Informatique pour tous
TSI 2ème année

P. Adroguer – E. Besson

Chapitre I

Piles (et files)

« I said I wasn't gonna lose my head
But then pop! goes my heart
(pop! goes my heart) »

M. Lawrence, *Music and Lyrics* (2007)

1. Piles

1.1. Généralités.

Définition 1.1. Une *pile* (en anglais : *stack*) est une structure de données collectant des éléments selon le principe « dernier arrivé, premier sorti ».

Le nom de « pile » provient de l'analogie avec un empilement d'objets physiques. On peut imaginer un paquet de cartes, par exemple. La seule carte immédiatement accessible est celle du dessus du paquet : si on souhaite accéder à une carte placée plus bas, il faut commencer par retirer les cartes du dessus. De même, le seul endroit où il est possible d'ajouter une carte est sur le dessus du paquet.

En conséquence, si on ajoute successivement plusieurs objets à la pile, celui qui sera immédiatement accessible est le dernier objet qu'on aura ajouté. C'est la raison pour laquelle on parle de structure « dernier arrivé, premier sorti » (en anglais : *last in, first out*, ou LIFO).

On définit trois opérations élémentaires sur les piles :

- la création d'une nouvelle pile (vide). Les contraintes d'implémentation imposent en général une capacité à la pile (nombre maximal d'objets pouvant y être stockés). On peut spécifier explicitement cette capacité ou laisser le système s'en charger.
- l'empilement (*push*) ajoute un nouvel objet en haut de la pile ; si la capacité de la pile est atteinte, une erreur (*stack overflow*) est renvoyée.
- le dépilement (*pop*) retire l'objet du haut de la pile et le renvoie ; si la pile était vide, une erreur (*stack underflow*) est renvoyée.

Certaines implémentations proposent des opérations supplémentaires, mais celles-ci peuvent toujours être construites à partir des opérations élémentaires.

Exemple(s). Construire les fonctions suivantes :
- *peek* : renvoie l'objet du haut de la pile, sans le dépiler ;
- vide : renvoie `Vrai` si la pile est vide, `Faux` sinon ;
- taille : renvoie le nombre d'objets présents dans la liste.

1.2. Implémentation en Python. Contrairement à d'autres langages de programmation, il n'existe pas en Python de type « pile ». Une pile en Python est représentée par une liste qui se lit de droite à gauche ; l'élément en dernière position de la liste est donc celui du dessus de la pile.

La syntaxe des opérations élémentaires est la suivante :
- `s = []` crée une liste/pile vide `s` ;
- `s.append(x)` empile sur la pile `s` l'objet `x` ;
- `s.pop()` dépile la liste `s`.

Remarque. On ne spécifie pas la capacité de la pile au moment de sa création, c'est Python qui la gère en fonction de la mémoire disponible.

Le fait que le type « pile » ne soit pas distinct du type « liste » permet de « tricher » en mélangeant les opérations de pile et les opérations de liste. Par exemple, on peut accéder à n'importe quel élément de la pile à partir de sa position sans appeler la fonction `pop` (avec la syntaxe `s[i]`). Mais il convient de ne pas trop pratiquer ce genre d'hybridation : l'utilisation d'une pile plutôt que d'une liste (ou vice-versa) est souvent guidée par un intérêt algorithmique (par exemple, l'ordre des éléments imposé par la structure de pile).

Mentionnons néanmoins que la fonction `pop` peut recevoir un argument : `s.pop(i)` retire le `i`-ème item de la liste `s` (peut-on encore l'appeler « pile » ?) et le renvoie. En particulier, `s.pop()` est strictement équivalent à `s.pop(-1)`.

1.3. Applications. Une application très courante des piles est le maintien d'un historique, par exemple les dernières pages visitées par un navigateur web ou l'historique d'annulation d'un logiciel (de traitement de texte ou d'images, etc.). Chaque fois qu'une nouvelle page est visitée (ou qu'une nouvelle opération est effectuée par l'utilisateur/trice), elle est empilée sur l'historique. À chaque clic sur le bouton « précédent » (ou « annuler »), on dépile l'historique et on restaure l'état ainsi renvoyé.

