2
Récursivité

Sommaire

2.1 Principe de la récursivité

2.1.1 Définition et premiers exemples

Définition 2.1 (Fonction récursive).

Une fonction est dite récursive si celle-ci s'appelle elle-même.

Comme premier exemple de fonction récursive nous allons (re)-voir la fonction factorielle. Implémentée de manière itérative cette fonction se présente sous la forme :

Algorithmes 1 (Factorielle itérative).

```
def fact(n):
    assert n >= 0
    f = 1
    for k in range(2,n+1):
        f *= k
    return f
```

Cette implémentation est directement inspirée de la définition suivante de la factorielle, pour $n \geq 0$:

$$n! = \begin{cases} 1 & \text{si} \quad n = 0 \\ \prod_{k=1}^{n} k & \text{sinon.} \end{cases}$$

Une autre définition de la factorielle est la suivante :

$$n! = \begin{cases} 1 & \text{si} \quad n = 0 \\ n \times (n-1)! & \text{sinon.} \end{cases}$$

Et celle-ci donne lieu à l'implémentation suivante :

Algorithmes 2 (Factorielle récursive).

```
def fact_rec(n):
    assert n >= 0
    if n == 0:
        res = 1
    else:
        res = n * fact_rec(n-1)
    return res
```

On y retrouve bien les deux cas $n = 0$ et $n > 0$. Le premier correspond à ce que l'on appelle le cas final ou encore la condition d'arrêt, le second à l'appel récursif de la fonction par elle-même.

Toutes les fonctions récursives présentent ces deux cas. Dans cet exemple, il est aisé de voir qu'il n'y aura qu'un nombre fini d'appels récursifs, ce qui prouve la terminaison de l'algorithme. Mais, même si c'est plus rare, il existe cependant des fonctions récursives pour lesquelles prouver la terminaison est extrêmement difficile. Par exemple, la terminaison de la fonction Q de Hofstadter, dont la définition est très simple, résiste toujours aux chercheurs.

Algorithmes 3 (Fonction Q de Hofstadter).

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

Cette fonction est un exemple de « récursivité mutliple », c'est-à-dire d'une fonction qui réalise plusieurs appels à elle même. Il existe encore d'autre types de récursivité, pour n'en citer que deux : la récursivité croisée et la récursivité imbriquée, dont voici des exemples[i] :

Algorithmes 4 (Exemples).

```
def is_even(n):
    if n == 0:
        res = True
    else:
        res = is_odd(n-1)
    return res

def is_odd(n):
    if n == 0:
        res = False
    else:
        res = is_even(n-1)
    return res
```

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

Donnons un autre exemple celui de la technique de multiplication du paysan russe[ii]. Son principe est simple, il repose sur la disjonction de cas suivants :

$$a \times b = \begin{cases} 0 & \text{si} \quad a = 0 \\ \lfloor a/2 \rfloor \times (b+b) & \text{si} \quad a \text{ est pair} \\ \lfloor a/2 \rfloor \times (b+b) + b & \text{si} \quad a \text{ est impair} \end{cases} .$$

Suivant ce principe voici une implémentation récursive de cet algorithme :

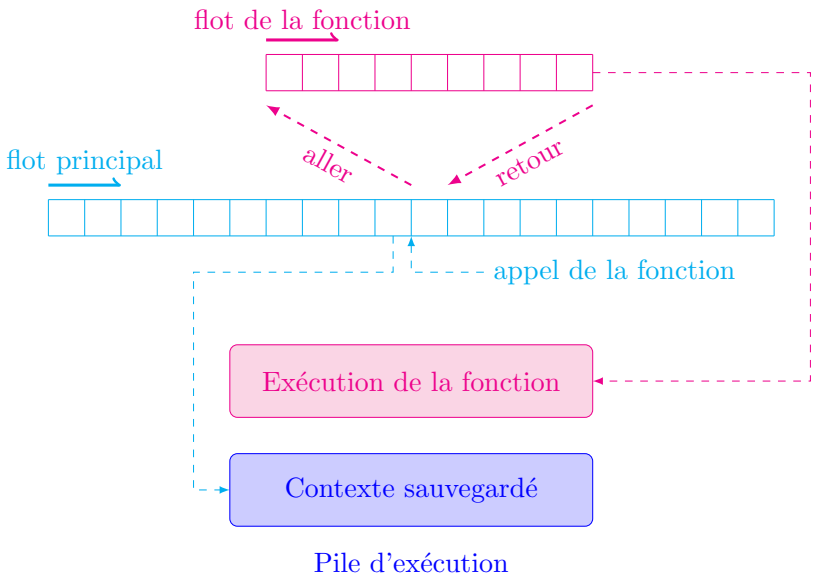
i. La fonction f est la célèbre fonction 91 introduite par Mc Carthy.
ii. Il s'agit d'un algorithme déjà décrit près de 2000 ans avant J.C.

