MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS

Liberté
Égalité
Fraternité

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

DIVISER POUR RÉGNER

L'algorithme de recherche dichotomique, présenté en première, permet la recherche d'un élément dans un tableau trié en temps logarithmique. Il consiste à séparer le tableau en deux à chaque étape, et à chercher l'élément dans l'un des sous-tableaux. Voir la ressource éduscol « Recherche dichotomique ».

L'approche « diviser pour régner » est une généralisation de la dichotomie. Un algorithme de type « diviser pour régner » se décompose en trois phases.

1. *Diviser* : étant donné le problème à résoudre, on découpe l'entrée en deux ou plusieurs morceaux.
2. *Résoudre* : à l'aide d'appels récursifs, on résout le problème sur chacun des morceaux.
3. *Combiner* : à partir des résultats des appels récursifs, on construit la solution du problème de départ.

Il est facile d'oublier, dans cette description, un aspect absolument fondamental : puisque l'on utilise des algorithmes récursifs, il ne faut pas oublier de traiter les cas de base.

Cette fiche présente trois exemples d'algorithmes « diviser pour régner » : le premier permet de tourner une image d'un quart de tour sans utiliser d'espace mémoire auxiliaire, le deuxième est l'algorithme du tri fusion qui est optimal en nombre de comparaisons, le troisième est l'algorithme de Karatsuba pour multiplier des entiers plus rapidement qu'avec l'algorithme naïf.

Les fichiers joints comprennent tous les codes présentés, ainsi que les figures (ou les codes permettant de les générer), pour faciliter leur réutilisation. Cliquer ici pour les télécharger.

La rotation d'image

L'objectif de cet algorithme est d'effectuer la rotation d'une image de 90° dans le sens horaire. On se restreint ici à des images de format carré, et dont les dimensions sont une puissance de 2. On illustre l'algorithme à l'aide de la bibliothèque `PIL`[1].

Algorithme de base

Le principe de l'algorithme est illustré par la figure suivante. Il suit les trois phases de l'approche « diviser pour régner ».

1. *Diviser* : l'image est découpée en quatre zones, nommées A, B, C et D.
2. *Résoudre* : chaque zone est tournée de 90° par un appel récursif.
3. *Combiner* : on place les zones tournées à leur emplacement final.

Pour effectuer la rotation d'une image de dimension n, on découpe l'image en quatre blocs de tailles identiques, on effectue récursivement la rotation de chaque bloc, puis on permute les quatre blocs.

On peut implanter directement cet algorithme, à l'aide de la bibliothèque `PIL`. L'instruction `image.crop(zone)`, où `zone` est un quadruplet `(g,h,d,b)`, renvoie une copie de la zone de l'image constituée des pixels `(x,y)` pour `x` allant de `g` (« gauche ») à `d` (« droit »), et `y` de `h` (« haut ») à `b` (« bas »). Elle nous sert à *découper* l'image. L'instruction `image.paste(coord, im)` colle l'image `im` dans `image`, en mettant son coin en haut à gauche aux coordonnées `coord = (x,y)`. Elle nous sert à *combiner* les blocs.

```
def rotation(image):
    m, n = image.size       # dimensions de l'image
    assert m == n, «l'image doit être carrée»
    assert n & (n-1) == 0, «la dimension doit être une puissance de 2»

    # Pas de cas de base : si n = 1 on ne fait rien
    if n > 1:
        k = n // 2
    # Diviser : découpe des blocs
        A = image.crop((0, 0, k, k))
        B = image.crop((k, 0, n, k))
        C = image.crop((k, k, n, n))
        D = image.crop((0, k, k, n))
    # Résoudre : rotation des blocs
        rotation(A)
        rotation(B)
        rotation(C)
        rotation(D)
    # Combiner : permutation des blocs
        image.paste((0,0), D)
        image.paste((k,0), A)
        image.paste((k,k), B)
        image.paste((0,k), C)
```

1. Nous renvoyons aux ressources SNT « Apprendre à manipuler une image numérique » et « Traitement d'images » pour une présentation de cette bibliothèque.