Un atout important des piles est leur capacité à gérer le retour sur trace (*backtracking*), c'est-à-dire la possibilité, lors de la résolution d'un problème, d'explorer une possibilité et de revenir en arrière quand celle-ci se révèle infructueuse.

On peut penser par exemple à un algorithme de résolution du sudoku. On maintient, pour chaque case vide, la liste des chiffres qu'il est permis d'y écrire. On choisit alors une case vide et on y inscrit un chiffre permis, puis une nouvelle case, etc. Si on arrive

ainsi à remplir la grille, c'est gagné. Si on rencontre un blocage en cours de route (une case où aucun chiffre n'est permis), on annule le dernier chiffre inséré (et on l'enlève de la liste des chiffres permis pour cette case), et on remonte ainsi jusqu'à retrouver une situation sans blocage.

1.4. Cas pratique : le problème des huit reines. On s'intéresse ici au problème de placer huit reines sur un échiquier (carré de taille 8×8) sans qu'aucune des reines ne soit en mesure d'en attaquer une autre. On rappelle qu'aux échecs, une reine peut attaquer toutes les cases situées sur la même ligne, la même colonne ou la même diagonale qu'elle. Comparons quelques algorithmes de résolution du problème.

Recherche exhaustive. La solution par recherche exhaustive (*brute force*) consiste à tester toutes les façons différentes de placer les reines, et éliminer toutes les configurations où deux reines peuvent s'attaquer.

Dans la façon la plus naïve d'attaquer le problème, chaque reine peut être placée sur n'importe quelle case, on a donc $64^8 = 2^{48} \simeq 2{,}8 \times 10^{14}$ possibilités.

Une version plus raffinée de la solution consiste à ne pas considérer les cas où deux reines sont sur la même case : on a donc 64 possibilités pour la première reine, 63 pour la deuxième, etc. ; le nombre de configurations possibles est alors $\frac{64!}{56!} \simeq 1{,}7 \times 10^{14}$ (ce qui n'est pas une amélioration très significative).

On peut largement réduire le nombre de possibilités en considérant que les huit reines doivent nécessairement occuper chacune une colonne différente de l'échiquier : on a alors $8^8 = 2^{24} \simeq 1{,}7 \times 10^7$ configurations différentes à étudier.

Encore une amélioration : les reines doivent chacune occuper une colonne *et* une ligne différentes. Parmi toutes les configurations restantes [1], il faudra alors uniquement s'occuper des attaques diagonales. Il s'agit en fait de chercher une permutation de l'ensemble $[\![1; 8]\!]$, dont il existe $8! \simeq 4{,}0 \times 10^4$ représentants.

Recherche par retour sur trace. Le problème de la recherche exhaustive est qu'on étudie beaucoup de situations redondantes. Par exemple, une configuration où les deux premières reines s'attaquent mutuellement va être ré-examinée pour toutes les positions possibles des 6 autres reines, ce qui peut représenter un nombre considérable de configurations en fait inutiles.

Dans la méthode par retour sur trace, on place les reines une par une. Dès qu'un conflit est rencontré, la dernière reine posée est retirée et posée sur une autre case. Si aucune case ne convient, on retire également la reine précédente, etc. L'intérêt est que les situations telles que décrites ci-dessus ne sont plus examinées qu'une seule fois. On peut

1. qui sont alors toutes des solutions du *problème des huit tours*

montrer que dans ce cas, le nombre total de configurations examinées sera seulement de $1{,}6 \times 10^4$ environ, nombre que l'on peut faire descendre jusqu'à $5{,}5 \times 10^3$ en raffinant davantage la manière de chercher.

Il est crucial ici de maintenir deux piles, l'une contenant les reines déjà placées et l'autre les reines restant à placer. Lorsque le placement d'une reine provoque un conflit, celle-ci est dépilée de la liste des reines placées et empilée sur la liste des reines à placer. Chaque reine doit également garder en mémoire la liste des positions déjà testées.

Réparation itérative. Une méthode radicalement différente est celle de la « réparation itérative » (*iterative repair*). Celle-ci consiste à partir d'une configuration illégale et à la modifier successivement jusqu'à aboutir à une configuration légale. Une façon de procéder est par exemple de repérer la reine dont la présence provoque le maximum de conflits et de la déplacer sur sa colonne vers la case où le nombre de conflits sera minimal, puis de recommencer.

