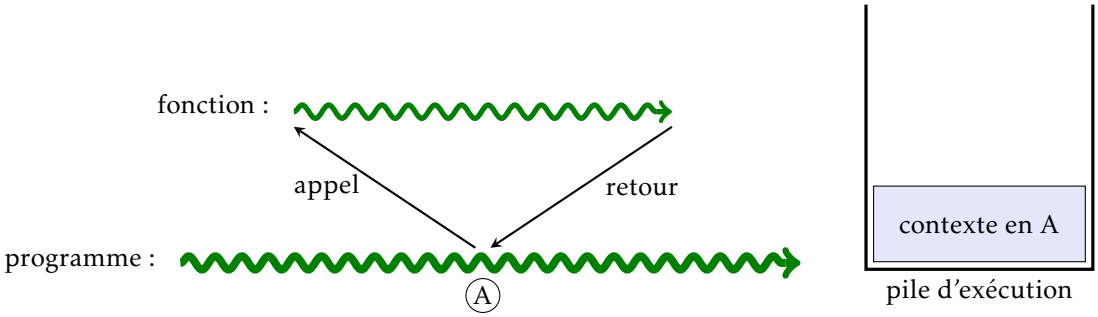


Figure 6 – L'exécution d'une fonction au sein d'un programme.

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* [3]. 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.

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 `multiply` dont le code est présenté figure 2, avec pour paramètres $x = 105$ et $y = 253$ (pour des raisons de lisibilité, seules les valeurs de ces deux paramètres sont présentées dans le contexte).

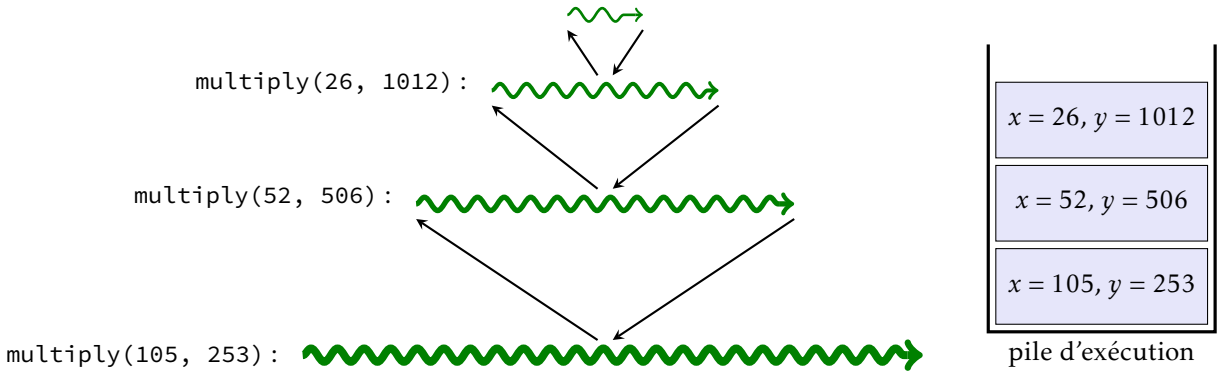


Figure 7 – Calcul récursif de 105×253 par la méthode du paysan russe.

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 du coût d'une fonction récursive.

3. 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.

1.5 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.

Cette fonctionnalité étant malheureusement absente en Python, nous allons devoir la créer nous mêmes, ce qui va être l'occasion d'aborder une nouvelle notion en Python : les décorateurs.

Syntaxe
Un décorateur est tout simplement une fonction qui prend en argument une fonction et renvoie une nouvelle fonction. Par exemple, si `madeco` est un décorateur et si je définis une fonction avec la syntaxe :

```
@madeco
def mafonction(...):
    ....
```

l'effet de la décoration va être de remplacer la fonction `mafonction` par le résultat de `madeco(mafonction)`. Typiquement, un décorateur va ajouter du code avant et après l'exécution de la fonction en utilisant la syntaxe suivante [4] :

```
def madeco(func):
    def wrapper(args):
        # ici le code à exécuter avant la fonction
        func(args)
        # ici le code à exécuter après la fonction
    return wrapper
```

En ce qui nous concerne, nous allons ajouter avant l'exécution d'une fonction la liste de ses paramètres, et après l'exécution le résultat de la fonction, à l'aide du décorateur suivant :

```
def trace(func):
    def wrapper(*args):
        print(' ' * wrapper.space, end='')
        print('{} <- {}'.format(func.__name__, str(args)))
        wrapper.space += 1
        val = func(*args)
        wrapper.space -= 1
        print(' ' * wrapper.space, end='')
        print('{} -> {}'.format(func.__name__, str(val)))
        return val
    wrapper.space = 0
    return wrapper
```

(L'attribut `space` permet de décaler les différents niveaux d'imbrication des appels récursifs.)
Il suffit désormais d'ajouter `@trace` devant la définition d'une fonction récursive pour visualiser la trace des appels de celle-ci.
On trouvera figure 8 le résultat des codes `multiply(105, 253)` et `mergesort([6, 2, 4, 3, 5, 1])`.

