# Chapitre 2 : Récursivité.

MP Lycée Camille Jullian

janvier 2022

Ce deuxième chapitre sera consacré à la programmation récursive. Comme le dit si bien une blague classique d'informaticiens : pour comprendre la récursivité, c'est très simple, il faut d'abord comprendre la récursivité. De fait, le principe d'un programme récursif est de faire appel à lui-même en cours d'exécution. Bien sûr, pour que cela fonctionne, il faut que l'appel « intérieur » se fasse avec des paramètres différents de l'appel « extérieur » et surtout que le programme finisse par être capable d'effectuer un calcul sans boucler indéfiniment sur des appels récursifs. En pratique, un programme récursif fonctionnera comme une récurrence mathématique (d'où le nom) : il faudra traiter à part un cas particulier servant d'initialisation, et s'arrenger pour que les appels récursifs finissent toujours par revenir à ce cas particulier. Les programmes récursifs ont souvent l'avantage de proposer une syntaxe plus élégante que les programmes « classiques » à base de boucles, mais il ne faut pas en abuser non plus car, comme on va le constater, leurs inconvénients sont réels.

## 1 Exemples et principe de fonctionnement.

Dans un premier temps, nous allons traiter quelques problèmes algorithmiques très classiques de deux façons : à l'aide d'un programme itératif, et à l'aide d'un programme récursif. Nous comparerons ensuite les deux méthodes pour déterminer si la version récursive présente des avantages, ce qui nous permettra aussi de nous familiariser avec les principes de fonctionnement de ce type de programmation.

### 1.1 Calcul de factorielles.

Pour calculer une factorielle, on dispose mathématiquement de deux possibilités : le calcul direct $n! = \prod_{k=1}^{n} k$, et la formule de récurrence $(n+1)! = (n+1) \times n!$, qui combinée avec la connaissance de $0! = 1$, permet de reconstituer pas à pas les valeurs des factorielles ultérieures. En Python, c'est exactement pareil. La formule directe va correspondre à un programme itératif classique :

```
def factorielle(n) :
    p=1
    for i in range(2,n+1) :
        p=p*i
    return p
```

Au contraire, la version « relation de récurrence » va donner lieu à un programme récursif :

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

Dans les deux cas, on a supposé l'utilisateur assez intelligent pour donner une valeur entière positive au paramètre $n$ (sinon, le premier programme refusera d'effectuer les calculs de la boucle si $n$ n'est pas

entier, et le secons bouclera indéfiniment ; en l'occurrence, Python est assez malin pour déclencher une erreur si on dépasse un certain nombre d'appels récursifs imbriqués). Expliquons rapidement le principe du programme récursif : si on demande à exécuter l'appel **recfatorielle(4)**, notre cher Python va se rendre compte que $4 \neq 0$, et donc stocker dans un coin la valeur 4 puis calculer la valeur de **recfactorielle(3)** pour pouvoir effectuer le produit de la dernière ligne du programme. Ce calcul de recfactorielle(3) va lui-même nécessiter de connaitre recfactorielle(2), etc, jusqu'à retomber sur l'appel à recfactorielle(0) qui pourra enfin s'effectuer directement (lignes 2 et 3 du programme). Pyhton n'a plus ensuite qu'à « remonter » tous les produits pour afficher la valeur souhaitée. L'un des deux programmes est-il plus efficace que l'autre ? Pas vraiment, puisqu'ils vont en fait exécuter exactement les mêmes calculs. Mais le premier programme itératif est moins gourmand en ressources : il se contente de stocker les valeurs intermédiaires dans la variable $p$ de type int, là où le programme récursif va imposer de conserve « quelque part » (cf ci-dessous) les valeurs 4, 3, 2 et 1 obtenues en cours de route avant de pouvoir faire le calcul final. Bref, le programme récursif est en fait moins bon (en pratique, ils sont équivalents pour des valeurs de $n$ pas trop grandes).