Version « en place »

L'implantation précédente copie chacune des quatre zones dans un espace temporaire, effectue leur rotation, puis colle les zones tournées à leur nouvel emplacement. Elle nécessite donc de l'espace auxiliaire, de taille supérieure à la taille de l'image elle-même puisqu'il faut au total $\frac{4}{3}n^2$ pixels auxiliaires[2].

On peut diminuer l'espace en remarquant qu'il suffit de ne stocker que deux zones au lieu de quatre, en réordonnant les calculs. En effet, si on a stocké une copie de la zone A en mémoire, on peut effectuer la rotation de la zone D et la coller directement à la place de la zone A. Cela permet ensuite de traiter de même la zone C puis la zone B , et enfin de traiter la zone A. On peut réutiliser le même espace mémoire pour traiter les trois zones B, C et D. On obtient un espace temporaire de taille $\frac{2}{3}n^2$.

On peut en réalité se passer totalement d'espace auxiliaire, c'est-à-dire se ramener à un espace auxiliaire de taille constante. Pour cela, on utilise deux idées. Premièrement, au lieu de copier les zones pour effectuer leur rotation, on effectue chacune des quatre rotations dans l'image elle-même. Deuxièmement, on effectue la rotation des blocs pixel par pixel, ce qui ne nécessite ainsi qu'un nombre constant de pixels à retenir. Toutes les fonctions doivent maintenant prendre en paramètre la `zone` de travail, qui est un quadruplet comme dans la méthode `crop` de `PIL`.

On commence par cette permutation de blocs, pixel par pixel. On remarque la similarité de cette méthode avec la partie de `rotation` qui effectue la permutation des blocs.

```
def permutation_blocs(image, zone):
    g,h,d,b = zone
    k = (d-g)//2
    x, y = g+k, h+k
    for i in range(k):
        for j in range(k):
            pixelA = image.getpixel((g+i,h+j))
            pixelB = image.getpixel((x+i,h+j))
            pixelC = image.getpixel((x+i,y+j))
            pixelD = image.getpixel((g+i,y+j))

            image.putpixel((g+i,h+j), pixelD)
            image.putpixel((x+i,h+j), pixelA)
            image.putpixel((x+i,y+j), pixelB)
            image.putpixel((g+i,y+j), pixelC)
```

Étant donné cette fonction de permutation des blocs, on peut écrire la nouvelle version de `rotation`, qui n'utilise qu'un espace auxiliaire constant. On implante ici une légère variante de l'algorithme : on effectue d'abord la permutation des blocs, avant de leur appliquer la permutation. Les deux variantes sont bien entendu parfaitement équivalentes.

2. L'espace mémoire utilisé est constitué de quatre images de dimensions $n/2$ pour chaque zone, plus l'espace nécessaire pour effectuer récursivement la rotation de chaque image. En supposant que les quatre rotations réutilisent le même espace mémoire, l'espace total vérifie la récurrence $S(n) = 4$. La résolution de cette récurrence fournit la borne $S(n) = O(n^2)$. Un calcul plus précis montre que $S(n) = \frac{4}{3}n^2$.

Rotation d'image en commençant par permuter les blocs avant de leur appliquer la rotation.

```
def rotation_en_place(image, zone = None):
    if zone is None:
        zone = (0, 0, image.width, image.height)
    g,h,d,b = zone

    permutation_blocs(image, zone)

    if d-g > 2:
        x = (g+d)//2
        y = (h+b)//2
        rotation_en_place(image, (g,h,x,y))
        rotation_en_place(image, (x,h,d,y))
        rotation_en_place(image, (g,y,x,b))
        rotation_en_place(image, (x,y,d,b))
```

Compléments et activités

Activité

L'intérêt de cet algorithme est son aspect très visuel. Il peut être présenté à deux niveaux : soit la version avec espace auxiliaire, soit la version en place. Dans le premier cas, il n'est pas question d'efficacité de l'algorithme, mais plutôt de relative simplicité : effectuer la rotation d'un quart de tour sans avoir à calculer la nouvelle position du pixel de coordonnées (i, j). Faire programmer cette version permet aussi de mettre en œuvre la compétence *Utiliser des API* du chapitre *Langages et Programmation*.
La version en place est bien plus délicate et peut faire l'objet d'un projet de programmation ambitieux.

