CONCEPTIONS ALGORITHMIQUES
COURS 1

OLIVIER BODINI

Résumé. Ce cours est une initiation à quelques grands principes algorithmiques (Diviser pour Régner, Programmation Dynamique, Algorithmes Gloutons,...) par leur mise en application dans des situations variées (numérique, graphique, géométrique,...). Cette première séance introductive est consacrée à quelques conventions et rappels (pseudo-code, comportement asymptotique), nous introduirons aussi la notion d'algorithme Diviser pour Régner, avec une première application dans le cas du problème du tri (Tri-fusion, Quicksort), de la multiplication d'entiers (Karatsuba) et de système de pesées. En conclusion, nous donnerons le "théorème maître" qui permet d'obtenir des bornes asymptotiques sur la plupart des fonctions rencontrées lors de l'analyse de coût des algorithmes Diviser pour Régner.

1. Introduction

De tout temps les hommes ont cherché des méthodes pour ≪ systématiser ≫ leurs raisonnements. La division ou l'algorithme d'Euclide pour trouver le PCGD de deux nombres remontent à 400 ans avant notre ère. ≪ Systématiser ≫, c'est assurer la reproductibilité (et donc l'apprentissage) des méthodes, c'est aussi assurer leur transmissibilité dans le temps et dans l'espace sous une forme compréhensible par tous. Afin de transmettre, il faut garantir que les méthodes proposées donnent bien ce à quoi l'on aspire (sinon on parle d'*heuristique*), à la fois dans le but de convaincre l'autre, mais aussi dans le souci constructiviste de garantir la cohérence de l'édifice des savoirs (par la certification de chacune de ses briques). Plus récemment, avec l'avènement de l'informatique, on accorda un soin tout particulier à la description formelle des algorithmes afin d'en faciliter l'exportation vers les différents langages informatiques modernes. C'est le but de l'écriture algorithmique en pseudo-code qui fera l'objet du premier préambule. Le dernier point, et ce n'est pas le moindre quand l'on pense algorithmique, consiste à trouver les méthodes les plus économiques possibles que ce soit en termes de temps, de mémoire ou de tout autre chose. Voilà donc ce que l'on entend ici par algorithmique et c'est ce que nous pouvons résumer ainsi :
l'*algorithmique* (terme qui vient du surnom Al-Khuwarizmi d'un célèbre mathématicien arabe (780-850 ap. JC) à qui l'on doit aussi le mot algèbre) est la science qui consiste à trouver des méthodes formelles et certifiées pour résoudre de manière automatique et le plus efficacement possible des problèmes donnés.

Dans les douze séances de ce cours, nous nous proposons, plus qu'à l'étude ou à la recherche d'algorithme particulier, de mettre en lumière trois "types" universels d'algorithmes que sont : les algorithmes de type **Diviser pour Régner**, les algorithmes de type **Programmation dynamique** et les algorithmes de type **Glouton**. Notre objectif étant de montrer qu'il existe au dessus du monde singulier des algorithmes, de grands principes algorithmiques communs. Pour insister sur le caractère transversal de ces principes, pour chacun d'eux nous proposerons des applications qui couvrent un champ large de l'algorithmique classique, disons pour rester catégoriel dans ses cinq ramifications suivantes : l'algorithmique numérique et matriciel, la géométrie algorithmique, l'algorithmique des graphes, l'ordonnancement et les tris, l'algorithmique des séquences. Ce cours s'architecturera autour de cette idée principale.

1.1. **Convention d'écriture et instructions du pseudo-code.** Nous écrirons dans la mesure du possible les pseudo-codes sous la forme suivante :

Algorithme 1 : Paradigme d'écriture des pseudo-codes

Entrées : deux entiers a et b.
Sorties : un entier.
1 ADDITION(a, b)
2 $s := a + b$
3 **Retourner** s

Les pseudo-codes que nous écrirons auront donc un traits impératifs dans le sens où nous disposerons pour les construire de l'*affectation* que l'on notera :=. Nous utiliserons les instructions suivantes qui seront toujours écrites en caractère gras.

- La boucle avec compteur :
  pour `variable` **allant de** `entier1` **à** `entier2`
  faire `expression`
- La boucle à arrêt conditionnel :
  tant que `condition`
  faire `expression`
- L'instruction conditionnelle :
  si `condition`
  alors `expression`
  sinon `expression`
- Le retour (qui renvoie la sortie de l'algorithme) :
  retourner `expression`
- L'arrêt forcé :
  arrêt `commentaire`

Enfin, si nécessaire, nous utiliserons les primitives classiques de manipulation des structures de données (listes, arbres, piles, files,...).

2. Diviser pour régner

2.1. **De l'usage de l'algorithmique, tri-insertion versus tri-fusion. Première partie : le tri-insertion.** Nous nous intéressons ici au problème du tri d'une séquence de nombres pour montrer en quoi les méthodes employées peuvent jouer un rôle important sur les temps d'exécution. Ce sera aussi l'occasion pour nous d'introduire la notion d'algorithme de type Diviser pour Régner.

Un premier algorithme "naïf" (qui n'est pas de type diviser pour régner) appelé **tri-insertion** peut être décrit récursivement comme suit : L'*entrée* de l'algorithme est une séquence de nombres $L = (a_1, ..., a_n)$. La *sortie* sera une séquence constituée des mêmes nombres mais triés dans l'ordre croissant. On commence avec la séquence de départ $L'_1 = L$, supposons par récurrence que L'_k soit la séquence dont les k premiers éléments ont été triés dans l'ordre croissant. On obtient L'_{k+1} en insérant a_{k+1} à la bonne place parmi les k premiers éléments de L'_k. On itère ce procédé, jusqu'à l'obtention de L'_n qui n'est autre que la séquence triée demandée.