## 1.2 La pile de récursivité.

Revenons rapidement sur ce que nous venons de dire : un programme récursif doit stocker des valeurs intermédiaires le temps d'exécuter les appels récursifs « internes ». En pratique, Python va utiliser pour cela une structure de Pile (mais oui, comme celles qu'on a étudiées au premier chapitre), ce qui est cohérent avec le fait que le dernier appel croisé sera le premier à être exécuté quand on va remonter les calculs. Par exemple, dans le cas de l'exécution de **recfactorielle(4)**, Python va stocker à chaque étape la valeur de $n$ ainsi que l'opération à effectuer lorsqu'on aura calculé la valeur de l'appel récursif, et également l'endroit du programme où il faudra reprendre l'exécution si besoin. Ici, quand Python arrive à la ligne 4 et qu'il se rend compte qu'il doit faire un appel récursif, il va créer une **Pile de récursivité** et y stocker quelque chose du genre « $n = 4$, recfactorielle(4)=4*recfactorielle(3), ligne 4 ». Il lance ensuite l'appel de recfactorielle(3), ce qui va le conduire à empiler sur la même Pile une nouvelle ligne du type « $n = 3$, recfactorielle(3)=3*recfactorielle(2), ligne 4 » et ainsi de suite jusqu'à réussir à calculer recfactorielle(0)=1. Il est alors temps pour lui de dépiler la pile de récursivité, en commençant bien sûr par la ligne permettant de calculer recfactorielle(1)=$1 \times 1 = 1$, puis en effectuant dans l'ordre logique les calculs suivants. Dans le cas du calcul de factorielle, on va dépiler toute la pile de récursivité d'un seul coup, mais ce ne sera pas toujours le cas (si on a plusieurs appels récursifs sur une même ligne par exemple).

Une précision importante : dans la mesure où ce stockage au sein de la pile de récursivité est bien évidemment coûteux en mémoire, Python impose par défaut une limite à la taille de cette pile, qui est en gros de 1 000 éléments. Autrement dit, une fonction récursive en Python qui nécessiterait plus de 1 000 appels récursifs imbriqués va systématiquement provoquer une erreur. Le programme écrit ci-dessus ne permettrait donc pas de calculer des factorielles de gros nombres entiers (contrairement à la version itérative). On peut modifier la valeur de ce paramètre, mais c'est plutôt déconseillé en pratique.

## 1.3 Quelques exemples élémentaires supplémentaires.

Pour voir si les inconvénients de la récursivité constatés sur notre premier exemple se généralisent, testons d'autres algorithmes classiques qui peuvent être facilement écrits de deux façons. Commençons avec l'algorithme d'Euclide de calcul du pgcd :

```
def euclide(n,p) :
    while p>0 :
        n,p=p,n%p
    return n
```

Algorithme classique reposant sur le fait que la division euclidienne conserve le pgcd. Cet algorithme

terminera car les valeurs successives de $p$ vont former une suite d'entiers strictement décroissante. À la différence du calcul de factorielles, le nombre d'étapes n'est ici pas connu à l'avance, d'où l'utilisation d'une boucle while. Aucun problème pour en donner une version récursive :

```
def receuclide(n,p) :
    if p==0 :
        return n
    return receuclide(p,n%p)
```

La version récursive est ici extrêmement naturelle et élégante (mais le programme initial aussi, pour être tout à fait honnête), mais ne fait absolument rien gagner, puisqu'encore une fois les calculs effectués seront rigoureusement les mêmes (et même, cette fois-ci, dans le même ordre). Au contraire, on y perd à nouveau en termes d'utilisation de la mémoire. Allez, exemple suivant, le calcul de puissances :

```
def puissance(x,n) :
    p=1
    for i in range(n) :
        p=p*x
    return p
```

Un programme simple et de bon goût qui ressemble énormément au premier donné pour le calcul de factorielle. On peut bien sûr l'écrire de façon récursive :

```
def recpuissance(x,n) :
    if n==0 :
        return 1
    return x*recpuissance(x,n-1)
```