En pratique

Hormis son efficacité en mémoire, qui peut être atteinte par d'autres méthodes, cet algorithme de rotation a deux avantages. Il est naturellement adapté aux représentations récursives d'images (*quadtree*) utilisées par exemple en compression. De plus, les copies de zones rectangulaires d'images (*blits*) sont des opérations très optimisées sur les cartes graphiques, rendant l'algorithme très efficace en pratique.

Au-delà

L'algorithme présenté fonctionne avec un espace auxiliaire constant pour des images carrées. L'existence d'un algorithme en espace constant et temps optimal pour la rotation d'images rectangulaires est un problème ouvert, qui met en jeu des questions fines de théorie des nombres.

Le tri fusion

En première ont été vus les tris par insertion et sélection, qui permettent de trier un tableau de nombres en temps quadratique. La méthode « diviser pour régner » permet de décrire un algorithme de tri de meilleure complexité, à savoir $O(n\log n)$ pour un tableau de taille n. Le principe de l'algorithme suit les phases de la méthode « diviser pour régner ».

1. *Diviser* : découpe du tableau initial en deux sous-tableaux de taille (environ) égale.
2. *Résoudre* : tri des deux sous-tableaux, grâce à deux appels récursifs.
3. *Combiner* : *fusion* des deux sous-tableaux triés pour produire le tableau trié complet.

La figure suivante illustre le tri fusion, sur un tableau de taille 8.

L'étape délicate est la fusion des tableaux triés. Pour la découpe, on coupe simplement le tableau par son milieu. S'il possède n éléments, les deux sous-tableaux sont `T1 = T[0 : n//2]` et `T2 = [n//2 : n]`.