Algorithmes 5 (Multiplication du paysan russe).

```
def russian_multi(a,b):
    if a == 0:
        m = 0
    elif a % 2 == 0:
        m = multi_russe(a//2 , b+b)
    else:
        m = multi_russe(a//2, b+b) + b
    return m
```

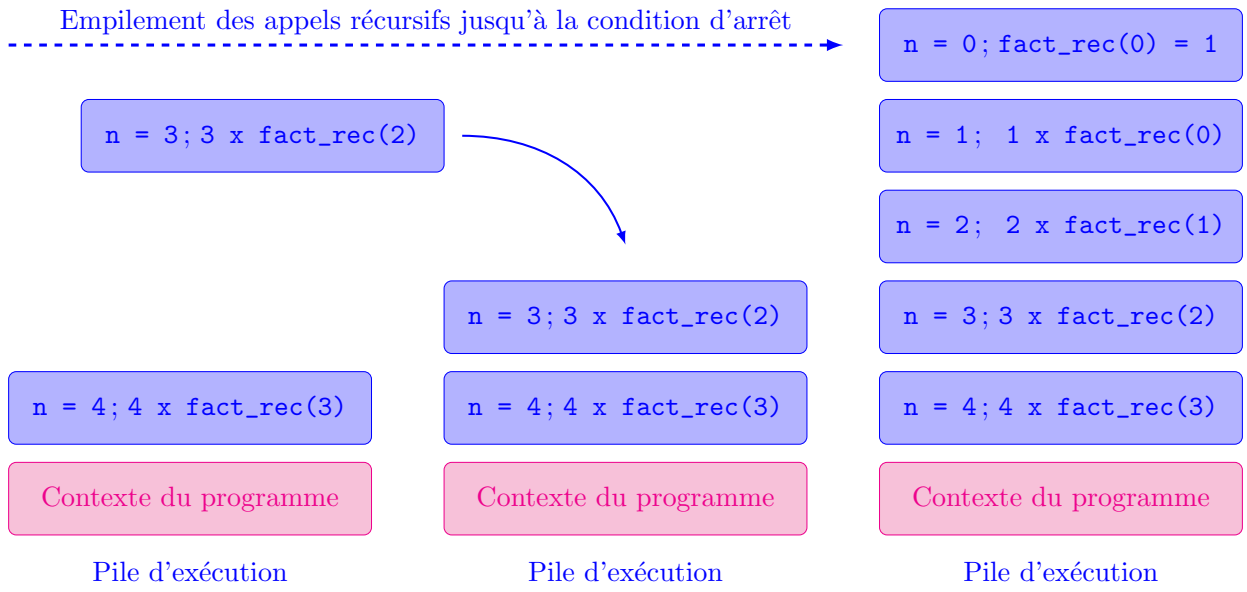
2.1.2 La pile d'exécution ou pile d'appels

En informatique, un programme s'exécute de manière séquentielle, toutes les instructions sont exécutées les unes après les autres dans leur ordre d'arrivée, on appelle ça le flot [iii] d'instructions. Lorsque qu'un appel à une fonction est passé, le flot principal du programme est interrompu et son état à cet instant ainsi que d'autres informations comme l'adresse de retour, qui forment ce qu'on appelle le contexte est sauvegardé. Dès que l'exécution de la fonction s'achève on revient à l'endroit où le flot principal s'est arrêté.



Les informations du contexte sont alors empilées sur la pile d'exécution ou pile d'appels, puis l'exécution de la fonction elle-même est empilée. Lorsque l'exécution de la fonction est achevée cette tâche est dépilée, puis le contexte précédent l'est pour être rétabli et enfin le programme se poursuit.

iii. Ce qui fait référence à l'écoulement d'un liquide. On dit aussi flux.



Après l'appel à une fonction récursive, les appels récursifs vont être empilés l'un après l'autre sur la pile d'exécution jusqu'à ce que la condition d'arrêt soit atteinte. Puis ils seront tous dépilés, jusqu'au premier appel à la fonction, puis le contexte du programme le sera avant de se poursuivre.

Il apparait immédiatement que le nombre d'appels récursifs pouvant être très important, la pile d'exécution peut prendre beaucoup de place en mémoire. Le coût spatial d'une fonction récursive n'est en général pas négligeable. Si bien que pour se protéger du dépassement de capacité de la pile, le « Stack Overflow », Python n'autorise pas plus de 1000 appels récursifs par défaut...