On parle d'algorithme « glouton » (*greedy algorithm*), puisqu'on procède en effectuant l'opération qui semble immédiatement la plus satisfaisante, en espérant que cela nous donnera une solution globale.

Contrairement aux méthodes précédentes, l'algorithme glouton ne garantit pas de trouver une solution et peut se retrouver « coincé » dans une boucle infinie ; il faut mettre en place le moyen de détecter un tel blocage et de recommencer à partir d'une nouvelle configuration.

En revanche, on constate empiriquement que cette méthode est très efficace. En moyenne, le problème de placer 1 000 reines sur un échiquier de taille $1\,000 \times 1\,000$ est résolu par réparation itérative en moins de 50 coups.

2. Files

2.1. Généralités.

Définition 2.1. Une *file* (en anglais : *queue*) est une structure de données collectant des éléments selon le principe « premier arrivé, premier sorti ».

Le nom de « file » provient de l'analogie avec une file d'attente. On peut imaginer le passage de clients à une caisse de supermarché, par exemple. Le prochain client à passer est celui qui est arrivé le premier, et si on n'est pas arrivé le premier, on doit attendre que tous les clients précédents soient passés. De même, le seul endroit où il est possible d'attendre lorsqu'on arrive est à la fin de la file (tant qu'on est quelqu'un de bien élevé...).

En conséquence, si on ajoute successivement plusieurs objets à la file, celui qui sera immédiatement accessible est toujours le premier objet qu'on aura ajouté. C'est la raison pour laquelle on parle de structure « premier arrivé, premier sorti » (en anglais : *first in, first out*, ou FIFO).

On définit trois opérations élémentaires sur les files :

- la création d'une nouvelle file (vide). Les contraintes d'implémentation imposent en général une capacité à la file (nombre maximal d'objets pouvant y être stockés). On peut spécifier explicitement cette capacité ou laisser le système s'en charger.
- l'enfilement (*enqueue*) ajoute un nouvel objet à la fin de la file ; si la capacité de la file est atteinte, une erreur (*queue overflow*) est renvoyée.
- le défilement (*dequeue*) retire l'objet du début de la file et le renvoie ; si la file était vide, une erreur (*queue underflow*) est renvoyée. En plus, tous les objets de la file sont avancés d'un cran.

On remarque d'ores et déjà une différence fondamentale entre les piles et les files : le défilement est une opération qui nécessite beaucoup plus de temps comparativement au dépilement : lors du dépilement, seulement le haut de la pile est à changer, alors que lors d'un défilement il est nécessaire de changer en plus la position de tous les éléments de la file. Une option pour pallier ce problème est souvent de garder en mémoire en plus de la liste deux indices qui correspondent au début et à la fin de la file, et ne modifier que ces deux indices pour ne pas avoir à réécrire la liste à chaque défilement (c'est aussi ce qui est fait pour une pile lorsque la taille de la pile est fixée, il faut garder en mémoire l'emplacement dans la liste du dernier objet empilé).

Certaines implémentations proposent des opérations supplémentaires, mais celles-ci peuvent toujours être construites à partir des opérations élémentaires.

Exemple(s). Construire les fonctions suivantes :

- *peek* : renvoie l'objet du début de la file, sans le défiler ;
- vide : renvoie `Vrai` si la file est vide, `Faux` sinon ;

- taille : renvoie le nombre d'objets présents dans la liste.
- print : imprime tous les éléments de la file

2.2. Implémentation en Python. Comme dit précédemment, Python ne gère pas nativement les piles, elles sont donc représentées par des listes. Il en va exactement de même pour les files, et donc la structure des opérations élémentaires est très similaire :

- `s = []` crée une liste/file vide `s` ;
- `s.append(x)` ajoute l'objet `x` à la fin de la file `s` ;
- `s.pop(0)` défile la liste `s`.

En effet, la seule différence par rapport aux piles est que l'on cherche à accéder uniquement à l'élément le premier stocké, donc celui qui est au début de la liste, donc on appelle la fonction `pop` à l'indice 0 pour les files, et Python se charge tout seul de décaler tous les éléments de la liste pour faire avancer la file.