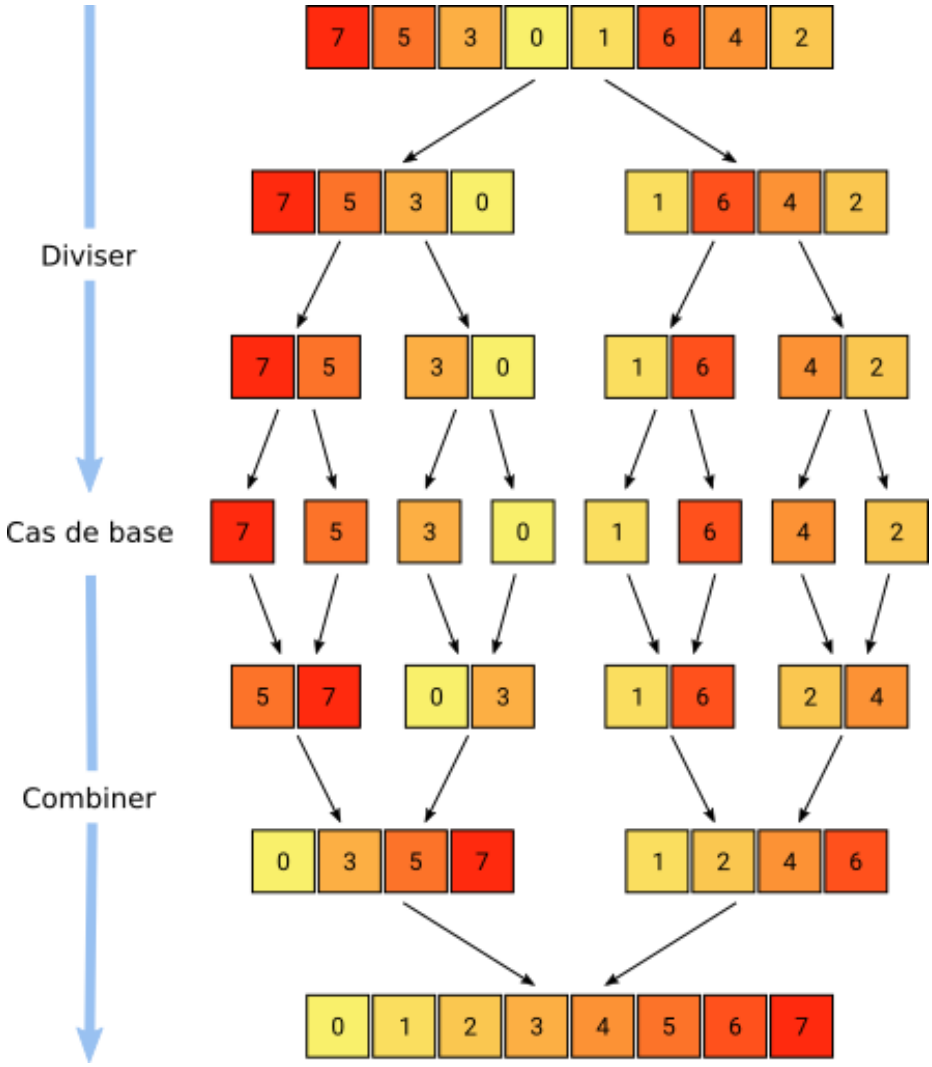


Illustration du tri fusion sur un tableau de taille 8. Les 3 premières étapes divisent le tableau initial et les trois dernières effectuent les fusions.

Fusion de tableaux triés

Le problème, qui constitue la brique de base du tri fusion, est le suivant : étant donnés deux tableaux T_1 et T_2, triés, il s'agit de construire le tableau T, lui aussi trié, qui contient l'ensemble des éléments présents dans T_1 et T_2. Cette brique de base n'utilise pas elle-même la méthode « diviser pour régner ».

Pour visualiser l'algorithme, le plus simple est de voir les tableaux T_1, T_2 et T comme des *piles*. On suppose que l'élément en haut de pile de T_1 et T_2 est leur plus petit élément. Pendant l'algorithme, on ne peut donc accéder qu'à deux éléments : le plus petit de chaque pile. L'algorithme consiste alors à itérativement dépiler un élément soit de T_1 soit de T_2 (selon lequel est le plus petit), pour l'empiler sur la pile de sortie T. Si une des deux piles est vide, on dépile entièrement l'autre pile.

L'implantation de l'algorithme peut se faire directement avec des tableaux, sans implanter une véritable structure de pile. On utilise pour chacun des deux tableaux un indice qui indique le haut de pile : dépiler consiste simplement à incrémenter l'indice, et empiler à mettre à jour une valeur de T et incrémenter l'indice. L'implantation la plus simple consiste à faire une première boucle qui s'occupe du cas où les deux piles sont encore non vides. On traite la fin (une des deux piles est vide) dans un second temps.

```
def fusion(T1, T2):
    «Fusionne les deux tableaux triés T1 et T2»
    T = [0] * (len(T1) + len(T2))
    i1 = i2 = j = 0 # indices des piles

    # Tant que les deux piles sont non vides
    while i1 < len(T1) and i2 < len(T2):
        if T1[i1] < T2[i2]:
            T[j] = T1[i1]
            i1 += 1
        else:
            T[j] = T2[i2]
            i2 += 1
        j += 1

    # Une des deux piles est vide :
    # seule une des 2 boucles while est exécutée !
    while i1 < len(T1):
        T[j] = T1[i1]
        i1 += 1
        j += 1
    while i2 < len(T2):
        T[j] = T2[i2]
        i2 += 1
        j += 1

    return T
```

Complexité

Cet algorithme effectue un nombre de comparaisons linéaire en la taille du tableau fusionné T. En effet, à chaque étape de la première boucle, une comparaison est effectuée et l'indice j est incrémenté de 1.

Correction

On utilise l'*invariant de boucle* suivant : à chaque entrée dans la première boucle `while`, les j premiers éléments de T sont triés et $T[j-1]$ est inférieur ou égal à $T_1[i_1]$ et $T_2[i_2]$. Cet invariant est vérifié avant la première entrée puisque $T[j-1]$ n'existe pas. Si l'invariant est correct à l'entrée d'une itération, puisque $T[j-1]$ est inférieur ou égal à $T_1[i_1]$ et $T_2[i_2]$, les $j+1$ premiers éléments de T sont toujours triés après l'itération. Supposons que $T_1[i_1] < T_2[i_2]$. Alors $T_1[i_1]$ est le $(j+1)^{\text{ème}}$ élément de T : il est bien inférieur à $T_1[i_1+1]$ et à $T_2[i_2]$, donc l'invariant reste valide après l'itération. L'autre cas est symétrique. Avant de rentrer dans l'une des deux dernières boucles, les premiers éléments de T sont triés et inférieurs aux éléments non encore insérés de la pile non vide. La dernière boucle exécutée conserve bien le fait que T est trié. Ainsi, l'algorithme est correct.