Le coup n'est pas seulement spatial, le temps de d'enregistrement du contexte principal et du contexte de chaque fonction avant l'appel récursif peut être important.

Aussi, comme Python n'est pas optimisé pour la programmation récursive, si une forme itérative d'une fonction est facile à implémenter il faut la préférer à la forme récursive dans quasiment tous les cas.

```
>>> def deep(n):
...     return deep(n+1)
...
>>> deep(0)

RecursionError: maximum recursion depth exceeded while calling a Python object
```

2.2 Trois autres exemples

2.2.1 L'algorithme d'Euclide

Le lemme d'Euclide affirme que si $a, b \in \mathbb{N} \setminus \{0\}$, avec $a = bq + r$ et $0 \leq r \leq b - 1$, alors $a \wedge b = b \wedge r$. Ce qui permet d'écrire une première fonction itérative qui calcule le pgcd de deux nombres basée sur des divisions euclidienne successives.

Algorithmes 6 (Algorithme d'Euclide itératif).

```
def pgcd(a : int,b : int)->int:
    while b > 0:
        a, b = b, a % b
    return b
```

Et sa version récursive :

Algorithmes 7 (Algorithme d'Euclide récursif).

```
def pgcd_rec(a : int,b : int)->int:
    if b == 0:
        pgcd = a
    else:
        pgcd = pgcd_rec(b, a % b)
    return pgcd
```

La version récursive de cet algorithme a une particularité, l'appel récursif est la dernière instruction à être évaluée dans la fonction, ce qui fait qu'il n'est pas réellement nécessaire de sauvegarder le contexte de chaque appel récursif. C'est ce qu'on appelle une fonction récursive terminale. Malheureusement Python ne sait pas optimiser ce genre de fonction [iv]. En revanche, la fonction `fact_rec` n'est pas terminale puisque elle retourne `n*fact_rec(n-1)`.

2.2.2 Le tri fusion (merge sort)

Un des plus anciens algorithmes de tri d'un tableau est le tri fusion, proposé par Von Neuman dès 1945. Il fait partie de la famille des algorithmes « diviser pour régner », dont le principe est de diviser un problème en sous-problèmes plus petits, de résoudre ces sous-problèmes puis de combiner les solutions pour obtenir une solution du problème globale.

Cet algorithme et tous ceux de cette famille se prêtent particulièrement bien à des implémentations récursives. Son principe est le suivant :

- si le tableau n'a qu'un élément : il est trié.
- sinon :
  - (Diviser) diviser le tableau en deux parties à peu près égales.
  - (Régner) trier récursivement les deux parties de tableau par tri fusion.
  - (Combiner) fusionner les deux tableaux triés en un tableau trié.

iv. Le créateur de Python Guido Van Rossum considère que la récursivité est une affaire de mathématicien.

Algorithmes 8 (Fusion des tableaux triés en un tableau trié).

```
def merge(tab : list, i : int, j : int, k :int )-> None:
    l = i
    r = j
    B = []
    for m in range(i,k):
        if  r == k or (l < j and tab[l] <= tab[r]):
            B.append(tab[l])
            l += 1
        else:
            B.append(tab[r])
            r += 1
    tab[i:k] = B
```

Algorithmes 9 (Tri fusion).

```
def merge_sort(tab : list, i : int, k : int)-> None:
    if k-i > 1:
        j = (i+k)//2
        merge_sort(tab, i, j)
        merge_sort(tab, j, k)
        merge(tab, i, j, k)
```

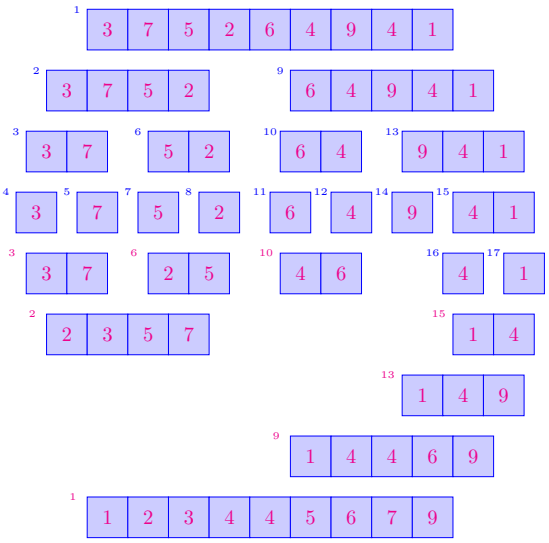


FIGURE 2.1 – Exemple de tri par tri fusion. Les numéros bleus correspondent aux appels récursifs et les rouges aux résultats de ces appels.