On retrouve les mêmes inconvénients que précédemment : ça ne va pas plus vite (on effectue les mêmes calculs) et on utilise nettement plus de mémoire, au point de ne pas pouvoir utiliser ce programme avec des valeurs de $n$ trop grandes. Toutefois, on peut écrire une version beaucoup plus intelligente de l'exponentiation en minimisant le nombre de produits effectués. Par exemple, pour calculer $x^{16}$, il est en fait inutile de faire 15 produits, on peut calculer $x \times x$, puis $x^2 \times x^2$ etc, ce qui devrait nécessiter seulement 4 produits. Le programme suivant exploite ce principe :

```
def puissancerapide(x,n) :
    if n==0 :
        return 1
    if n%2==0 :
        a=puissancerapide(x,n/2)
        return a*a
    a=puissancerapide(x,(n-1)/2)
    return a*a*x
```

Je laisse les courageux se convaincre qu'on a cette fois-ci optimisé le nombre d'opérations à effectuer, et surtout que cet algorithme a une complexité logarithmique quand les précédentes étaient linéaires. Il permet également de calculer des puissances beaucoup plus grandes (avant d'atteindre un nombre d'appels récursifs atteignant le millier, on aura des valeurs de $n$ astronomiques). Mais bien entendu, ce n'est plus le même algorithme que le précédent, et il est tout à fait possible d'écrire une version itérative de celui-ci également (c'est un peu plus pénible que la version récursive).

### 1.4 Les dangers de la récursivité.

Terminons ce paragraphie introductif avec un exemple très classique illustrant les dangers d'un recours insuffisamment réfléchi à la récursivité : on souhaite écrire un programme calculant les termes de la suite de Fibonacci (définie par $F_0 = 0$, $F_1 = 1$ et $\forall n \in \mathbb{N}$, $F_{n+2} = F_{n+1} + F_n$. Dans la mesure où il s'agit d'une suite récurrente, on est tenté d'écrire un programme récursif « naturel » comme celui qui suit :

```
def recfibo(n) :
    if n==0 :
        return 0
    elif n==1 :
        return 1
    return recfibo(n-1)+recfibo(n-2)
```

En pratique, le programme précédent commence à donner des signes de fatigue pour $n = 30$ et refuse complètement de donner par exemple la valeur de $F_{50}$, qui est pourtant calculée instantanément par n'importe quelle version itérative correctement rédigée. À quoi cela est-il du ? Une cinquantaine d'additions d'entiers, ça devrait de fait être immédiat. Oui, mais analysons ce que va vraiment faire Python pour calculer **recfibo(50)** : il va effectuer deux appels récursifs simultanés (ici, il ne s'agit pas d'appels imbriqués, d'où le fait qu'on ne sature jamais la pile de récursivité bien que le programme ne termine jamais) à **recfibo(49)** et à **recfibo(48)**. Bien sûr, nous êtres humains savons très bien que $F_{48}$ sera calculé « en passant » pour obtenir $F_{49}$, mais Python, lui, n'en a pas du tout conscience ! Les deux appels récursifs sont traités indépendamment, et vont eux-même engendrer quatre nouveaux appels récursifs (dont deux sont exactement les mêmes mais vont être traités indépendamment !), puis huit etc. En fait, si on note $a_n$ le nombre d'appels récursifs effectués par ce programme, la suite $(a_n)$ va vérifier la même relation de récurrence que la suite de Fibonacci elle-même : $a_{n+2} = a_{n+1}+a_n$, avec toutefois $a_0 = a_1 = 0$ (pas besoin de récurrence pour les deux premières valeurs). Un calcul mathématique que je vous épargne montre que $a_n \sim C \times \varphi^n$, où $C$ est une constante positive, et $\varphi = \dfrac{1+\sqrt{5}}{2}$ (le fameux nombre d'or) un réel strictement supérieur à 1. Ceci prouve que notre algorithme a une complexité exponentielle, ce qui explique les gros problèmes qu'on a avec ! On peut en fait créer des versions récursives du calcul de $F_n$ qui auront un coût linéaire, par exemple celle-ci qui renvoie la valeur de $F_n$ et de $F_{n-1}$ :

```
def recfibo2(n) :
    if n==1 :
        return 0,1
    a,b=recfibo2(n-1)
    return b,a+b
```