2.3. Applications. Par leur structure, les files sont naturellement privilégiées lors du stockage de données quand on veut conserver leur ordre d'arrivée. En informatique, c'est le cas des données échangées entre deux serveurs par exemple (protocoles internet), ou encore des commandes d'impression envoyées à un serveur d'impression (on souhaite que les documents soient imprimés dans l'ordre dans lequel ils ont été envoyés).

Elles sont aussi couramment utilisées lors de l'exploration de graphe, pour permettre ce qu'on appelle la recherche en largeur.

3. Graphes

Un graphe est un ensemble de sommets reliés par des arêtes. On peut en avoir une idée en se les représentant comme un ensemble de villes reliés par un réseau (routes, autoroutes, chemins de fer, liaisons aériennes, etc) ou encore lors la fabrication d'arbres généalogiques, quand différentes personnes sont reliées ou non par des liens familiaux. Les questions naturelles qui se posent alors concernent la connexité entre deux points (Est-il possible d'aller d'une ville à l'autre ? Deux personnes sont-elles de la même famille ?) ou, lorsque les deux sommets sont reliés, la recherche du chemin le plus court entre eux.

Une fois que l'on a un graphe, et un sommet particulier comme origine (ville de départ ou une personne), on utilise souvent la notion d'arbre : il faut imaginer ce sommet comme au sommet du graphe, et les points qui lui sont reliés en dessous (comme pour un arbre généalogique en partant d'un ancêtre). En psychologie, représenter des arbres à l'envers est souvent mauvais signe pour le diagnostic, donc à vous de voir la conclusion à tirer de cette représentation habituelle en informatique...

Exemple(s). Déterminer comment représenter un graphe avec Python, en particulier comment stocker les informations, et comment y accéder.

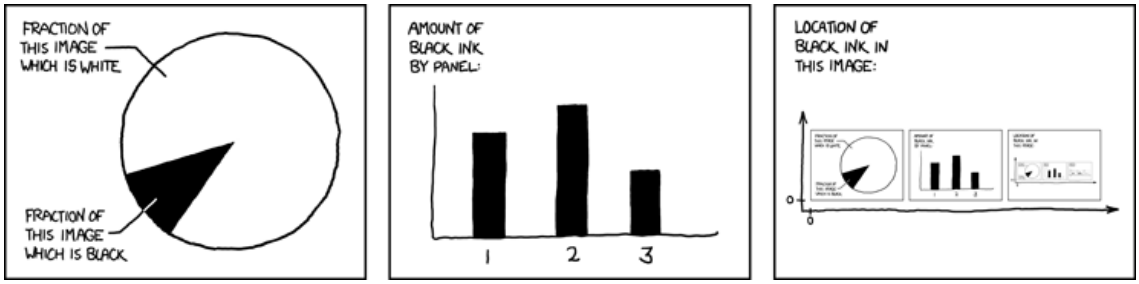
On souhaite maintenant explorer un graphe, pour examiner une composante connexe (quels sont tous les sommets reliés à un sommet donné). Il est logique de commencer par regarder quels sommets sont reliés au premier, et on obtient alors un liste de sommets, qu'il convient de stocker en mémoire avant de refaire cette opération sur tous ces nouveaux sommets. La question est alors de savoir comment stocker ces données, pile ou file ?

Exemple(s). Avec un graphe quelconque, on stocke les données dans une pile ou une file.

- Quelle est la différence pour l'exploration du graphe ?
- Quelle est la meilleure méthode pour trouver toutes les destinations accessibles depuis un aéroport en deux escales ou moins ?
- Si on part de Charlemagne (dont on suppose connu tout l'arbre généalogique), quelle méthode est la plus rapide pour vérifier que vous en êtes un descendant ?

Chapitre II

Récursivité



xkcd.com

1. Première approche

Supposons que l'on souhaite construire un algorithme permettant de calculer $n!$, la factorielle d'un entier naturel n quelconque.

Les méthodes que vous connaissez jusqu'à présent nécessitent l'utilisation d'une boucle *for* ou d'une boucle *while*. Par exemple, avec une boucle *for* on peut imaginer un indice variant de 1 à n et multiplier la variable *resultat* par cet indice à chaque itération, ou bien avec une boucle *while* où une variable d'indice affectée à n au départ est décrémentée à chaque itération de la boucle et la boucle s'arrête quand elle est arrive à zéro.Dans les deux cas, on parle d'algorithmes itératifs.