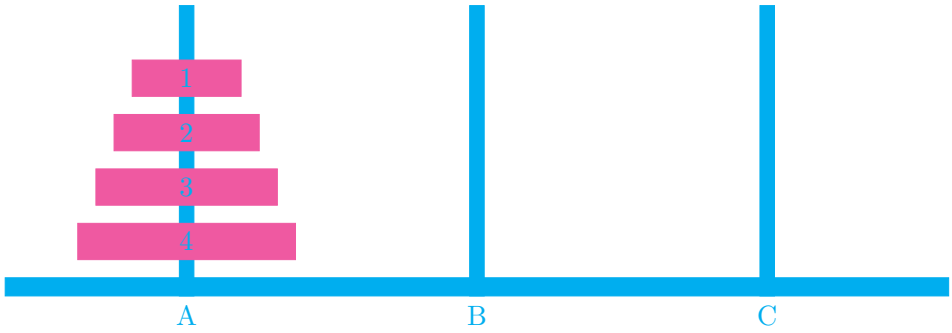
2.2.3 Les Tours de Hanoï

Un des problème qui se prête le mieux à une résolution récursive est celui des Tours de Hanoï. Petite rappel : Les tours de Hanoï (originellement, la tour d'Hanoïa) sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes :

(a) on ne peut déplacer plus d'un disque à la fois ;

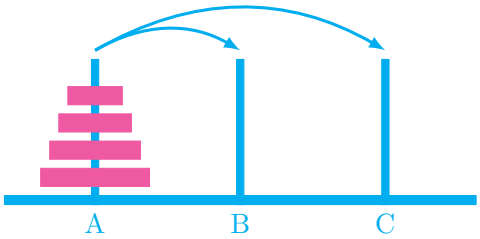
(b) on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ.
(Source : Wikipédia)

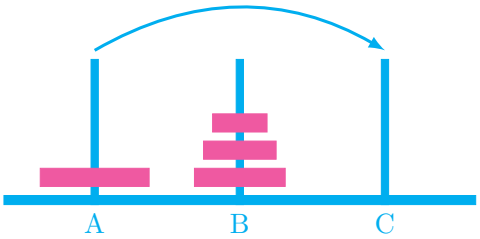


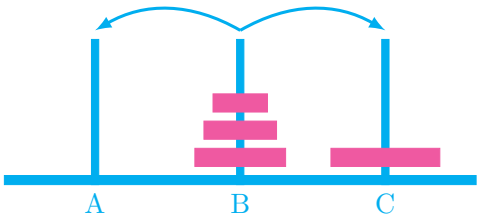
Pour déplacer le dernier disque sur la tige C, il faut d'abord déplacer les $n-1$ disques posés sur celui-ci de la tige A à la tige B, en utilisant la tige C comme pivot. Puis il reste à déplacer les $n-1$ disques sur la tige C en utilisant le tige A comme pivot.

Utilisation de deux pivots pour déplacer $n-1$ disques en B

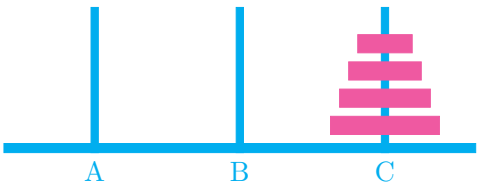


Déplacement du n-ième disque





Utilisation de deux pivots pour déplacer $n-1$ disques en C



Gagné !

Aussi pour déplacer les n disques de la tige A vers la tige C, il suffit de savoir déplacer $n-1$ disques de la tige A à la tige B.

Dès qu'on a compris le principe, il suffit de réduire le problème au déplacement de n disques de la tige A à la tige C en utilisant la tige B comme pivot.

Algorithmes 10 (tower of Hanoi).

```
def tower_of_hanoi(n, a="A", b="B", c="C"):
    if n == 0:
        return None
    tower_of_hanoi(n-1, a, c, b)
    print("Déplacement du disque {} de la tige {} vers la tige {}.".format(n, a, c))
    tower_of_hanoi(n-1, b, a, c)
```