Pour la représentation en pseudo-code de la séquence de nombres, on utilisera un tableau, ce qui permet un accès direct aux données, mais aussi d'effectuer des opérations sur le tableau lui-même sans copie. On dit alors que le TRI-INSERTION est un algorithme de tri *sur place*. Nous obtenons le pseudo-code suivant :

Algorithme 2 : TRI-INSERTION

Entrées : $T[1..l]$ un tableau de nombres représentant la séquence.
Sorties : Rien mais le tableau T est trié.

```
1 TRI-INSERTION(T)
2 Soit l la longueur de T
3 pour j allant de 2 à l faire
4 |   tmp := T[j]
5 |   i := j − 1
6 |   tant que i ≠ 0 et T[i] >tmp faire
7 |   |   T[i + 1] := T[i]
8 |   |   i := i − 1
9 |   T[i + 1] := tmp
```

La seule écriture d'un pseudo-code ne suffit certes pas à valider un algorithme. Nous distinguons deux points fondamentaux à vérifier impérativement :
– s'assurer que l'algorithme se termine.
– vérifier qu'il donne toujours le bon résultat.
De plus, on ne peut concevoir l'écriture d'un algorithme sans essayer de donner une idée de son coût algorithmique.

Analyse de l'algorithme TRI-INSERTION.

Preuve de Terminaison :
La boucle **Tant que** ligne 6 s'achève toujours (elle contient au plus l itérations). De même, la boucle **Pour** j **allant de** 2 **à** l produit $l-1$ itérations. Donc, l'algorithme se termine après un nombre fini d'étapes.

Preuve de validité :
On a un *invariant de boucle* pour la boucle **Pour** j **allant de** 2 **à** l de la ligne 3 qui est « les j premiers éléments sont triés en l'ordre croissant ». Donc à la fin de l'algorithme, c'est-à-dire quand j=l, le tableau est totalement trié.

Analyse de la complexité dans le pire des cas en nombre de comparaisons :
Dans le pire des cas, la boucle **Tant que** fait $j-1$ comparaisons à l'étape j de la boucle de la ligne 3. Ce qui fait en sommant, $1+2+...+(l-1)$ comparaisons au total, soit $l(l-1)/2$ comparaisons. Or, ce cas se produit quand on cherche à trier toute séquence strictement décroissante. **Le tri-insertion est donc un algorithme quadratique dans le pire des cas en nombre de comparaisons**.

Nous allons maintenant proposer un autre algorithme de tri appelé **tri-fusion**, qui repose sur le premier principe que l'on désire mettre en exergue dans le cours, le paradigme, Divide ut imperes (Diviser pour Régner) formule latine attribuée à Philippe de Macédoine (359-336) et reprise par Nicolas Machiavel (1469-1527).

2.2. **A propos de Diviser pour Régner.** La stratégie Diviser pour Régner consiste à scinder un problème en sous-problèmes de même nature sur des instances plus petites, à résoudre ces sous-problèmes, puis à combiner les résultats obtenus pour apporter une solution au problème posé. Il s'agit donc d'une démarche essentiellement récursive. Le paradigme « Diviser pour Régner » donne donc lieu à trois étapes à chaque niveau de récursivité :
Diviser : Le problème est scindé en un certain nombre de sous-problèmes ;
Régner : Résoudre les sous-problèmes récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;
Combiner : Réorganiser les solutions des sous-problèmes en une solution complète du problème initial.

2.3. **De l'usage de l'algorithmique, tri-insertion versus tri-fusion. Deuxième partie : le tri-fusion.** Dans le cas du tri-fusion l'algorithme repose sur la décomposition suivante :
DIVISER : On scinde la séquence de longueur l en 2 séquences de taille $\lceil \frac{l}{2} \rceil$ et $\lfloor \frac{l}{2} \rfloor$.
RÉGNER : On résout chacune des deux sous-séquences en utilisant récursivement le tri-fusion si elle n'est pas réduite à un élément et en ne faisant rien sinon.
COMBINER : Fusionner les deux sous-séquences triées en une séquence triée.

Commençons par décrire en pseudo-code l'algorithme FUSION(T, p, q, r) qui fusionne les tableaux triés $T[p..q]$ et $T[q+1..r]$ où $T[a..b]$ désigne le sous-tableaux de T constitué des éléments compris entre a et b inclus.

Algorithme 3 : FUSION

Entrées : $T[1..l]$ un tableau de nombres, p, q et r trois entiers.
Sorties : Rien mais le tableau T est modifié.

```
1  FUSION(T, p, q, r)
2  Soit l la longueur de T
3  n_1 := q − p + 1
4  n_2 := r − q
5  créer tableaux G[1..n_1 + 1] et D[1..n_2 + 1]
6  pour i allant de 1 à n_1 faire
7      G[i] := T[p + i − 1]
8  pour j allant de 1 à n_2 faire
9      D[j] := T[q + j]
10 G[n_1 + 1] := ∞
11 D[n_2 + 1] := ∞
12 i := 1
13 j := 1
14 pour k allant de p à r faire
15     si G[i] < D[j] alors
16         T[k] := G[i]
17         i := i + 1
18     sinon
19         T[k] := D[j]
20         j := j + 1
```

Le pseudo-code du tri-fusion est alors :

Algorithme 4 : TRI-FUSION

Entrées : $T[1..l]$ un tableau de nombres, p et r des entiers.
Sorties : Rien mais le tableau T est trié entre p et r.

```
1 TRI-FUSION(T, p, r)
2 si p < r alors
3     q := ⌊(p+r)/2⌋
4     TRI-FUSION(T, p, q);
5     TRI-FUSION(T, q + 1, r);
6     FUSION(T, p, q, r);
```

Pour obtenir un tableau trié T, il suffit de faire l'appel TRI-FUSION$(T, 1, longueur(L))$. Mais avant de continuer, vérifions la validité de ces deux algorithmes.