Ces deux algorithmes correspondent à une définition mathématique de la factorielle comme $n! = 1 \times 2 \times 3 \times \ldots \times n = \Pi_{k=1}^{n} k$.

Toutefois, une autre définition (heureusement équivalente à la précédente) de la factorielle utilise une propriété de récurrence en définissant :

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

On voit que dans ce cas, la définition de factorielle n requiert la définition de factorielle $n-1$, et ainsi de suite, c'est cette propriété qui est à la base des algorithmes récursifs.

2. Implémentation en Python

Depuis les premiers langage de programmation qui ont permis des algorithmes récursifs, maintenant quasiment tous les langages permettent de créer facilement un algorithme récursif. Ainsi en Python un algorithme récursif permettant de calculer factorielle peut s'écrire :

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

Comme on le voit, la définition est extrêmement aisée, il suffit de faire apparaitre dans la définition de la fonction la fonction elle-même quand on en a besoin.

Supposons alors que ce programme soit utilisée pour calculer factorielle 4. Le premier appel de la fonction va renvoyer pour résultat 4 fois le résultat de factorielle 3, qu'il faut donc calculer maintenant. Ce résultat en attente est alors stocké dans une pile de taille suffisante (normalement pour ne pas qu'il y ait d'erreur, mais il est possible de changer cette taille). Cette pile stockera ensuite le résultat de factorielle 2 puis celui de factorielle 1. On sait alors calculer factorielle 1, et en dépilant successivement on retourne factorielle 2 puis factorielle 3 et enfin factorielle 4.

Exemple(s). A votre avis, que va retourner l'appel de `fact_rec` pour un nombre négatif ? Et pour un nombre non-entier ?

3. Terminaison

Comme lors de la démonstration par récurrence d'une propriété en mathématique, il est nécessaire avant toute chose de définir un (ou plusieurs) cas de base (équivalents à l'initialisation de la récurrence. Dans le cas de la factorielle, ceci correspond à donner la valeur de factorielle 1. On peut voir que c'est d'ailleurs nécessaire aussi pour la définition mathématique par récurrence de la factorielle.

Toutefois, comme nous l'avons vu dans l'exemple précédent, ce n'est pas suffisant pour assurer la terminaison du programme, il faut aussi qu'au bout d'un certain nombre d'appels, on finisse par appeler la fonction sur un de ces cas de base. Une manière possible de s'en assurer lorsque l'argument de la fonction est un entier naturel (comme dans le cas de la factorielle déjà décrit) consiste à s'assurer que tous les appels récursifs de la fonction se font avec un argument strictement inférieur à celui de départ. Par exemple, dans le cas du programme décrit précédemment, la calcul de factorielle n nécessite celui

de factorielle $(n-1)$ qui est bien un entier strictement inférieur à n, le programme terminera donc bien une fois arrivé au cas de base de factorielle 0.

Il est courant de devoir démontrer la correction et la terminaison d'un algorithme récursif, c'est-à-dire de montrer qu'il effectue bien la tâche demandée, et que l'algorithme termine bien, sans rester bloqué dans une boucle ou en effectuant un nombre d'appels infinis. La méthode naturelle pour cette démonstration est évidemment une démonstration par récurrence, en vérifiant que le résultat demandé est correct pour les cas de base (initialisation), puis que la véracité du résultat pour tous les arguments strictement inférieurs à n implique la véracité de la propriété à l'argument n.

Exemple(s).
- Ecrire un algorithme récursif permettant de calculer la puissance n-ième d'un entier a.
- Prouver sa correction et sa terminaison.

4. Complexité

Voici 2 exemples d'algorithmes récursifs permettant de calculer la valeur approchée de la racine carré d'un réel a, en utilisant le fait que la suite définie comme $u_0 = a$ et $u_{n+1} = \frac{1}{2}\left(u_n + \frac{a}{u_n}\right)$ est telle que $u_n \to_{n\to\infty} \sqrt{a}$.

```
def rac_1(a,n) :
  if n == 0 :
    return a
  else :
    return (rac_1(a,n-1) + a/rac_1(a,n-1))/2

def rac_2(a,n) :
  if n == 0 :
    return a
  else :
    b = rac_2(a,n-1)
    return (b+ a/b)/2
```

Exemple(s).
- Vérifier leur terminaison et correction.
- Dans le cas où $n = 3$, combien d'appels à la fonction sont effectués dans chaque algorithme ?