L'algorithme du tri fusion

Étant donné l'algorithme de fusion, l'algorithme du tri fusion ne fait plus de difficulté. Il faut simplement ne pas oublier les cas de base.

```
def tri_fusion(T):
    «Renvoie le tableau T trié»
    n = len(T)
    # Cas de base
    if n < 2:
        return T[:] # copie de T
    # Diviser : découpe de T
    T1 = T[0 : n//2]
    T2 = T[n//2 : n]
    # Résoudre : appels récursifs
    T1 = tri_fusion(T1)
    T2 = tri_fusion(T2)
    # Combiner : fusion des tableaux triés
    return fusion(T1, T2)
```

La preuve de correction de l'algorithme est une preuve par récurrence sur la taille de T. Les cas de base (taille nulle ou 1) sont bien traités. Soit $n > 1$ et supposons par hypothèse de récurrence que l'algorithme trie correctement les tableaux de taille n. Puisque T_1 est de taille $\lfloor \frac{n}{2} \rfloor < n$ et T_2 de taille $n - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil < n$, les deux tableaux T_1 et T_2 sont correctement triés par les appels récursifs. Puisque l'algorithme `fusion` est correct, le tableau renvoyé est bien trié.

Pour la complexité de l'algorithme, on se réfère à la figure d'illustration. On remarque d'abord que les divisions de tableaux en sous-tableaux sont « gratuites », au moins en termes de comparaisons. Il s'agit donc de borner le nombre de comparaisons effectuées lors des fusions. On a dit dans la partie précédente que la fusion de deux tableaux dont la somme des tailles vaut m nécessite $O(m)$ comparaisons. À chaque

étage de fusion, la somme des tailles de tous les tableaux fusionnés est exactement n : le nombre de comparaisons effectuées à chaque étage est donc $O(n)$. Il ne reste plus qu'à compter le nombre d'étages. Partons du bas : au dernier étage, il existe un seul tableau, de taille n ; à l'avant-dernier étage, deux tableaux de taille $n/2$; etc. jusqu'au premier étage de fusion, qui contient n tableaux de taille 1. Le nombre d'étages est donc le nombre de fois qu'il faut diviser n par 2 pour arriver à 1 : ce nombre est donc $O(\log_2 n)$. L'algorithme a donc une complexité $O(n\log_2 n)$.

Compléments et activités

Activité

Cet algorithme se prête très bien à une activité débranchée. On peut par exemple appliquer l'algorithme pour trier un paquet de carte selon un ordre défini. La division du paquet en deux paquets de taille environ égale est évidente. Comme dans l'algorithme, la partie délicate et intéressante est la fusion. Pour cela, on dispose les deux paquets en piles avec les cartes face visible. Le but est de fusionner les deux piles en une troisième pile, qu'on va construire face cachée pour montrer qu'il n'est pas nécessaire de l'inspecter. À chaque étape, on sélectionne la plus petite des deux cartes visibles et on la déplace sur la pile de sortie, face cachée. Si une des piles est vide, on sélectionne simplement la carte de la pile restante.

Autres algorithmes

Il existe de nombreux algorithmes de type « diviser pour régner » opérant sur des tableaux. Par exemple, le « tri rapide » est un autre algorithme de tri, de même complexité que le tri fusion si on utilise de l'aléatoire. Son principe est d'effectuer les comparaisons au moment de la division plutôt qu'au moment des fusions. Plus précisément, on commence par choisir un *pivot*, qui est n'importe quel élément du tableau T. Puis le tableau est divisé en deux sous-tableaux T_b et T_h, de telle sorte que T_b contienne tous les éléments de T inférieurs au pivot, et T_h ceux supérieurs au pivot. Par appel récursif, on trie T_b et T_h puis on les fusionne. Cette dernière étape est simple : il suffit de concaténer les deux tableaux.