Analyse de l'algorithme FUSION :

Preuve de Terminaison :
Nous n'avons dans cet algorithme que des boucles de type compteur. Donc l'algorithme s'arrête après un nombre fini d'étapes.

Preuve de Validité :
Nous allons utiliser pour ce faire, l'invariant de boucle (pour la boucle de la ligne 14) suivant : le tableau $T[p..k-1]$ contient les $k-p$ plus petits éléments de la concaténation $G[1..n_1+1]D[1..n_2+1]$, en ordre croissant et $G[i]$ et $D[j]$ sont les plus petits éléments de leurs tableaux à ne pas avoir été copiés dans T. Vérifions que c'est bien un invariant de boucle. Au premier passage ligne 14, on a $k = p$ et donc $T[p..k-1]$ est vide. Donc on peut dire qu'il contient bien les 0 plus petits éléments de $G[1..n_1+1]D[1..n_2+1]$ en ordre croissant. Comme $i = j = 1$, $G[i]$ et $D[j]$ sont les plus petits éléments de leur tableau respectif à ne pas avoir été copiés.
Par induction : Supposons que l'invariance soit vérifiée pour les k premières itérations de la boucle. Nous divisons le problème en deux cas : soit $G[i] \leq D[j]$, soit $G[i] > D[j]$, nous traitons le premier cas, le deuxième étant identique mutatis mutandis. Comme $G[i]$ est le plus petit élément à ne pas avoir encore été copié, il s'ensuit que $G[i]$ est copié en $T[k]$ et comme $T[p..k-1]$ par hypothèse contient les $k-p$ plus petits éléments, on a que $T[p..k]$ contient les $k-p+1$ plus petits éléments classés en ordre croissant. Finalement i est incrémenté ligne 17 ce qui assure que $D[i]$ est bien maintenant le plus petit éléments de D non copié. On a bien à la nouvelle itération l'invariant conservé.
Validité finale : La boucle s'arrête quand $k = r+1$. Donc d'après l'invariant le tableau $T[p..k-1]$ qui n'est autre que $T[p..r]$ contient en ordre croissant tous les éléments de $G[1..n_1+1]$ et $D[1..n_2+1]$ sauf les deux plus grands (en effet on s'arrête après avoir rangé $r-p+1$ éléments et $G[1..n_1+1]$ et $D[1..n_2+1]$ contiennent à eux deux $r-p+3$ éléments. Or les deux plus grands éléments ne sont autres que les deux *sentinelles* que l'on avait rajoutées ($G[n_1+1] =: \infty$ et $D[n_2+1] =: \infty$).

Analyse de la Complexité en nombre de comparaisons :
On fait exactement une comparaison par itérations de la boucle ligne 14. Soit exactement n_1+n_2 comparaisons pour fusionner deux listes de tailles respectives n_1 et n_2.

Analyse de l'algorithme TRI-FUSION :

Preuve de Terminaison :
Par induction sur la longueur de la zone $[p..r]$ à trier, pour TRI-FUSION(T, p, r) avec $p = r$, ne fait que retourner T donc est fini. Supposons que pour tous r et p tels que $r-p < k$, l'algorithme TRI-FUSION(T, p, r) se termine, alors pour tout p, TRI-FUSION$(T, p, p+k)$ se termine aussi car il appelle recursivement trois fonctions qui par hypothèse d'induction se terminent.

Preuve de Validité :
Par induction sur la longueur de la zone à trier, si $p = r$, c'est trivial. Supposons que pour tous r et p tels que $r-p < k$, l'algorithme TRI-FUSION(T, p, r) soit valide. Etudions, TRI-FUSION(T, p, r) pour $r-p = k$, comme TRI-FUSION$(T, p, \left\lfloor \frac{(p+r)}{2} \right\rfloor)$ et TRI-FUSION$(T, \left\lfloor \frac{(p+r)}{2} \right\rfloor + 1, r)$ sont valides par hypothèse d'induction et que FUSION donne un tableau trié à partir de deux tableaux triés, on a que TRI-FUSION(T, p, r) renvoie bien le tableau T trié entre p et r.

2.4. **Analyse de la complexité des algorithmes Diviser pour Régner.** Supposons que l'on ait un algorithme de type Diviser pour Régner, qui suit le principe suivant : pour être résolu le problème initial sur une instance de taille n est divisé en a sous-problèmes sur des instances de

tailles $\approx \frac{n}{b}$. Puis ces solutions sont combinées en une solution du problème de taille n. Il y a donc plusieurs événements qui peuvent avoir un coût :

- le coût (qui dépend de la taille de l'instance) nécessaire à diviser le problème de taille n. Il sera noté $D(n)$.
- le coût (dépendant lui-aussi de la taille de l'instance) pour combiner les solutions des sous-problèmes en une solution du problème de taille n, noté $C(n)$.
- Enfin, notons $T(n)$ le temps nécéssaire pour trouver la solution de taille n.