On voit dans ce cas que la manière d'écrire un algorithme a une influence considérable sur le temps de calcul : ici, le premier algorithme nécessite un nombre d'appel de la fonction qui est exponentiel en n, alors que dans le deuxième algorithme il est linéaire en n. En admettant en ordre de grandeur 1 milliseconde par opération (ce qui n'est clairement pas le cas pour un processeur actuel), on passe pour le calcul de `racine(a,1000)`

d'un temps de calcul de l'ordre de la seconde pour l'algorithme linéaire à un temps de calcul de l'ordre de 2^{1000} ms, soit environ 10^{300} ms ou encore 10^{290} années (alors que l'âge de l'univers est estimé à 13 milliards d'anneés, donc 10^{10}...).

On classe donc les algorithmes en fonction de leur complexité, exprimée généralement par le terme dominant dans l'expression de la complexité en fonction de la taille n de l'argument.

On vient donc de voir que la complexité du premier algorithme (naïf) est $c_n = 2^{n+1}-1$ donc une complexité exponentielle en n (passer de n à $n+1$ double le temps de calcul).

Le deuxième algorithme est lui de complexité linéaire puisque sa complexité est $c_n = n$, donc on double le temps de calcul quand on passe de n à $2n$.

On trouve aussi la classe des algorithmes de complexité polynomiale (avec une complexité $c_n = n^\alpha$, pour lesquels le but est d'avoir α le plus petit possible : si on passe de n à $2n$, le temps de calcul est multiplié par 2^α) et les algorithmes de complexité logarithmique (avec une complexité $c_n = \mathcal{O}(\log n)$).

Puisque les ordinateurs sont très utiles pour traiter un algorithme avec n grand, il est primordial d'écrire un algorithme en ayant en tête de viser la complexité qui croît le plus lentement possible quand $n \to \infty$.

On peut retenir qu'en terme d'efficacité d'un algorithme :

$$\text{logarithmique} \gg \text{linéaire} \,(\gg \text{polynomial}) \gg \text{exponentiel}$$

5. Application : diviser pour régner

Dans le but de trouver des algorithmes récursifs avec une complexité basse, il existe une stratégie couramment employée appelée *diviser pour régner*.

La structure en est la suivante pour résoudre un problème de taille n :

(1) *diviser* : on découpe le problème initial en deux (ou plus) sous-problèmes de taille inférieure ;
(2) *régner* : on résout les sous-problèmes obtenus, soit directement (cas de base), soit récursivement ;
(3) *combiner* : on résout le problème initial en utilisant les résultats obtenus pour les sous-problèmes.

Exemple(s). L'algorithme de recherche dichotomique est un algorithme permettant de trouver rapidement si un élément appartient à une liste *déjà triée*.

Soit $T = [t_1, t_2, \ldots, t_n]$ la liste d'entiers de départ, rangés par ordre croissant, et a un entier. Un algorithme naïf consisterait à comparer a et chacun des t_i, donc un total de n comparaisons au maximum (complexité linéaire). La stratégie diviser pour régner nous conseille plutôt l'algorithme suivant, appelé *algorithme de recherche dichotomique* (du grec couper en deux) :

(1) On coupe la liste en deux pour avoir les listes $T_1 = [t_1, \ldots, t_{n/2}]$ et $T_2 = [t_{n/2+1}, \ldots, t_n]$.
(2) Si $a = t_{n/2}$ on a fini, sinon, on effectue la recherche sur T_1 si $a < t_{n/2}$ et sur T_2 sinon.

On peut alors vérifier que dans le pire des cas (si a n'appartient pas à la liste), on n'effectue que $\log_2 n$ comparaisons puisque le nombre de comparaisons pour une liste de taille $2n$ est seulement 1 de plus que pour une liste de taille n.