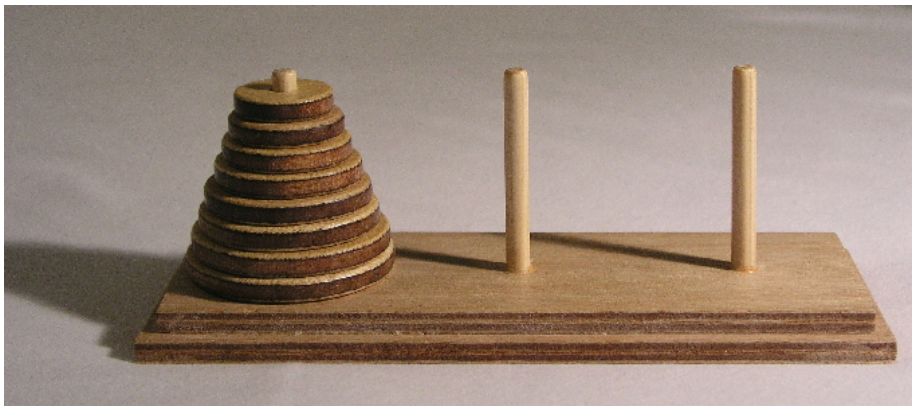
## 2 Utilité de la récursivité pour des problèmes plus complexes.

Jusqu'ici on ne peut pas dire que nos comparaisons aient été en faveur des programmes récursifs. De fait, quand on dispose d'un programme alternatif à base de boucles qui s'écrit simplement, il faudra privilégier ce dernier. Mais il existe des problèmes plus compliqués pour lesquels on n'a en fait pas vraiment le choix, et où l'algorithme récursif s'imposera de lui-même.

### 2.1 Le problème des tours de Hanoï.

Il s'agit d'un jeu de réflexion classique : on dispose de trois piquets sur lesquels on peut empiler $n$ disques troués de rayons strictement croissants. Initialement, les disques sont tous empilés sur le premier piquet (le plus petit en haut, le plus grand en bas). Le but est simplement de les déplacer tous vers le second piquet (ou le troisième, peu importe), en respectant les deux règles suivantes :

- on ne déplace qu'un seul disque à la fois.
- on ne peut jamais poser un disque au-dessus d'un disque plus petit.



Par exemple, pour $n = 3$, on peut résoudre le problème en faisant 7 déplacements : on place le plus petit disque sur le piquet 2, puis le moyen sur le piquet 3 et le petit au-dessus du moyen. Ensuite, on déplace le grand disque sur le piquet 2, puis on remet le petit disque sur le piquet 1, le moyen sur le 2, et enfin le petit sur le deuxième piquet. Il n'est pas très dur de prouver qu'on ne peut pas s'en sortir avec moins de déplacements. Dans le cas général, par contre, il est bien difficile de décrire un algorithme itératif simple, alors qu'en exploitant la récursivité c'est en fait assez simple :

- on déplace les $n-1$ premiers disques (en ne touchant pas au plus grand) du piquet 1 au piquet 3 en exploitant la récursivité.
- on déplace le grand disque du piquet 1 vers le piquet 2.
- on déplace à nouveau les $n-1$ premiers disques du piquet 3 vers le piquet 2 (sans jamais retoucher au grand disque).