Le principe récursif diviser pour régner nous amène à une récurrence du type $T(n) = aT(\frac{n}{b}) + D(n) + C(n)$. De plus on peut supposer que pour les instances de taille 1, le coup pour trouver la solution est fixé et donc constant, $T(1) = c$. Les coûts d'exécution des algorithmes de type Diviser pour Régner suivent donc toujours le même type de récurrence mathématique. Nous allons voir qu'en général on sait assez bien trouver le comportement asymptotique des fonctions T définies ainsi. Avant de donner une borne pour le coût d'exécution de l'algorithme TRI-FUSION, accordons nous quelques rappels concernant les comportements asymptotiques.

2.5. **Rappel sur les notions de comportement asymptotique.** On note $\mathcal{F}_{\mathbb{N}}$, l'ensemble des fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$ (Ce sont les suites à valeurs dans $\mathbb{R}^+$). En algorithmique, nous nous intéressons spécifiquement au comportement au voisinage de l'infini de fonctions de $\mathcal{F}_{\mathbb{N}}$, ce qui est une (bonne ?) mesure permettant de comparer l'efficacité des algorithmes. Pour ce faire, étant données deux fonctions f et g de $\mathcal{F}_{\mathbb{N}}$, on dira que f est *dominée* par g au voisinage de l'infini (ou encore que g est une *borne asymptotique supérieure* de f) si et seulement s'il existe deux constantes strictement positives c et n_0 telles que pour tout $n > n_0$ on a $f(n) < cg(n)$. On notera alors $f = O(g)$ ou bien $g = \Omega(f)$.

On dira que f et g sont *semblables* au voisinage de l'infini (ou encore que g est une *borne asymptotique approchée* de f) si et seulement si $f = O(g)$ et $f = \Omega(g)$. On notera alors $f = \Theta(g)$.

En particulier, étant donnée une fonction f, on essayera toujours de lui trouver un équivalent parmi une échelle de comparaisons simples (du type $n^a \log(n)^b$, exponentielle,...).

Propriété 1. *La relation R définie par $fRg \Leftrightarrow f = \Theta(g)$ est une relation d'équivalence.*
La relation R_1 (resp. R_2) définie par $fR_1g \Leftrightarrow f = \Omega(g)$ (resp. $fR_2g \Leftrightarrow f = O(g)$) est un préordre (i.e. une relation binaire, reflexive et transitive) sur $\mathcal{F}_{\mathbb{N}}$.

2.6. **Coût asymptotique en nombre de comparaisons pour l'algorithme TRI-FUSION.** On remarque que dans le TRI-FUSION, le nombre de comparaisons est le même pour toutes les instances (les séquences) de même taille. En particulier, il faut autant de temps au TRI-FUSION pour trier $(1, 2, 3, 4, 5, 6, 7, 8, 9)$ que pour trier $(5, 1, 8, 7, 4, 9, 6, 2, 4)$!, alors que pour le TRI-INSERTION le nombre de comparaisons dépendait du "désordre" de la suite. Donc, pour TRI-FUSION, il suffit de chercher à trouver le nombre de comparaisons faites pour une instance de longueur n.

Le fait de diviser une séquence en deux sous-séquences se fait sans comparaison. La seule partie de TRI-FUSION où il y a des comparaisons est dans FUSION. C'est-à-dire dans l'algorithme qui combine les sous-solutions. Or nous savons que FUSION fait n comparaisons pour construire une séquence triée de longueur n. Donc, si $T(n)$ désigne le nombre de comparaisons dans TRI-FUSION pour une instance de longueur n, on obtient la récurrence suivante : $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n$. De plus, il est clair que $T(1) = 0$. Pour évaluer cette fonction au voisinage de l'infini, on va commencer par calculer $T(2^n)$.

Lemme 1. *Soit T définie par $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n$ et $T(1) = 0$, alors $T(2^n) = n2^n$.*

Preuve. Si $n = 0$, on a bien $T(2^n) = T(1) = 0$. Montrons que $\forall n, T(2^n) = n2^n \Rightarrow T(2^{n+1}) = (n+1)2^{n+1}$. Comme $T(2^{n+1}) = 2T(2^n) + 2^{n+1}$, si $T(2^n) = n2^n$ alors $T(2^{n+1}) = 2(n2^n) + 2^{n+1} = (n+1)2^{n+1}$. □

Lemme 2. *Soit T définie par $T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + n$ et $T(1) = 0$, alors T est croissante.*

Preuve. $T(1) - T(0)$ est clairement positif. Montrons que

$$\forall n, (\forall i \leq n, T(i) - T(i-1) \geq 0) \Rightarrow T(n+1) - T(n) \geq 0.$$

Comme pour n pair

$$T(n+1) - T(n) = (T(n/2) + T(n/2+1) + n + 1) - (T(n/2) + T(n/2) + n) \\ = T(n/2+1) - T(n/2) + 1 \quad (1)$$

et pour n impair

$$T(n+1) - T(n) = (T((n+1)/2) + T((n+1)/2) + n + 1) - (T((n-1)/2) + T((n+1)/2) + n) \\ = T((n+1)/2) - T((n+1)/2 - 1) + 1, \quad (2)$$

si $\forall n, (\forall i \leq n, T(i) - T(i-1) \geq 0)$, alors en utilisant (1) et (2), on a $T(n+1) - T(n) \geq 0$. Par induction, il s'ensuit que T est croissante. □

Théorème 1. *Le nombre de comparaisons $T(n)$ que fait l'algorithme TRI-FUSION pour trier une séquence de longueur n vérifie $T(n) = \Theta(n \log(n))$.*

Preuve. Pour tout $n \in \mathbb{N}$, il existe un unique m tel que $2^m \leq n < 2^{m+1}$ (il suffit de prendre $m = \lfloor \log_2 n \rfloor$). Comme T est croissante, on a $T(2^m) \leq T(n) < T(2^{m+1})$. Donc $m2^m \leq T(n) < (m+1)2^{m+1}$. Ceci implique pour n assez grand que $c_1 n \log n < T(n) < c_2 n \log n$. □

2.7. **Le théorème maître.** Le théorème maître permet d'avoir une idée du comportement asymptotique de la plus part des récurrences induites par les algorithmes de type Diviser pour Régner, nous le proposons sans démonstration à ce stade.