Une variante de cet algorithme permet de calculer une médiane, ou plus généralement le $k^{\text{ème}}$ plus petit élément, dans un tableau. La division est identique, basée sur le choix d'un pivot. Mais il suffit ensuite d'effectuer un unique appel récursif, dans le sous-tableau qui contient le $k^{\text{ème}}$ plus petit élément. Pour plus de détails, nous renvoyons vers les *Ressources pour la classe* de la Méthode « diviser pour régner », dans la « Sitographie pour le programme de Terminale ».

La multiplication de Karatsuba

L'algorithme de Karatsuba est un algorithme de multiplication d'entiers. Alors que l'algorithme classique de multiplication (qu'on apprend à *poser* à l'école primaire) nécessite un nombre quadratique d'additions et de multiplications chiffre-à-chiffre, cet algorithme permet d'effectuer le calcul en temps $O(n^{1,58})$ environ. Cet algorithme a été découvert en 1960 par Anatolii Karatsuba, alors étudiant auprès d'Andrei Kolmogorov. L'histoire étonnante de cette découverte est présentée sur Wikipédia.

L'idée de base de l'algorithme de Karatsuba peut être illustrée par la multiplication de deux nombres complexes. Le produit $(a+ib)\cdot(c+id) = ac - bd + i(ad+bc)$ semble nécessiter quatre produits de nombres réels. En fait, on peut se contenter de trois multiplications, en remarquant que $ad + bc = ac + bd - (a-b)(c-d)$. Puisque ac et bd sont calculés de

toute façon pour la partie réelle du résultat, une seule multiplication supplémentaire suffit à obtenir la partie imaginaire, au prix de quelques additions et soustractions supplémentaires.

L'algorithme de Karatsuba est une généralisation de cette remarque, pour multiplier deux entiers écrits par exemple en base 10. Soit A et B deux entiers ayant chacun n chiffres en base 10. *Dans toute la suite, on suppose par simplicité que* n *est une puissance de* 2. On peut écrire A et B sous la forme

$$A = \sum_{i=0}^{n-1} a_i\, 10^i \text{et} B = \sum_{i=0}^{n-1} b_i\, 10^i$$

où les a_i et b_i sont des chiffres compris entre 0 et 9. On peut diviser A et B en deux nombres de $n/2$ chiffres : $A = A_0 + 10^{n/2}A_1$ et $B = B_0 + 10^{n/2}B_1$, où A_0, A_1, B_0, B_1 ont tous $n/2$ chiffres. On suit ensuite la même structure que pour la multiplication de nombres complexes. D'une part, on peut développer le produit sous la forme