C'est la « magie » du récursif, nous avons simplement réduit le problème des n disques à deux sous-problèmes identiques à $n-1$ disques, puis à deux sous-problèmes à $n-2$ disques, etc... Jusqu'à aboutir une problèmes de taille élémentaire trivialement résolubles. Cette idée est un paradigme de programmation que l'on nomme « diviser pour règner ». Nous reviendrons en détails sur ce principe dans un chapitre ultérieur.

```
>>> tower_of_hanoi(4)
Déplacement du disque 1 de la tige A vers la tige B.
Déplacement du disque 2 de la tige A vers la tige C.
Déplacement du disque 1 de la tige B vers la tige C.
Déplacement du disque 3 de la tige A vers la tige B.
Déplacement du disque 1 de la tige C vers la tige A.
Déplacement du disque 2 de la tige C vers la tige B.
Déplacement du disque 1 de la tige A vers la tige B.
Déplacement du disque 4 de la tige A vers la tige C.
Déplacement du disque 1 de la tige B vers la tige C.
Déplacement du disque 2 de la tige B vers la tige A.
Déplacement du disque 1 de la tige C vers la tige A.
Déplacement du disque 3 de la tige B vers la tige C.
Déplacement du disque 1 de la tige A vers la tige B.
Déplacement du disque 2 de la tige A vers la tige C.
Déplacement du disque 1 de la tige B vers la tige C.
```

On montre facilement par récurrence que le nombre de mouvements produit par notre fonction est de $2^n - 1$, ce qui est optimal. Sa complexité est donc exponentielle.

2.3 Les limites de récursivité

2.3.1 Appels multiples

L'exemple le plus célèbre certainement du chevauchement des appels récursifs est celui de la suite de Fibonacci. Le problème est de calculer le n-ième terme de la suite définie par :

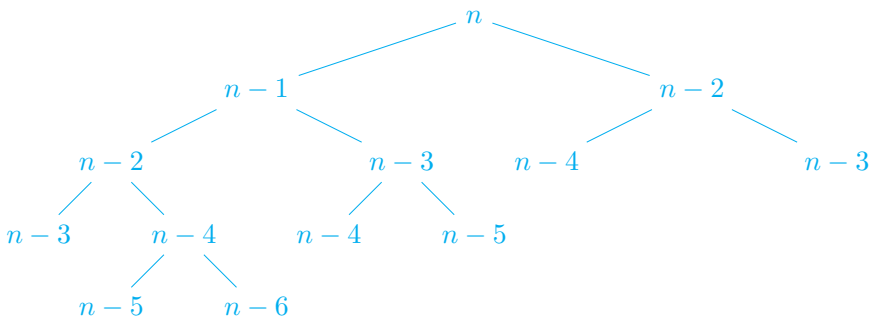
$$F_n = \begin{cases} 0 & \text{si} \quad n = 0 \\ 1 & \text{si} \quad n = 1 \\ F_{n-1} + F_{n-2} & \text{sinon.} \end{cases}$$

Tout naturellement on programme la fonction suivante.

Algorithmes 11.

```
def fibo_rec(n):
    if n in [0,1]:
        res = n
    else:
        res = fibo_rec(n-1) + fibo_rec(n-2)
    return res
```

Observons l'arbre des appels récursifs pour calculer F_n :



Il semble que le nombre d'appels devient rapidement très important, et surtout que certains calculs sont effectués à de nombreuses reprises. Si l'arbre ne nous convainc pas regardons combien d'appels sont effectués pour calculer F_{40} :

Exemple 2.1 (Calcul de F_{40}).

```
d = {}
calls = 0
def fibo_rec(n):
    global calls
    calls += 1
    d[n] = d.get(n,0)+1
    if n in [0,1]:
        res = n
    else:
        res = fibo_rec(n-1) + fibo_rec(n-2)
    return res

fibo_rec(40)
print("calls : ",  calls)
print(d)
```

```
calls : 331160281
{40: 1, 39: 1, 38: 2, 37: 3, 36: 5,
35: 8, 34: 13, 33: 21, 32: 34,
31: 55, 30: 89, 29: 144, 28: 233,
27: 377, 26: 610, 25: 987,
24: 1597, 23: 2584, 22: 4181,
21: 6765, 20: 10946, 19: 17711,
18: 28657, 17: 46368,
16: 75025, 15: 121393, 14: 196418,
13: 317811, 12: 514229, 11: 832040,
10: 1346269, 9: 2178309,
8: 3524578, 7: 5702887,
6: 9227465, 5: 14930352,
4: 24157817, 3: 39088169,
2: 63245986, 1: 102334155, 0: 63245986}
```

Si u_n est le nombre d'appels récursifs pour calculer F_n, il apparait que $u_0 = u_1 = 0$, et que pour $n \geq 2$, $u_n = 2 + u_{n-1} + u_{n-2}$. Ce qui donne $u_n + 2 = (u_{n-1} + 2) + (u_{n-2} + 2)$. La résolution de la relation de récurrence double donne alors :

$$u_n = \frac{2}{\sqrt{5}}\left(\varphi^{n+1} - (1-\varphi)^{n+1}\right) - 2 \text{ avec } \varphi = \frac{1+\sqrt{5}}{2}.$$

Ce qui conduit à l'équivalent $u_n \sim C\varphi^n$, où $C > 0$.