Théorème 2. *(Résolution de récurrences « Diviser pour Régner »). Soient $a \geq 1$ et $b > 1$ deux constantes, soient f une fonction à valeurs dans $\mathbb{R}^+$ et U une fonction de $\mathbb{N}^*$ dans $\mathbb{R}^+$ vérifiant pour tout n suffisamment grand l'encadrement suivant : $aU(\lfloor \frac{n}{b} \rfloor) + f(n) \leq U(n) \leq aU(\lceil \frac{n}{b} \rceil) + f(n)$. U peut alors être bornée asymptotiquement comme suit :*

1. *Si $f(n) = O(n^{(\log_b a) - \epsilon})$ pour une certaine constante $\epsilon > 0$, alors $U(n) = \Theta(n^{\log_b a})$.*
2. *Si $f(n) = \Theta(n^{\log_b a})$, alors $U(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Si $f(n) = \Omega(n^{(\log_b a) + \epsilon})$ pour une certaine constante $\epsilon > 0$, et si on a asymptotiquement pour une constante $c < 1$, $af(n/b) < cf(n)$, alors $U(n) = \Theta(f(n))$.*

Remarques : Le théorème maître ne couvre pas toutes les possibilités pour $f(n)$. Ainsi, il y a une lacune entre les cas 1 et 2 (et les cas 2 et 3). Par exemple la récurrence $U(n) = 2U(n/2) + n \log(n)$, n'est pas traitable avec le théorème maître.

Exemples : $U(n) = 2U(\lfloor n/2 \rfloor) + O(n^a)$ si $a = 1/2$, on est dans le cas 1 et donc $U(n) = \Theta(n)$. Si $a = 1$ alors on est dans le cas 2 et donc $U(n) = \Theta(n \ln(n))$. Si $a = 2$ alors on est dans le cas 3 et donc $U(n) = \Theta(n^2)$.

2.8. **Le TRI-RAPIDE.** Le TRI-RAPIDE, aussi appelé "tri de Hoare" (du nom de son inventeur) ou "tri par segmentation" ou "tri des bijoutiers" ou en anglais "quicksort", est un algorithme de tri intéressant à plus d'un titre. Tout d'abord, comme le TRI-INSERTION, l'algorithme TRI-RAPIDE opère uniquement sur la séquence à trier. En particulier, il ne crée pas de tableau auxiliaire comme c'est le cas pour le TRI-FUSION. De plus, l'algorithme TRI-RAPIDE a un coût en nombre de comparaisons raisonnable. Nous allons montrer en effet que le nombre moyen de comparaisons effectuées pour trier une séquence "quelconque" de longueur n est $\Theta(n \log n)$. Ce qui en moyenne est aussi bon que le TRI-FUSION.

L'algorithme TRI-RAPIDE repose sur le principe Diviser pour Régner de la manière suivante :

Diviser : Soit $T[p..r]$ le tableau contenant la séquence d'entiers que l'on étudie. On fixe une valeur v de T (ici $v = T(r)$) appelée *pivot* et on réarrange le tableau $T[p..r]$ de telle sorte qu'après cela les tableaux éventuellement vides, $T[p..q-1]$, $T[q+1..r]$, contiennent respectivement tous les éléments de $T[p..r-1]$ dont les valeurs sont plus petites ou égales à v et toutes les valeurs de

$T[p..r-1]$ strictement plus grandes que v et que $T[q] = v$.
RÉGNER : On résout par appel récursif le problème sur les sous-tableaux $T[p..q-1]$ et $T[q+1, r]$;
COMBINER : Il n'y a rien à faire.

Commençons par donner le pseudo-code de REARRANGEMENT(T, p, r).

Algorithme 5 : REARRANGEMENTS

Entrées : $T[1..n]$ un tableau de nombres, p et r des entiers.
Sorties : un entier (la position du pivot).

```
1  REARRANGEMENTS(T, p, r)
2  v :=T[r];
3  i := p − 1;
4  pour j allant de p à  r − 1 faire
5  |  si T[j] ≤ v alors
6  |  |  i := i + 1;
7  |  |  tmp := T[i] ; T[i] := T[j] ; T[j] := tmp;
8  tmp := T[i + 1];
9  T[i + 1] := T[r];
10 T[r] := tmp;
11 retourner i + 1;
```

Analyse de l'algorithme REARRANGEMENT

Preuve de Terminaison :
Nous n'avons dans cet algorithme qu'une boucle de type compteur. Donc l'algorithme s'arrête après un nombre fini d'étapes.

Preuve de Validité :
Nous allons utiliser pour ce faire l'invariant de boucle (pour la boucle de la ligne 4) suivant :

(1) Si $p \leq k \leq i$, alors $T[k] \leq v$.

(2) Si $i + 1 \leq k \leq j - 1$, alors $T[k] > v$.

(3) Si $k = r$, alors $T[k] = v$.