1.6 Récursivité terminale

Il y a un cas où il n'est pas nécessaire de garder mémoire du contexte lorsqu'on fait appel à une fonction : c'est quand cet appel est la *dernière* instruction du programme. Une fonction *récursive terminale* est une fonction pour laquelle l'appel récursif est la dernière instruction à être évaluée ; dans ce cas il n'est en théorie pas nécessaire de stocker le contexte dans la pile d'exécution de la fonction. Ainsi, une fonction récursive terminale peut en principe *ne pas engendrer de coût spatial*.
Un exemple typique de fonction récursive terminale est le calcul du pgcd par la méthode d'Euclide (voir figure 9). La trace de cette fonction est éloquente : les retours ne font que transmettre le résultat au niveau suivant.

Les langages fonctionnels (et Caml en particulier) peuvent généralement détecter la récursivité terminale et optimiser son exécution en transformant la récursivité en itération, ce qui permet d'économiser l'espace de

4. *Wrapper* signifie « emballage » en français.

```
multiply <- (105, 253)
 multiply <- (52, 506)
  multiply <- (26, 1012)
   multiply <- (13, 2024)
    multiply <- (6, 4048)
     multiply <- (3, 8096)
      multiply <- (1, 16192)
       multiply <- (0, 32384)
       multiply -> 0
      multiply -> 16192
     multiply -> 24288
    multiply -> 24288
   multiply -> 26312
  multiply -> 26312
 multiply -> 26312
multiply -> 26565
```

```
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 `multiply` et `mergesort`.

```
@trace
def pgcd(a, b):
    if b == 0:
        return a
    return pgcd(b, a % b)
```

```
pgcd <- (132, 48)
 pgcd <- (48, 36)
  pgcd <- (36, 12)
   pgcd <- (12, 0)
   pgcd -> 12
  pgcd -> 12
 pgcd -> 12
pgcd -> 12
```

Figure 9 – La fonction pgcd est récursive terminale.

la pile d'exécution (le contexte n'est plus sauvegardé) et donc permettre d'envisager des appels récursifs très nombreux sans craindre l'épuisement de l'espace alloué à la pile. Malheureusement, Guido van Rossum, le créateur de Python, est farouchement opposé à l'optimisation de la récursivité terminale, et il n'y a donc aucun intérêt en Python de chercher à obtenir des versions terminales des algorithmes récursifs. Nous n'en parlerons donc plus.

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.

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 déterminer si x se trouve dans t. La démarche est connue, et vous l'avez déjà 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 10.

```
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 10 – 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 le coût est donc de la forme : $C(n) = C(n/2) + O(n)$, ce qui conduit à $C(n) = O(n)$. Cet algorithme est de coût 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 $t[k+1 \dots j-1]$.

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

FIGURE 11 – Une version récursive correcte de la recherche dichotomique.

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

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 12) et

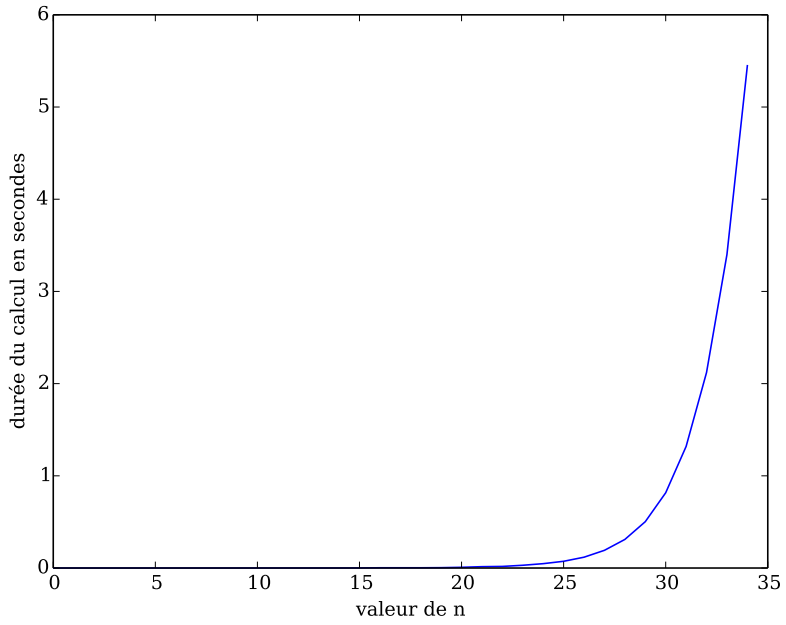


FIGURE 12 – Temps d'exécution en secondes pour calculer f_n par l'algorithme récursif.