La complexité $T(n)$ de cet algorithme de calcul de F_n vérifie donc $T(n) = \Theta(F_n)$ et est donc exponentielle...

On peut aussi trouver un encadrement de cette complexité en écrivant que :

$$T(n) = \begin{cases} \Theta(1) & \text{si} \quad n < 2 \\ T(n-1) + T(n-2) & \text{sinon.} \end{cases}.$$

Et sans faire de calcul, on encadre :

$$2T(n-2) < T(n) < 2T(n-1),$$

donc $\Theta(\sqrt{2}^n) < T(n) < \Theta(2^n)$.

C'est pourquoi personne n'implémente le calcul de F_n de manière récursive.

Une solution pour palier le problème du coût spatial et temporel réside dans ce qu'on appelle mémoïsation : il s'agit simplement d'enregistrer les résultats des calculs au fur et à mesure qu'ils sont faits, de manière à ne pas les refaire inutilement.

Algorithmes 12.

```
cache = {0: 0, 1: 1}
def fibo_m(n):
    if n in cache:
        res = cache[n]
    else:
        cache[n] = fibo_m(n-1)+cache[n-2]
        res =  cache[n]
    return res
```

Observons qu'avec cet algorithme chaque sous-problème n'est traité qu'une seule fois, pour calculer F_40 :

Exemple 2.2 (Calcul de F_{40} avec un cache.)**.**

```
calls :  40
{40: 1, 39: 1, 38: 1, 37: 1, 36: 1, 35: 1, 34: 1, 33: 1, 32: 1, 31: 1,
 30: 1, 29: 1, 28: 1, 27: 1, 26: 1, 25: 1, 24: 1, 23: 1, 22: 1, 21: 1,
 20: 1, 19: 1, 18: 1, 17: 1, 16: 1, 15: 1, 14: 1, 13: 1, 12: 1, 11: 1,
 10: 1, 9: 1, 8: 1, 7: 1, 6: 1, 5: 1, 4: 1, 3: 1, 2: 1, 1: 1}
```

On peut gagner en complexité spatiale, si l'on remarque que seule les deux dernières valeurs sont nécessaires au calcul du terme suivant. La version itérative que l'on obtient est de complexité linéaire.[v] :

v. Nous verrons l'année prochaine que c'est la programmation dynamique qui nous y fera penser

Algorithmes 13.

```
def fibo(n):
    assert n >= 0
    a, b = 1, 1
    for __ in range(n):
            a, b = b, a + b
    return b
```

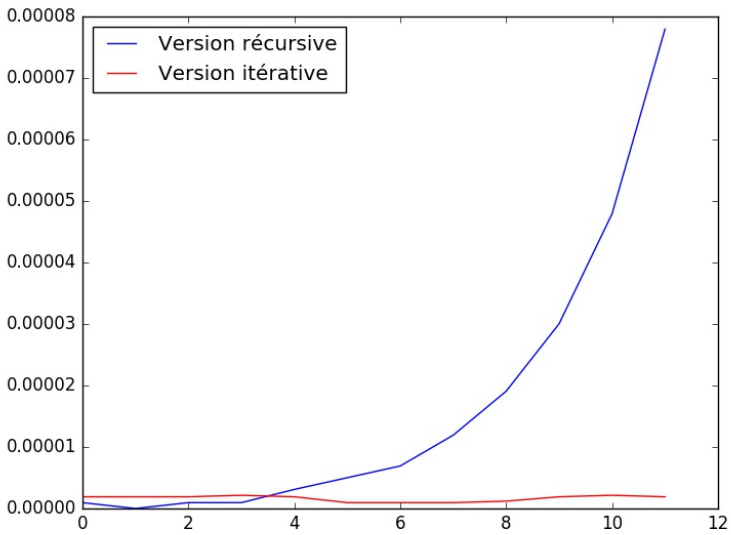


FIGURE 2.2 – Comparaison des temps de calcul suivant les deux méthodes.

2.3.2 Les coûts cachés

Nous avons déjà vu l'algorithme de recherche par dichotomie dans une liste triée. Si `liste` est une liste de nombres triée par ordre croissant et x un élément, pour savoir si x est dans `liste` on procède comme suit : si n est la taille de la liste on pose $k = \lfloor n/2 \rfloor$:

(a) si $x =$ `liste[k]` c'est fini

(b) si $x <$ `liste[k]` on recommence avec la liste `liste[0:k]`

(c) sinon on recommence avec la liste `liste[k+1:n]`.