5.1. Algorithme de Karatsuba. L'algorithme de Karatsuba est un algorithme servant à effectuer rapidement la multiplication de deux entiers a et b composés de n chiffres chacun. Dans les années 50, Kolmogorov (mathématicien soviétique ayant fourni des avancées considérables en théorie de l'information, complexité des algorithmes, probabilités, topologie, logique, ...) avait postulé que l'algorithme standard de multiplication était de complexité minimale, et donc en posant le calcul à la main on effectuait un nombre minimal de multiplication, à savoir n^2 (on doit multiplier chaque chiffre d'un nombre avec chaque chiffre de l'autre).

Cet algorithme naïf peut aussi se reprśenter sous la forme d'un algorithme *diviser pour régner* de la manière suivante :

(1) on pose $k = n/2$, et on décompose les deux nombres a et b selon $a = a_1.10^k + a_2$ et $b = b_1.10^k + b_2$;
(2) le produit à calculer s'écrit alors $a \times b = (a_1 \times b_1).10^{2k} + (a_1 \times b_2 + a_2 \times b_1).10^k + (a_2 \times b_2)$;
(3) on obtient donc 4 produits à calculer, avec des nombres de taille $n/2$ chacun, ce que l'on fait récursivement jusqu'à n'avoir à multiplier que deux nombres d'un seul chiffre ;
(4) on recombine les résultats avec la formule précédente (les multiplications par 10^k sont faciles puisque ce ne sont que des décalages de chiffres vers la gauche, ainsi que les additions).

Cet algorithme est bien équivalent à celui qui est suivi lors d'une multiplication comme apprise en primaire, et il y a bien n^2 multiplication de chiffres effectuées ($c_n = 4c_{n/2}$).

D'après Kolmogorov, si on n'avait pas trouvé un algorithme plus efficace c'était vraisemblablement parce qu'il n'en existait pas. Il présenta cette idée lors d'un séminaire auquel assistait Karatsuba, et une semaine plus tard, ce dernier avait trouvé son algorithme plus rapide.

Exemple(s). Algorithme de Karatsuba :

On voit que dans l'algorithme naïf, il faut calculer 4 produits a_1b_1, a_2b_1, a_1b_2 et a_2b_2. Karatsuba a remarqué qu'on pouvait gagner du temps en calculant à la place de $a_1b_2 + a_2b_1$ l'expression $a_1b_1 + a_2b_2 - (a_1 - a_2)(b_1 - b_2)$. Il ne suffit donc que de calculer 3 multiplications puisque a_1b_1 et a_2b_2 sont déjà calculés.

En termes de complexité, on a alors la relation de récurrence $c_n = 3c_{n/2}$, donc $c_n = \mathcal{O}(n^{log_2(3)})$ or $\log_2(3) \simeq 1,585 < 2$. Ceci semble être qu'une amélioration marginale de cet algorithme de complexité polynomiale, mais si on souhaite multiplier deux nombres de 1000 chiffres, le nombre de multiplication passe de un million pour l'algorithme naïf à seulement 50 000.

5.2. Exemple d'application. Le chiffrement RSA est un algorithme de cryptographie (envoi de messages secrets). Pour l'utiliser, on utilise l'asymétrie de complexité entre le calcul du produit de deux nombres premiers (très grands, par exemple de 1024 chiffres en binaire), requis pour l'opération de chiffrement et l'obtention de la décomposition en nombres premiers du chiffre ainsi obtenu si l'on ne connait pas les deux nombres de départ, requise pour le déchiffrement.

Ainsi, l'algorithme de Katsubara que l'on vient de voir nous indique qu'il est possible de calculer le produit en environ 50 000 opérations (certains algorithmes sont encore meilleurs, mais la complexité est dans tous les cas polynomiale). Toutefois, les algorithmes classiques (donc pas en informatique quantique) connus à l'heure actuelle permettant la décomposition en facteurs premiers sont eux de complexité sous-exponentielle (pire que tout algorithme de complexité polynomiale mais meilleur que tout algorithme de complexité exponentielle). Ainsi l'algorithme GNFS pour un nombre de 2048 chiffres nécessite environ 10^{35} opérations...

Toutefois, les algorithmes quantiques de factorisation sont eux de complexité polynomiale (en n^3 pour l'algorithme de Shor), ce qui à terme met en danger la sécurité des algorithmes de chiffrement comme RSA (mais pour l'instant, la difficulté principale est de réaliser un ordinateur quantique).

Table des matières