Bien sûr l'initialisation peut se faire à $n = 1$ où un déplacement suffit. Comme les piquets utilisés pour les déplacements ne sont pas toujours les mêmes, il est pratique de créer une fonction auxiliaire lors de l'écriture du programme. Ici, la fonction **aux(n,i,j)** décrira les déplacements à effectuer pour déplacer une pile de $n$ disques du piquet $i$ vers le piquet $j$. On donnera ces déplacements sous forme d'une liste de couples (piquet de départ, piquet d'arrivée). Ainsi, la liste correspondant à la résolution du problème pour $n = 3$ données plus haut serait [[1,2],[1,3],[2,3],[1,2],[3,1],[3,2],[1,2]].

```
def tourhanoi(n) :
    def aux(n,i,j) :
        if n==1 :
            return [[i,j]]
        k=6-i-j
        return aux(n-1,i,k)+[[i,j]]+aux(n-1,k,j)
    return aux(n,1,2)
```

Peut-on traduire facilement ce même agorithme par un programme itératif ? Pas facilement (sauf à recréer une structure de Pile qui reviendrait en fait à faire de la récursivité sans le dire). C'est en fait souvent le cas quand les appels récursifs ne sont pas tous imbriqués les uns dans les autres. Ici, on fera un nombre d'appels récursifs de l'ordre de $2^{n-1}$, mais absolument pas imbriqués (on en a deux à chaque sous-étape, avec $n-1$ étapes principales). Notons quand même que la complexité de notre algorithme est donc exponentielle, mais on ne peut tout simplement pas faire mieux pour ce problème (on utilise $2^n - 1$ déplacements pour résoudre le jeu, et c'est optimal). Pour un calcul rigoureux de cette complexité, voir le paragraphe suivant !

## 2.2 Analyse d'algorithmes récursifs.

Maintenant que nous avons enfin sous la main un bon exemple de programme récursif utile, essayons de l'analyser sous les angles habituels : preuve de terminaison et de correction, calcul de complexité. Pour prouver que l'algorithme termine, il suffit comme toujours de trouver un variant (ou un invariant) de boucle intéressant. C'est en général assez évident pour un algorithme récursif : ici c'est bêtement la valeur de $n$ qui va diminuer strictement à chaque appel récursif, et donc finir par être égal à 1, ce qui assurera la terminaison. Dans le cas par exemple du programme d'exponentiation rapide, la décroissance de $n$ est encore plus rapide puisqu'on le divise en gros par 2 à chaque étape (ce qui explique la complexité nettement moindre). Pour prouver la correction de l'algorithme, on procède très naturellement par récurrence : on prouve que le programme renvoie ce qu'il faut pour le cas initial (ici quand $n = 1$ et que la propriété « renvoyer ce qu'il faut » est héréditaire, ce qui est en général assez évident (ici, si la fonction aux effectue le travail demandé au rang $n-1$, elle le fera également au rang $n$ par construction même de l'algorithme récursif). Dans le cas de la fonction d'exponentiation rapide, il faudra effectuer une récurrence forte : si la fonction puissancerapide renvoie la valeur de $x^k$ pour tous les entiers $k < n$, alors ce sera en particulier le cas pour $k = \dfrac{n}{2}$ et $k = \dfrac{n-1}{2}$, donc puissancerapide(n) calculera au choix (selon la parité de $n$), la valeur $x^{\frac{n}{2}} \times x^{\frac{n}{2}} = x^n$, ou la valeur $x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}} \times x = x^n$, ce qui est toujours la valeur attendue.

Plus intéressant, le calcul de complexité passera théoriquement souvent par un calcul de terme général de suite récurrente. Dans les cas simples, on peut s'en sortir à coups d'astuce. Par exemple, pour les tours de Hanoï, notons $C(n)$ la complexité du programme, on a par construction $C(n) = 2C(n-1) + O(1)$ (pour effectuer l'algorithme au rang $n$, on effectue deux fois le même algorithme au rang $n-1$, et on ajoute une étape dont le coût est constant). En divisant tout par $2^n$, on a donc $C\left(\dfrac{n}{2^n}\right) = C\left(\dfrac{n-1}{2^{n-1}}\right) + O\left(\dfrac{1}{2^n}\right)$. En sommant toutes ces relations pour $k$ variant entre 1 et $n$, on obtient alors $C\left(\dfrac{n}{2^n}\right) = O\left(\displaystyle\sum_{k=1}^{n} \frac{1}{2^k}\right) = O(1)$ (vous savez calculer des sommes géométriques, n'est-ce pas ?), ce qui revient à dire que $C(n) = O(2^n)$. On a comme prévu un algorithme exponentiel (mais on ne peut pas faire mieux !).