Cet algorithme se traduit très simplement en :

Algorithmes 14 (Recherche dichotomique dans une liste triée (mauvaise implémentation)).

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

Si cet algorithme est très simple, il n'est pas plus efficace qu'un algorithme de recherche par comparaison terme à terme dans une lise non triée. En effet la copie des listes `liste[:k]` et `liste[k+1:]` a un coût temporel, et spatial, en $O(n)$. La relation de récurrence que vérifie la complexité au pire de notre algorithme est donc $T(n) = T(n/2) + O(n)$, ce qui donne bien $O(n)$.

Pour palier le problème de la recopie des tableaux, il faut donc implémenter une nouvelle version de l'algorithme qui s'en passe.

Algorithmes 15 (Recherche dichotomique dans une liste triée).

```
def dichotomic_search2(x, liste, *args):
    if len(args) == 0:
        i, j = 0, len(liste)
    else:
        i, j = args
        if j <= i:
            res = False
        k = (i + j) // 2
        if x == liste[k]:
            res = True
        elif x < liste[k]:
            res = dichotomic_search2(x, liste, i, k)
        else:
            res = dichotomic_search2(x, liste, k+1, j)
        return res
```

Comme toutes les opérations avant l'appel récursif sont en $O(1)$, la relation de récurrence de sa complexité est $T(n) = T(n/2) + O(1)$, ce qui donne bien $O(\ln(n))$.

2.4 Exercices

Exercice 2.1.

Implémenter une version itérative de la multiplication du paysan russe.

Exercice 2.2.

Déterminer la complexité de la fonction `fact_rec`.

Exercice 2.3.

Prouver la terminaison de la fonction `dicho_rec2`.

Exercice 2.4.

On considère la fonction :

```
def f(a,b):
    """ a et b sont des entiers positifs """
    if b == 1:
        return a
    return a + f(a, b-1)
```

(a) Que retourne `f(2,3)` ?
(b) Quelle est la condition d'arrêt de cette fonction ?
(c) Prouver la terminaison de cette fonction.
(d) Que retourne `f(a,b)` ?

Exercice 2.5.

Écrire une fonction récursive qui détermine le maximum d'une liste de nombres.

Exercice 2.6.

Écrire une fonction récursive qui teste si une liste de nombres est triée de manière croissante.

Exercice 2.7.

Écrire une fonction récursive qui calcule $\binom{n}{k}$ d'après la formule de Pascal.

Exercice 2.8.

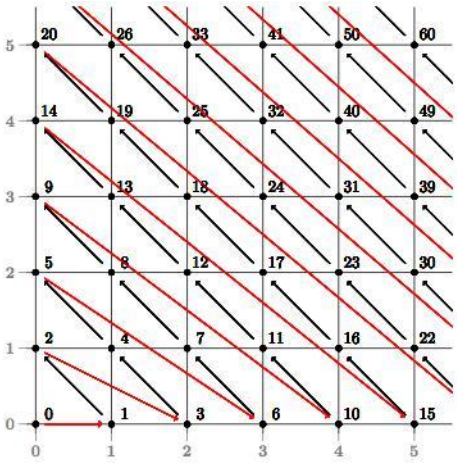
Écrire une fonction récursive qui retourne le nombre de chiffres dans l'écriture décimale de l'entier n.

Exercice 2.9.

Écrire une fonction qui retourne le n-ième terme de la suite de Syracuse, définie par $u_0 \in \mathbb{N}$, et pour tout $n \geq 0$, $u_{n+1} = \dfrac{u_n}{2}$ si u_n est pair et $3u_n + 1$ sinon.

Exercice 2.10.

Programmer la diagonale de Cantor suivant le schéma suivant :



Précisément écrire une fonction `cantor(x,y)` qui retourne le numéro du point de coordonnées (x, y).

Exercice 2.11.

Écrire une fonction récursive qui inverse une liste.

Exercice 2.12.

Écrire une fonction récursive qui teste si une chaîne de caractères est un palindrome ou non.

Exercice 2.13.

Écrire une fonction qui calcule le déterminant d'une matrice de manière récursive en utilisant le développement par rapport à la première colonne.

Exercice 2.14.

Soient $k, n \in \mathbb{N}^*$. Une décomposition de n en k termes est une suite $(x_1, \dots, x_k)$ d'entiers supérieurs à 1 tels que $\sum_{i=1}^{k} x_k = n$. Écrire une fonction `nb_decomposition(n,k)` qui retourne le nombre des décomposition de n en somme de k termes.

Exercice 2.15.