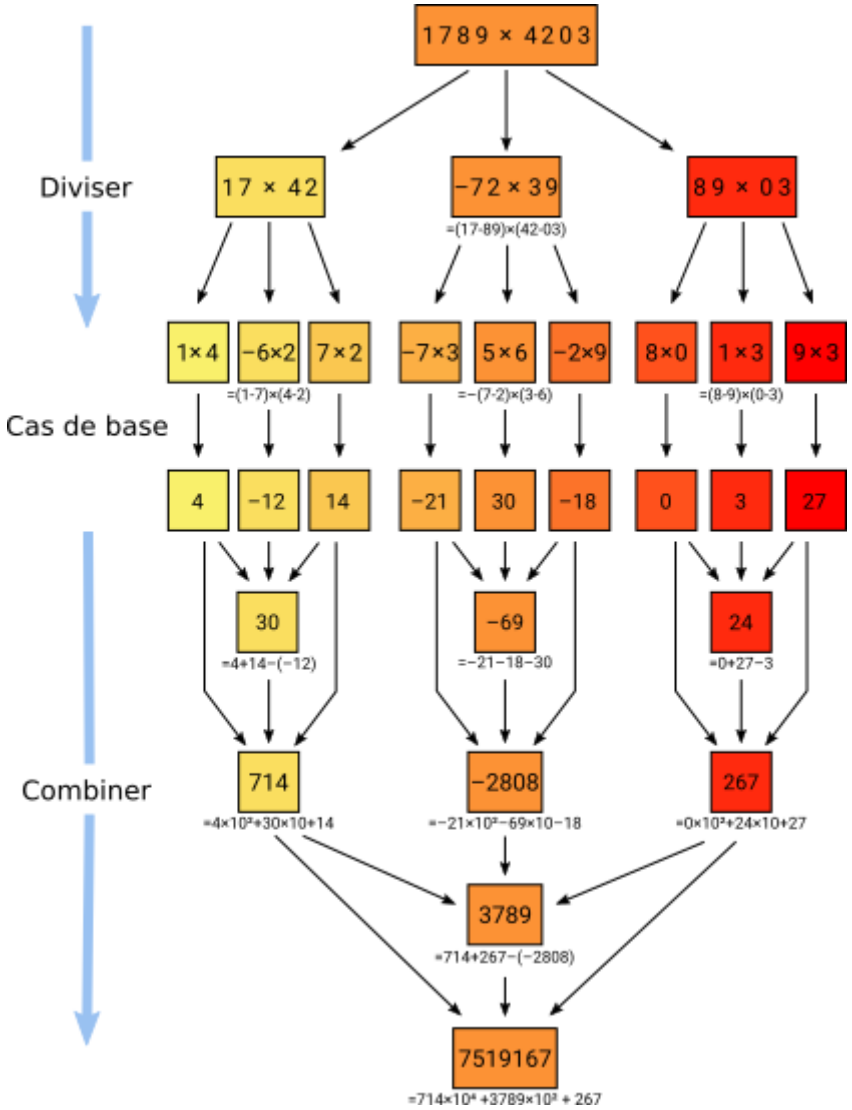
$$A \times B = \left(A_0 + 10^{n/2}A_1\right) \times \left(B_0 + 10^{n/2}B_1\right) = A_0B_0 + 10^{n/2}(A_0B_1 + A_1B_0) + 10^nA_1B_1.$$

Le produit AB *semble* nécessiter quatre produits en taille $n/2$ mais on remarque que

$$A_0B_1 + A_1B_0 \ = A_0B_0 + A_1B_1 - (A_0 - A_1) \times (B_0 - B_1)$$

pour réduire le nombre de multiplications à 3. On obtient ainsi un algorithme de type « diviser pour régner », avec ses trois phases, illustré par la figure suivante.

1. *Diviser* : découpe de A sous la forme $A_0 + 10^{n/2}A_1$ et B sous la forme $B_0 + 10^{n/2}B_1$, et calcul de $A_{0-1} = A_0 - A_1$ et $B_{0-1} = B_0 - B_1$.
2. *Résoudre* : calcul de $C_0 = A_0B_0$, $C_1 = A_1B_1$ et $C_{0-1} = A_{0-1}B_{0-1}$ par appels récursifs.
3. *Combiner* : calcul de $C = C_0 + 10^{n/2}(C_0 + C_1 - C_{0-1}) + 10^nC_1$.



Exemple de multiplication de 1789 *pa* 4203 *par l'algorithme de Karatsuba*

Correction

La correction de l'algorithme est immédiate, en vérifiant les équations précédentes.

Complexité

On exprime la complexité en *nombre d'opérations chiffre à chiffre*, autrement dit en nombre de consultations des tables d'addition, soustraction ou multiplication. Supposons que $n = 2^k$ et notons $T(n)$ le coût de l'algorithme pour des entrées de n chiffres. Alors l'étape de découpe nécessite deux soustractions de nombres de $n/2$ chiffres, ce qui coûte $O(n)$ opérations. La complexité des trois appels récursifs est par définition $3T(n/2)$. Enfin, la combinaison nécessite des additions de nombre de $O(n)$ chiffres, et des multiplications par des puissances de 10 qui ne coûtent rien car ce ne sont que des décalages. On obtient l'équation de récurrence $T(n) = 3T(n/2) + O(n)$. On peut noter $T_k = T(2^k)$, de sorte que l'équation devienne $T_k = 3T_{k-1} + O(n)$. On en déduit que $T_k = O(3^k)$, ce qui est équivalent à $T(n) = O(n^{\log_2 3})$, qui vaut environ $O(n^{1,58})$.