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 :

```
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
```

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.
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 $f_n \sim \dfrac{1}{\sqrt{5}}\varphi^n$ (où φ est le nombre d'or) on obtient $a_n = \Theta(\varphi^n)$; 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. La solution mathématique consiste à itérer la suite de vecteurs (f_n, f_{n+1}) de manière à ne plus avoir qu'un seul appel récursif :

$$(f_0, f_1) = (0, 1) \qquad \text{et} \qquad (f_n, f_{n+1}) = (f_n, f_{n-1} + f_n) = \varphi(f_{n-1}, f_n)$$

```
def fib(n):
    def aux(n):
        if n == 0:
            return (0, 1)
        else:
            (x, y) = aux(n-1)
            return (y, x + y)
    return aux(n)[0]
```

mais nous allons rejeter cette solution, qui s'éloigne de la simplicité de la première version récursive que nous souhaitons préserver.

La solution 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 clé à une valeur. Dans le cas qui nous intéresse, la clé est l'entier n et la valeur, l'entier f_n.

- `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 clé 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) ;
- `del d[c4]` supprime une association.

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

FIGURE 13 – Un rappel des principales fonctions des dictionnaires.

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(10)
55
>>> d_fib
{0: 0, 1: 1, 2: 1, 3: 2, 4: 3, 5: 5, 6: 8, 7: 13, 8: 21, 9: 34, 10: 55}
```

Décorateur et mémoïsation
Il est possible de donner une solution encore plus élégante en utilisant un décorateur, bien adapté à la mémoïsation. En effet, il suffit de vérifier l'existence d'une association *avant* l'application de la fonction, et d'ajouter une paire d'association *après* avoir calculé la nouvelle valeur. Ceci nous conduit à définir le décorateur suivant :

```
def memoise(func):
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper
```

Il suffit dès lors de décorer une définition récursive pour obtenir un algorithme aussi efficace qu'un algorithme itératif (mais au prix bien sûr d'un coût spatial) :

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

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 :

```
@memoise
def binom(n, p):
    if p == 0 or n == p:
        return 1
    return binom(n-1, p-1) + binom(n-1, 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 1 On considère la fonction récursive suivante :

```
def f(n):
    if n > 100:
        return n-10
    return f(f(n + 11))
```

Prouver sa terminaison lorsque $n \in \mathbb{N}$, et déterminer ce qu'elle calcule (sans utiliser l'interprète de commande).

Exercice 2 Prouver la terminaison de la fonction G de Hofstadter, définie sur $\mathbb{N}$ de la façon suivante :

```
def g(n):
    if n == 0:
        return 0
    return n - g(g(n - 1))
```

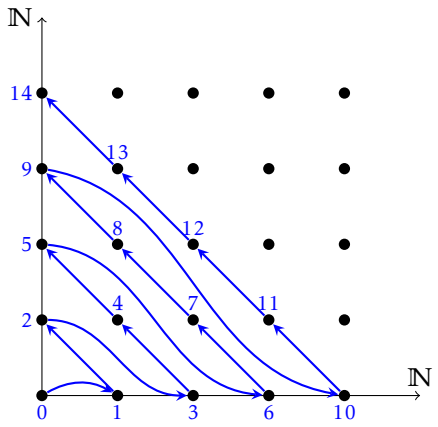
(Difficile.) Prouver que $g(n) = \left\lfloor \frac{n+1}{\alpha} \right\rfloor$ où α est le nombre d'or.

Exercice 3 Écrire une fonction récursive qui calcule a^n en exploitant la relation : $a^n = a^{\lfloor n/2 \rfloor} \times a^{\lceil n/2 \rceil}$, puis une seconde fonction qui utilise en plus la remarque suivante :

$$\lceil n/2 \rceil = \begin{cases} \lfloor n/2 \rfloor & \text{si } n \text{ est pair} \\ \lfloor n/2 \rfloor + 1 & \text{sinon} \end{cases}$$

Évaluer le nombre de multiplications effectuée dans les deux cas.