Écrire une fonction récursive `partition(n : int, k : int)->list[list[int]]`, qui retourne la liste des partitions de l'entier n en somme de k entiers.

Exercice 2.16.

On considère la fonction d'Ackermann-Péter définie par :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m, n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

(a) Implémenter la fonction de Ackermann.

(b) Calculer $A(m,n)$, pour $0 \leq m \leq 3$ et $0 \leq n \leq 4$.

(c) Essayer de calculer $A(4,1)$. Que vous dit Python ?

(d) Le module `sys` permet d'augmenter le nombre d'appels récursifs autorisés avec la commande :

```
import sys
sys.setrecursionlimit(nb)
```

où `nb` est le nombre d'appels souhaités. Essayer avec $nb = 5000$. Est-ce assez ? Essayer $nb = 10000$...
Comme vous pouvez le voir le nombre d'appels récursifs est énorme pour de petites valeurs de m et de n. Le calcul de $A(3,4)$ nécessite 10 306 appels à la fonct A...
Cette fonction est utilisée pour tester l'implémentation d'un langage de programmation. Elle croit extrêmement rapidement $A(4,2)$ est un nombre de 19 729 chiffres, ce qui est plus que le nombre d'atomes visibles dans l'univers...

Exercice 2.17.

Récursivité terminale et Tail Call Elimination
La Tail Call Elimination (TCE), également connue sous le nom d'optimisation de la récursivité terminale, est une technique d'optimisation couramment utilisée pour les langages de programmation fonctionnels tels que Scheme, mais elle peut également être appliquée à Python.
La TCE consiste à remplacer une récursivité terminale par une boucle, de sorte que la pile d'appels ne soit plus nécessaire pour exécuter la fonction, ce qui peut améliorer les performances et éviter un éventuel débordement de la pile d'appels.
Pour le faire il suffit de remarquer qu'un appel récursif terminal revient à effectuer un **GOTO** avec une mise à jour des différents paramètres. Prenons l'exemple de l'algorithme d'Euclide du calcul du pgcd : (on suppose :math :'a>b') :

```
def pgcd_rec(a : int, b : int)->int:
    if b == 0:
        pgcd = a
    else:
        pgcd = pgcd_rec(b, a % b)
    return pgcd
```

L'appel récursif est bien terminal. Si l'on se sert de la condition d'arrêt comme condition d'une boucle `while` et que l'on mets les différentes variables à jour on obtient l'algorithme itératif suivant :

```
def pgcd(a : int,b : int)->int:
    while b > 0:
        a, b = b, a % b
    return b
```

(a) Éffectuer une TCE sur la fonction suivante :

```
def s(a : int, b : int)->int
    if a == 0:
        return b
    else:
        return s(a-1,b+1)
```

(b) Éffectuer une TCE sur la fonction suivante :

```
def palindrome(mot : str)->bool:
    if len(mot) < 2:
        return True
    elif mot[0] != mot[-1]:
        return False
    else:
        return palindrome(mot[1:-1])
```

Exercice 2.18.

La TCE n'est possible que si la récursion est terminale. Dans certains cas il est possible de transformer une récursion quelconque en une récursion terminale. Pour cela on a recourt à des accumulateurs pour stocker les résultats intermédiaires et les passer comme argument lors de chaque appel récursif. Par exemple la fonction récursive suivante n'est pas terminale :

```
def somme(n : int)->int:
    if n == 0:
        return 0
    else:
        return n + somme(n-1)
```

Pour transformer cette fonction en une fonction récursive terminale, on peut utiliser un accumulateur pour stocker la somme partielle et le passer comme argument lors de chaque appel récursif :

```
def somme(n : int, acc = 0)->int:
    if n == 0:
        return acc
    else:
        return somme(n-1, acc+n)
```

Dans cette fonction, `acc` est l'accumulateur qui stocke la somme partielle. L'appel initial de la fonction utilise une valeur par défaut de 0 pour l'accumulateur. Lors de chaque appel récursif, la fonction passe `acc+n` comme nouvel accumulateur.

(a) Utiliser un, ou des, accumulateur(s) pour transformer les fonctions récursives suivantes :

```
def puissance(base : int, exposant : int):
    if exposant == 0:
        return 1
    else:
        return base * puissance(base, exposant - 1)

def somme_carres(n : int)->int:
    if n == 0:
        return 0
    else:
        return n**2 + somme_carres(n-1)

def fibonacci(n : int)->int:
    if n in [0,1]:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

def concat_strings(lst : list[str])->str:
    if len(lst)==0:
        return ''
    else:
        return lst[0] + concat_strings(lst[1:])
```

(b) Écrire des versions itératives de toutes les dernières fonctions.