Dans le cas de l'exponentiation rapide, en supposant $n$ pair pour simplifier, on aurait plutôt quelque chose du genre $C(n) = 2C\left(\dfrac{n}{2}\right) + O(1)$, donc $C(2^k) = 2C(2^{k-1}) + O(1)$, dont on déduit facilement $C(2^k) = O(k)$, ce qui revient à dire que $C(n) = O(\log_2(n))$, la complexité est bien logarithmique. Il existe en fait des théorèmes assez puissants qui évitent de refaire les calculs à chaque fois, le plus connu est celui qui porte le nom particulièrement modeste de « master theorem » (la version donnée ici n'en est en fait qu'un cas particulier) :

**Théorème 1.** Si la complexité $C(n)$ d'un algorithme récursif vérifie une relation de récurrence du type $C(n) = aC\left(\dfrac{n}{2}\right) + O(n^b)$, alors :

- si $b < \log_2(a)$, alors $C(n) = O(n^{\log_2(a)})$.
- si $b = \log_2(a)$, alors $C(n) = O(n^b \log(n))$.
- si $b > \log_2(a)$, alors $C(n) = O(n^b)$.

Ce théorème n'est pas vraiment à connaitre, mais on peut s'amuser à vérifier que les exemples précédents correspondent à certains cas cités ci-dessus. Pour terminer cette section, deux exemples plus ou moins vicieux mais intéressants. Commençons avec un programme classique (mais récursif !) de recherche dichotomique dans une liste triée par ordre croissant :

```
def dichorec(x,L) :
```

```
n=len(L)
if n==0 :
        return False
y=L[n//2]
if y==x :
        return True
elif y<x :
        return dichorec(x,L[n//2+1 :])
else :
        return dichorec(x,L[ :n//2])
```

Un très bel exemple qui devrait logiquement tourner en temps logarithmique, non ? L'algorithme termine effectivement (la longueur des listes dans lesquels on cherche $x$ décroit strictement à chaque appel récursif), et renvoie la valeur correcte, mais il n'est pas du tout logarithmique, à cause du fait idiot que le calcul des « tranches » L[n//2+1 :] et L[ :n//2] se fait en temps linéaire. En gros, on aura ici $C(n) = 2C\left(\frac{n}{2}\right) + O(n)$, ce qui donne une complexité linéaire (pas mieux qu'une recherche débile parcourant la liste). Pour obtenir un vrai coût logarithmique, il faut en fait se débrouiller pour faire la dichotomie en gardant une seule liste de travail (ce qui est tout à fait possible, mais pas très pratique à écrire récursivement).

Allez, on termine avec un classique toujours sympathique. Les suites de Syracuse sont définies de la façon suivante : on part d'un entier $n$, puis on applique récursivement le calcul suivant : si $n$ et pair, on le remplace par $\frac{n}{2}$, sinon on le remplace par $3n+1$, et ce jusqu'à obtenir la valeur 1. Le but est de compter le nombre d'étapes nécessaires avant d'obtenir ce fameux 1. On peut le faire à l'aide du programme suivant :

```
def syracuse(n) :
    if n==1 :
        return 0
    elif n%2==0 :
        return syracuse(n//2)+1
    else :
        return syracuse(3*n+1)+1
```

En pratique, le programme tournera fort bien, mais je suis incapable de vous donner sa complexité. Pire, je ne suis même pas capable de prouver que ce programme termine. En effet, pas de quantité strictement décroissante évidente à exhiber ici, et le fait qu'on va finir par retomber sur 1 n'est pas non plus vraiment clair. En fait, ça l'est tellement peu que c'est un problème ouvert !