Initialisation : Avant la première itération, $i = p - 1$ et $j = p$. Il n'y a pas de valeur entre p et i, ni de valeur entre $i + 1$ et $j - 1$, de sorte que les deux premières conditions de l'invariant de boucle sont satisfaites de manière triviale. L'affection en ligne 2 satisfait à la troisième condition.
Par induction : il y a deux cas à considérer, selon le résultat du test en ligne 5. Quand $T[j] > v$ l'unique action faite dans la boucle est d'incrémenter j. Une fois j incrémenté, la condition 2 est vraie pour $T[j-1]$ et tous les autres éléments restent inchangés, donc l'invariant est préservé. Quand $T[j] \leq v$, la variable i est incrémenté, $T[i]$ et $T[j]$ sont échangés, puis j est incrémenté. Compte tenu de la permutation, on a maintenant, $T[i] \leq v$ et la condition 1 est respectée. De même, on a aussi $T[j - 1] > v$, car l'élément qui a été permuté avec $T[j - 1]$ est, d'après l'invariant de boucle, plus grand que v.
Validité finale : À la fin, $j = r$. Par conséquent, chaque élément du tableau est dans l'un des trois ensembles décrits par l'invariant et l'on a réarrangé les valeurs du tableau de telle sorte qu'il débute par les valeurs inférieures ou égales à v, puis l'élément v, puis se finit par les valeurs supérieures à v.

Analyse de la Complexité en nombre de comparaisons :
On fait une comparaison à chaque itération de la boucle de la ligne 4, soit $r - p$ comparaisons. Donc pour une instance de longueur n, l'algorithme REARRANGER opère $n - 1$ comparaisons.

Maintenant, il est aisé de décrire un algorithme de type Diviser pour Régner à partir de REARRANGEMENT, en voici le pseudo-code :

Algorithme 6 : TRI-RAPIDE

Entrées : $T[1..n]$ un tableau de nombres, p et r des entiers.
Sorties : Rien mais le tableau T est trié entre p et r.

```
1 TRI-RAPIDE(T, p, r)
2 si p < r alors
3 |  q := REARRANGEMENT(T, p, r);
4 |  TRI-RAPIDE(T, p, q − 1);
5 |  TRI-RAPIDE(T, q + 1, r);
```

Pour trier un tableau T, il suffit alors de faire l'appel TRI-RAPIDE(T, 1, longueur(T)).

Analyse de l'algorithme TRI-RAPIDE

Preuve de Terminaison :
Par induction sur la longueur de la zone $[p..r]$ à trier, l'algorithme TRI-RAPIDE(T, p, r) avec $p = r$ ne fait aucun appel à REARRANGEMENT(T, p, r) donc il est fini. Supposons que pour tous r et p tels que $r - p < k$, l'algorithme TRI-FUSION(T, p, r) se termine, alors pour tout p, TRI-FUSION$(T, p, p + k)$ se termine aussi car il appelle récursivement trois fonctions qui par hypothèse d'induction se terminent.

Preuve de Validité :
Par induction sur la longueur de la zone à trier, si $p = r$, c'est trivial. Supposons que pour tous r et p tels que $r - p < k$, l'algorithme TRI-RAPIDE(T, p, r) renvoie le tableau T trié entre p et r inclus, Considérons TRI-RAPIDE(T, p, r) avec $r - p = k$. Comme TRI-RAPIDE$(T, p, q - 1)$ et TRI-RAPIDE$(T, q + 1, r)$ sont valides par hypothèse d'induction, on a bien que tout le tableau T est trié de p à r.

Analyse de la Complexité en nombre de comparaisons dans le pire des cas :
Notons $T(n)$ le nombre de comparaisons dans le pire des cas pour les instances de taille n. En utilisant la construction récursive de TRI-RAPIDE et le fait que l'algorithme REARRANGEMENT opère $n - 1$ comparaisons sur une séquence de longueur n, on obtient la récurrence suivante :

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + n - 1.$$

On va vérifier par récurrence l'hypothèse que pour tout n, $T(n) < n^2$. Supposons que ceci soit vrai pour $n < n_0$, alors

$$T(n_0) < \max_{0 \leq q \leq n_0 - 1} \left(q^2 + (n_0 - q - 1)^2\right) + n_0 - 1.$$

Or la fonction $f(q) = q^2 + (n_0 - q - 1)^2$ est maximum aux extrémités de l'intervalle $0 \leq q \leq n_0 - 1$. Donc,

$$\max_{0 \leq q \leq n_0 - 1} \left(q^2 + (n_0 - q - 1)^2\right) = (n_0 - 1)^2,$$

et $T(n_0) < (n_0 - 1)^2 + n_0 - 1 < n_0^2$. Donc $T(n) = O(n^2)$
En regardant le fonctionnement de l'algorithme TRI-RAPIDE sur l'instance $(1, 2, ..., n)$, on remarque qu'il exécute $(n - 1) + (n - 2) + ... + 1$ comparaisons, donc $T(n) = \Theta(n^2)$.

On peut être quelque peu déboussolé au regard des résultats obtenus. En effet, dans le pire des cas, l'algorithme TRI-RAPIDE est plutôt mauvais (quadratique). De plus, il est mauvais sur des instances déjà triées ! Pourtant, comme nous allons le montrer le nombre de comparaisons en moyenne est très bon (en $n \log n$ pour une instance de taille n) et surtout, il permet de trier en place, c'est à dire sans faire de copie du tableau. Aux vues des autres avantages qu'il présente, c'est un algorithme

très souvent implémenté (par exemple, c'est l'algorithme implémenté dans JAVA pour les tris).

Analyse de la Complexité en nombre moyen de comparaisons :
Le calcul de la complexité du tri rapide en moyenne est un peu plus compliqué. Déjà, il faut définir un espace de probabilité. On va considérer l'ensemble E_n de toutes les séquences de longueur n dont tous les éléments sont distincts. Soit $L = (a_1, ..., a_n)$ une séquence prise au hasard (équiprobablement) dans E_n, la probabilité que a_n soit après triage le i-ème élément de la séquence est clairement $1/n$, de sorte que le coût moyen en nombre de comparaisons C_n vérifie : $C_0 = 0$, $C_1 = 0$ et $C_n = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (C_i + C_{n-i-1})$.