Exercice 4 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 :



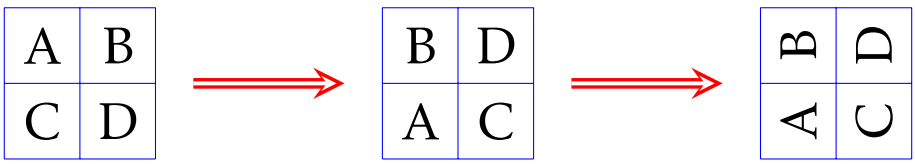
Rédiger une fonction *récursive* qui retourne le numéro du point de coordonnées (x, y).
Rédiger la fonction réciproque, là encore de façon récursive.

Exercice 5 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 6 Les processeurs graphiques possèdent en général une fonction de bas niveau appelée *blit* (ou transfert de bloc) qui copie rapidement un bloc rectangulaire d'une image d'un endroit à un autre.
L'objectif de cet exercice est de faire tourner une image carrée de $n \times n$ pixels de 90° dans le sens direct en adoptant une stratégie récursive : découper l'image en quatre blocs de tailles $n/2 \times n/2$, déplacer chacun des ces blocs à sa position finale à l'aide de 5 blits, puis faire tourner récursivement chacun de ces blocs :



On supposera dans tout l'exercice que n est une puissance de 2.

a) Exprimer en fonction de n le nombre de fois que la fonction blit est utilisée.

b) Quel est le coût total de cet algorithme lorsque le coût d'un blit d'un bloc $k \times k$ est en $\Theta(k^2)$?

c) Et lorsque ce coût est en $\Theta(k)$?

En supposant qu'une image est représentée par une matrice numpy $n \times n$, rédiger une fonction qui adopte cette démarche pour effectuer une rotation de 90° dans le sens direct (on simulera un blit par la copie d'une partie de la matrice vers une autre en décrivant ces parties par le *slicing*).

Exercice 7 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).

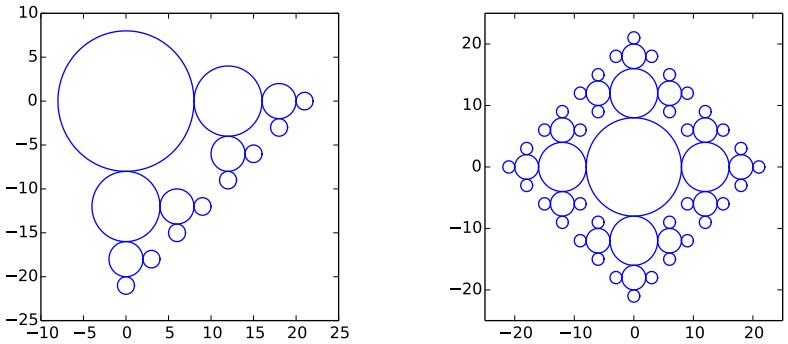


FIGURE 14 – Le résultat des fonctions `bubble1(4)` et de `bubble2(4)`.

Exercice 8 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).

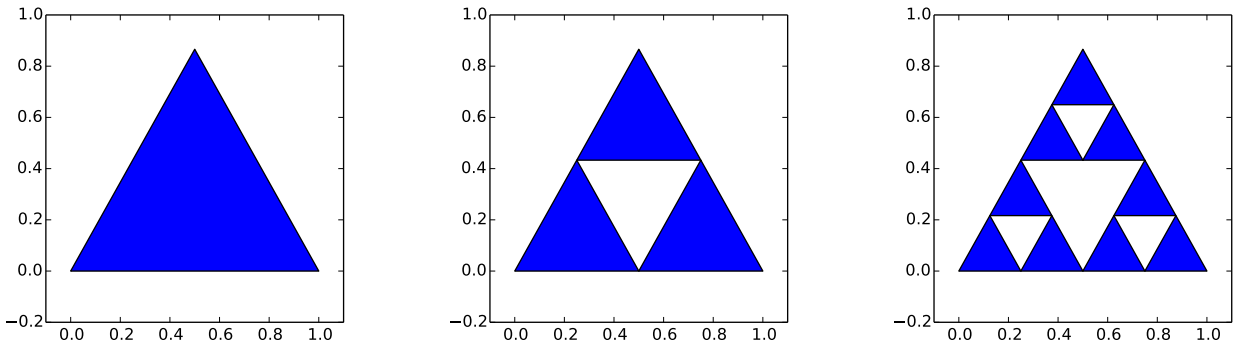


FIGURE 15 – Le résultat des fonctions `sierpinski(n)` pour $n = 1, 2, 3$.