Compléments et activités

Activité

Cet algorithme n'est pas facile à implanter en machine. Les difficultés viennent des opérations d'additions et soustractions avec décalage (multiplication par 10^t), simples en théorie mais délicates en pratique, et de la gestion des signes qui nécessite d'implanter également une comparaison d'entiers. Dans le code fourni, ces opérations sont faites via l'arithmétique de Python. L'utilisation de l'algorithme en version *débranchée* pour effectuer à la main des produits d'entiers est plus adaptée.

Cet exemple permet d'interroger deux notions évoquées dans le programme. Premièrement, la complexité d'un algorithme (coût d'exécution) est un décompte d'opérations élémentaires. La définition de ces opérations est bien souvent laissée implicite car évidente. Il faut expliciter ici que les opérations sur un chiffre sont élémentaires, et que les opérations sur des nombres ne le sont pas. Deuxièmement, le recours aux tables d'additions ou de multiplication illustre le fonctionnement très bas niveau du processeur. Au lieu de fonctionner avec des entiers en base 10, il fonctionne avec des entiers en base 2^{64}. La multiplication de deux entiers de 64 bits est effectuée directement par un circuit électronique qui joue le rôle de table de multiplication. Pour dépasser 64 bits, il faut programmer un algorithme qui utilise en brique de base la multiplication d'entiers de taille ≤ 64 bits. Dans le langage Python, l'algorithme utilisé pour les grands entiers est justement l'algorithme de Karatsuba.

Autres algorithmes

La technique « diviser pour régner » est au fondement de très nombreux algorithmes efficaces pour les opérations mathématiques. Il existe des algorithmes de multiplication d'entiers plus efficace que celui de Karatsuba. Les plus rapides sont fondés sur la *transformée de Fourier rapide*, technique issue du traitement du signal de type « diviser pour régner ». Cette méthode, poussée dans ses retranchements, a permis de découvrir très récemment[3] un nouvel algorithme qui effectue le produit de deux entiers de n chiffres en temps $O(n\log n)$. Il est remarquable que ce problème, étudié depuis des millénaires, soit encore l'objet d'améliorations algorithmiques.

Analyse d'algorithmes « diviser pour régner »

Les algorithmes de type « diviser pour régner » sont des algorithmes récursifs. Leur complexité est naturellement définie par une équation de récurrence. Cette équation reflète les trois phases, et prend très souvent la forme suivante :

$$T(n) = d(n) + aT(n/b) + c(n)$$

où

- $d(n)$ est le coût de l'étape *diviser* ;
- $aT(n/b)$ est le coût de l'étape *résoudre*, consistant en a appels récursifs sur des objets de taille n/b ;
- $c(n)$ est le coût de l'étape *combiner*.

3. La publication scientifique date de 2021.

Le *théorème maître* donne une solution à cette équation de récurrence, en fonction de a, b, $c(n)$ et $d(n)$. On en présente une version simplifiée, dans laquelle on suppose que $c(n) + d(n) = O(n^e)$ pour une certaine constante e. Alors

$$T(n) = \begin{cases} O(n^e) & \text{si} b^e > a \\ O(n^e \log n) & \text{si} b^e = a \\ O\left(n^{\log_b a}\right) & \text{si} b^e < a. \end{cases}$$

On retrouve les complexités en temps ou espace des trois exemples étudiés :

- pour la rotation, le coût en espace de la version gourmande en mémoire vérifie $S(n) = n^2 + S(n/2)$, d'où $a = 1, b = 2, c(n) + d(n) = O(n^2)$. On est dans le cas $b^e = 2^2 > a = 1$, d'où $S(n) = O(n^e) = O(n^2)$;
- pour le tri fusion, puisqu'on a dans ce cas $a = 2$ appels récursifs en taille $n/b = n/2$, et une combinaison en temps $c(n) = n$ (la découpe est gratuite). On est dans le cas $b^e = a$, donc le temps de calcul est $O(n^e \log n) = O(n \log n)$;
- pour l'algorithme de Karatsuba, il y a $a = 3$ appels récursifs en taille $n/b = n/2$, et $c(n) + d(n) = O(n)$. Ainsi, $b^e < a$ et le temps de calcul est $O\left(n^{\log_2 3}\right)$.