Le terme générique C_n peut s'écrire :

$$
\begin{aligned}
C_n &= n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} C_i \\
&= n - 1 + \frac{2}{n} C_{n-1} + \frac{2}{n} \sum_{i=0}^{n-2} C_i \\
&= n - 1 + \frac{2}{n} C_{n-1} + \frac{n-1}{n} \left(n - 2 + \frac{2}{n-1} \sum_{i=0}^{n-2} C_i \right) - \frac{(n-1)(n-2)}{n} \\
&= \frac{2}{n} C_{n-1} + \frac{n-1}{n} (C_{n-1}) + \frac{2n-2}{n} \\
&= \frac{n+1}{n} C_{n-1} + \frac{2(n-1)}{n}
\end{aligned}
\tag{3}
$$

En posant $D_n = C_n/(n+1)$, la récurrence s'écrit alors :

$$D_0 = 0 \text{ et } D_n = D_{n-1} + \frac{2}{(n+1)} - \frac{2}{(n+1)\,n}.$$

On rappelle que la série $H_n = H_{n-1} + 1/n$ est la *série harmonique* qui vérifie asymptotiquement $H_n = \ln(n) + 0,577... + \Theta(1/n)$ où 0,577... est la *constante d'Euler*. On en déduit, comme le terme $\frac{2}{(n+1)n}$ est négligeable, que $D_n = \Theta(\ln(n))$ et donc que $C_n = \Theta(n \ln(n))$.

2.9. **Recherche d'un élément défectueux avec une balance.** On dispose d'un ensemble de n pièces d'or toutes de même poids sauf une défectueuse qui est plus légère. Le problème consiste à retrouver la pièce défectueuse en utilisant une balance de type Roberval (balance à 2 plateaux qui permet de dire uniquement s'il y a un plateau plus lourd que l'autre). On va modéliser le problème de la façon suivante : Soit $T[1..n]$ un tableau de caractères qui contient $(n-1)$ lettres P (les bonnes pièces d'or) et une lettre Q (la pièce défectueuse). On cherche donc à trouver l'indice de la case qui contient la lettre Q grâce à une fonction PESÉE(T, n_1, n_2, n_3, n_4) qui renvoie $(n_1, n_2 - 1)$ si le Q est dans l'intervalle $[n_1, n_2[$, $(n_2, n_3 - 1)$ si le Q est dans l'intervalle $[n_2, n_3[$ et (n_3, n_4) sinon. De plus, on suppose que l'on ne peut utiliser la fonction PESÉE que si les intervalles $[n_1, n_2]$ et $[n_2, n_3]$ sont de même longueur. Par convention, $[n, m[= \emptyset$ si $m \leq n$.
On propose de résoudre ce problème grâce à l'algorithme de type diviser pour règner suivant :
Diviser : On répartit les n pièces en 3 tas de tailles respectivement $[n/3]$, $[n/3]$, $n - 2[n/3]$. ($[x]$ indique l'entier le plus proche de x et n si $x = n + 1/2$).
Régner : On pèse les deux premiers tas, s'ils sont de même poids, on recommence inductivement avec le tas 3, sinon on recommence avec le tas vers lequel la balance ne penche pas (le plus léger).

Ceci donne le pseudo-code suivant :

Algorithme 7 : ChercheQ

Entrées : T un tableau contenant des P et un Q, d et f deux entiers. On suppose que la position du Q est entre d et f.

Sorties : n la position de la lettre Q dans $T[d..f]$.

```
1 ChercheQ(T, d, f)
2 si d = f alors
3 |  retourner d
4 sinon
5 |  e = [(f−d+1)/3];
6 |  n1 = d + e;
7 |  n2 = d + 2e;
8 |  (p, q) =PENSÉE(T, d, n1, n2, f);
9 |  retourner ChercheQ(T, p, q);
```

Analyse de l'algorithme ChercheQ

Preuve de Terminaison :
Par induction sur la longueur $f - d$ de l'intervalle $[d, f]$, si la longueur est 1, l'algorithme retourne d, sinon il rappelle une instance qui par induction se termine.

Preuve de Validité :
Par induction sur la longueur de l'intervalle $[f, d]$, si la longueur est 1, l'algorithme retourne la bonne valeur. Supposons qu'il retourne la bonne valeur pour les intervalles plus petits que l, quand on lance l'exécution sur un intervalle de longueur l, la pensée permet de déterminer un sous-intervalle de taille inférieure à l (de l'ordre de $l/3$) dans lequel se trouve le Q. Par induction, l'appel de ChercheQ sur cet intervalle renvoie la bonne valeur. Donc ChercheQ renvoie la bonne valeur.

Analyse de la Complexité en nombre de pesées :
Notons $T(n)$ le nombre de pesées nécessaires pour trouver Q dans un tableau de taille n. Le principe Diviser pour Régner permet de donner la formule de récurrence pour T suivante : $T(1) = 0$ et $T(n) = T(n/3) + 1$. On en déduit que $T(n) = \Theta(\log_3(n))$.

2.10. **Multiplication de deux entiers par la méthode de Karatsuba.** Dans cette section, nous évaluerons le coût des algorithmes en nombre de multiplications élémentaires effectuées (multiplications de nombres à un chiffre). On remarque que pour le produit de deux nombres de n chiffres en utilisant la méthode naïve (celle apprise à l'école), on fait n^2 multiplications élémentaires, le coût T en calcul d'une multiplication de deux nombres à n chiffres est donc $T(n) = \Theta(n^2)$.