### 2.3 L'algorithme de Karatsuba.

Notre dernier exemple vise à améliorer l'algorithme naturel pour effectuer un calcul en apparence simple : le produit de deux polynômes. On considère donc deux polynômes de même degré $P = \sum_{k=0}^{n} a_k X^k$ et $Q = \sum_{k=0}^{n} b_k X^k$, qui seront représentés en Python par la liste de leurs coefficients (en commençant par le coefficient constant $a_0$). L'algorithme naïf consiste à faire quelque chose comme ceci :

```
def prodpoly(P,Q) :
    n=len(P)
    L=[0 for i in range(2*n-1)]
    for k in range(n) :
        for l in range(n) :
```

```
                L[k+l]+=P[k]*Q[l]
    return L
```

Cet algorithme est assez clairement quadratique (on a simplement deux boucles imbriquées de longueur $n+1$, avec un calcul en temps constant à l'intérieur). Il est a priori difficile de faire mieux. Examinons pourtant ce qui se passe pour deux polynômes de degré 1 : $P = a + bX$, $Q = c + dX$, donc $PQ = ac + (ad + bc)X + bdX^2$. Il faut a priori quatre multiplications et une addition pour obtenir les coefficients de ce produit. On peut en fait s'en sortir un peu mieux en calculant seulement trois produits : $ac$, $bd$ et $(a + b) \times (c + d) = ac + bd + bc + ad$. On en déduira en effet le coefficient $ad + bc$ en soutrayant au troisième produit calculé la somme des deux précédents. Au total, on aura effectué trois multiplications et quatre additions (ou soustractions). C'est un peu plus d'opérations que par la méthode classique, mais comme les multiplications sont les opérations qui prennent (de loin) le plus de temps à effectuer, on y gagne. On peut généraliser le principe pour deux polynômes de degré $n$ :

- On pose $m = \lfloor \frac{n}{2} \rfloor$, et on écrit $P = P_0 + X^m P_1$, en séparant les termes de degré inférieur à $m$ des autres. De même, on écrit $Q = Q_0 + X^m Q_1$.
- On calcule alors les trois produits $T_0 = P_0 Q_0$, $T_1 = P_1 Q_1$ et $T_2 = (P_0 + P_1)(Q_0 + Q_1)$ (on les calculera en pratique récursivement pour augmenter encore l'efficacité).
- On constate alors que $PQ = T_0 + X^m(T_2 - T_1 - T_0) + X^{2m} T_1$.

Le programme Python correspondant :

```
def karatsuba(P,Q) :
    n=len(P)
    if n==1 :
        return [P[0]*Q[0]]
    m=n//2
    P0,P1,Q0,Q1=P[ :m],P[m :],Q[ :m],Q[m :]
    T0,T1=karatsuba(P0,Q0),karatsuba(Q0,Q1)
    for i in range(m) :
        P1[i],Q1[i]=P0[i]+P1[i],Q0[i]+Q1[i]
    T2=karatsuba(P1,Q1)
    L=[0 for i in range(2*n-1)]
    for i in range(len(T0)) :
        L[i],L[i+m]=T0[i],-T0[i]
    for i in range(len(T1)) :
        L[i+m]-=T1[i]
        L[i+2*m]+=T1[i]
    for i in range(len(T2)) :
        L[i+m]+=T2[i]
    return L
```

Peut-on estimer la complexité de cet algorithme ? Pour multiplier deux polynômes de degré $n$, on effectue trois appels récursifs sur des polynômes de degré environ égal à $\frac{n}{2}$ (ça dépend de la parité de $n$), et un certain nombre d'opérations dont le coup va être linéaire (les dernières boules notamment), ce qui devrait donner une relation de récurrence du type $C(n) = 3C\left(\frac{n}{2}\right) + O(n)$. Si on applique le master theorem, on a donc $a = 3$ et $b = 1$, donc $b < \log_2(a)$ et la complexité devrait être en $O(n^{\log_2(3)})$. C'est sensiblement mieux que du quadratique ($\log_2(3) \simeq 1.58$) même si en pratique lalgorithme ne devient vraiment performant que pour des polynômes de degré assez élevé.