L'algorithme de Karatsuba (A. Karatsuba, Ofman Yu, Multiplication of multiple numbers by mean of automata, Dokadly Akad. Nauk SSSR 145, no 2, 1962, pp293-294.) est une application du principe " diviser pour régner " qui permet d'améliorer le coût des multiplications des grands nombres. Le principe en est assez simple :

Soient a et b deux nombres positifs écrits en base 2 de $n = 2k$ chiffres, on peut écrire $a = (a_1 \times 2^k + a_0)$ et $b = (b_1 \times 2^k + b_0)$ avec a_0, b_0, a_1, b_1 des nombres binaires à k chiffres. On remarque alors que le calcul de $(a_1 \times 2^k + a_0)(b_1 \times 2^k + b_0) = a_1b_1 \times 2^{2k} + (a_1b_0 + a_0b_1) \times 2^k + a_0b_0$ ne nécessite pas les quatre produits a_1b_1, a_1b_0, a_0b_1 et a_0b_0, mais peut en fait être effectué seulement avec les trois produits a_1b_1, a_0b_0 et $(|a_1 - a_0|)(|b_1 - b_0|)$ en regroupant les calculs sous la forme suivante (on note $sgn(x)$ le signe de x) :

$(a_1 \times 2^k + a_0)(b_1 \times 2^k + b_0) = a_1b_1 \times 2^{2k} + (a_1b_1 + a_0b_0 - sgn(a_1 - a_0)sgn(b_1 - b_0)(|a_1 - a_0|)(|b_1 - b_0|)) \times 2^k + a_0b_0$

Ce que l'on peut écrire ainsi :
Diviser : On décompose a et b des nombres de $2k$ chiffres en $a = (a_1 \times 2^k + a_0)$ et $b = (b_1 \times 2^k + b_0)$ où a_0, b_0, a_1, b_1 sont des nombres à k chiffres .

RÉGNER : On résout par appel récursif le problème pour a_0b_0, a_1b_1 et $(|a_1 - a_0|)(|b_1 - b_0|)$;
COMBINER : On obtient le produit ab en faisant $(a_1 \times 2^k + a_0)(b_1 \times 2^k + b_0) = a_1b_1 \times 2^{2k} + (a_1b_1 + a_0b_0 - sgn(a_1 - a_0)sgn(b_1 - b_0)(|a_1 - a_0|)(|b_1 - b_0|)) \times 2^k + a_0b_0$.

On admet que les opérations $/2^k$ (division entière par une puissance de 2) et $\bmod 2^k$ (reste de la division entière par une puissance de 2) ne nécessitent pas de multiplication. En fait, ces opérations se font en machine en temps constants et sont négligeables. D'où le pseudo-code suivant :

Algorithme 8 : KARATSUBA

Entrées : deux entiers positifs.
Sorties : un entier.

```
1  KARATSUBA(a, b)
2  k := max(⌊log2 a⌋ + 1, ⌊log2 b⌋ + 1);
3  si k ≤ 1 alors
4  |  retourner ab;
5  sinon
6  |  s = ⌊k/2⌋;
7  |  a0 = a mod 2^s;
8  |  a1 = a/2^s;
9  |  b0 = b mod 2^s;
10 |  b1 = b/2^s;
11 |  x := KARATSUBA(a0, b0);
12 |  y := KARATSUBA(a1, b1);
13 |  z := KARATSUBA(((|a1 − a0|), (|b1 − b0|));
14 |  retourner y × 2^2s + (x + y − sgn(a1 − a0)sgn(b1 − b0)z) × 2^s + x;
```

Analyse de l'algorithme KARATSUBA

Preuve de Terminaison :
Par induction sur $k = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor)$. Si $k = 1$ alors, on ne fait qu'une multiplication, donc l'algorithme est fini. Supposons que pour tout $k < n_0$, l'algorithme KARATSUBA(a, b) se termine, alors pour tous a et b avec $n_0 = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor)$, KARATSUBA$(a, b)$ se termine aussi car il appelle recursivement trois copies de KARATSUBA sur des instances plus petites (qui par hypothèse d'induction se terminent).

Preuve de Validité :
Par induction sur $k = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor)$. Si $k = 1$ alors, KARATSUBA(a, b) renvoie ab. Supposons que pour tout $k < n_0$, l'algorithme KARATSUBA(a, b) renvoie ab, alors pour tous a et b avec $n_0 = \max(\lfloor \log_2 a \rfloor, \lfloor \log_2 b \rfloor)$, KARATSUBA$(a, b) = y \times 2^{2k} + (x + y - sgn(a_1 - a_0)sgn(b_1 - b_0)z) \times 2^k + x$ qui vaut ab car par récurrence $x = a_0b_0$, $y = a_1b_1$ et $z = (|a_1 - a_0|)(|b_1 - b_0|)$).

Analyse de la Complexité en nombre de multiplications élémentaires :
Notons $T(n)$ le nombre de multiplications élémentaires nécessaires pour multiplier deux nombres de n chiffres. Le principe Diviser pour Régner permet de donner la formule de récurrence pour T suivante : $T(1) = 1$ et $T(n) = 3T(n/2)$. On en déduit que $T(n) = \Theta(n^{\frac{\ln(3)}{\ln(2)}})$. Or $\frac{\ln(3)}{\ln(2)} \approx 1.585$, ce qui bien mieux que le n^2 de la multiplication naïve.

E-mail address: olivier.bodini@lipn.fr