Fiche : Complexité algorithmique et tris

Contenus et objectifs

Cette fiche introduit deux mesures de performance d'un algorithme : la complexité temporelle pour le temps de calcul des traitements et la complexité spatiale pour l'occupation mémoire des données. Trois algorithmes de tri (par insertion, rapide et fusion) sont présentés en guise d'exemples d'illustration.

A la fin de cette fiche, l'étudiant doit être capable de calculer la complexité d'un algorithme dans le pire cas et de comprendre le fonctionnement des algorithmes de tri classiques.

Seule la complexité temporelle dans le pire cas est abordée ici, pour celles et ceux qui souhaitent aller plus loin, ils peuvent lire le chapitre 2 du livre « Types de données et algorithmes » de Christine Froidevaux, Marie-Claude Gaudel, et Michèle Soria. Ediscience International. Ou encore les chapitres 1 à 4 du livre « Introduction à l'Algorithmique » de Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest et Clifford Stein. Dunod/ McGraw-Hill.

A. Introduction

Face à la résolution d'un problème donné, nous avons souvent le choix entre plusieurs algorithmes/programmes possibles. Nous sommes alors amenés à nous poser plusieurs questions :

- ⇨ Lequel ou lesquels sont les plus adaptés à notre application ?
- ⇨ Quels critères doit-on prendre en compte ?
- ⇨ Comment mesurer l'efficacité d'un algorithme ?
- ⇨ Cette mesure est-elle vraie indépendamment des machines utilisées ?
- ⇨ L'augmentation de la taille des données peut-elle rendre un problème en pratique insolvable ?

La complexité algorithmique a donc pour objectif d'apporter des réponses à ces questions.

ATTENTION : Ne pas confondre la complexité théorique d'un problème qui mesure la difficulté à trouver un algorithme de résolution efficace, avec la complexité algorithmique qui mesure son efficacité. Ces deux complexités sont complémentaires. Toutefois, la complexité d'un problème est beaucoup plus difficile à démontrer (voir le livre théorie de la complexité de Garey et Johnson ou encore les cours à propos des problèmes d'Optimisation Combinatoire et Recherche Opérationnelle).

B. Deux grandeurs caractéristiques d'un programme

L'exécution d'un programme demande à la fois :

- ⇨ du **temps** pour exécuter les opérations (traitements) et
- ⇨ de **l'espace mémoire** pour stocker le programme, les données et les résultats.

Une mesure possible du temps serait par exemple de compter le nombre d'opérations fondamentales exécutées sur une machine donnée et de mesurer le temps de chaque opération.

Pour l'espace mémoire, on pourrait compter le nombre d'octets (un octet est l'unité de base pour la taille mémoire d'un ordinateur) pour stocker les instructions, les données ainsi que les structures de données sous forme de classes d'objets qui les manipulent.

Mais… avec ces mesures, on serait amené à un résultat assez restrictif du genre :
" L'algorithme A implémenté avec un programme P sur une machine M et exécuté sur les données D, nécessite un temps T pour le calcul et un nombre de O octets de mémoire. "

Or, d'un résultat certes intéressant mais aussi restrictif, on préfère avoir un résultat indépendant de la machine, du langage de programmation, du compilateur :
" Sur toute machine, et quelque soit l'environnement de programmation utilisé, l'algorithme A1 est meilleur que l'algorithme A2 pour des données de grande taille. "

ou encore

" L'algorithme A est optimal pour résoudre le problème Q. "

Ici, l'idée est vraiment de capter **l'efficacité intrinsèque** d'un algorithme en **faisant abstraction de tout détail d'implémentation**.

B1. Identification des opérations fondamentales à mesurer

Pour cette mesure, il faut d'abord déterminer les opérations fondamentales des programmes :
⇨ Or à quel niveau doit on considérer qu'une opération est fondamentale : binaire, instruction machine, langage de programmation ?
⇨ Quelles sont les opérations d'un programme doit-on considérer comme fondamentales ?
- Pour un calcul d'algèbre linaire, les multiplications et les additions sont fondamentales.
- Pour une liste triée, on peut considérer que les comparaisons entre deux éléments et les déplacements des éléments comme fondamentaux.
- Pour une application de Base de Données, les Entrées-Sorties sur disque peuvent être fondamentales.
- Pour une application WEB, les accès aux réseaux sont fondamentaux.
- Etc.

Nous pouvons donc conclure qu'il n'existe pas de règle absolue de mesure, mais toutefois, pour chaque algorithme, il convient de **bien préciser à quel niveau d'abstraction on se place et quelles sont les opérations fondamentales considérées et ce sur quelles données sont effectuées ces opérations.**

Nous devons ainsi souligner deux points importants :
1. Suivant le niveau d'abstraction et les opérations fondamentales choisies, le degré de précision de l'analyse peut varier.
2. Le temps d'exécution est proportionnel aux opérations fondamentales choisies. Ainsi si ces opérations fondamentales choisies sont trop différentes d'un algorithme à un autre, les algorithmes peuvent ne plus être comparables entre eux.

B2. Calcul de la complexité temporelle

Une fois les opérations fondamentales définies, il suffit de les compter. Malheureusement, il n'existe pas de système de comptage tenant compte de la syntaxe des algorithmes. Nous allons pour simplifier la compréhension et sans perdre de généralité **nous placer au niveau des instructions du langage algorithmique**.

On peut ainsi faire les remarques suivantes :

a) Lorsque les opérations sont dans une **séquence d'instructions, leurs nombres s'ajoutent**.

Par exemple :

```
a=0 ;  (1 opération : c'est l'affectation)
c=0 ;  (idem)
a=a+1; (2 ops. : 1 addition et 1 affectation)
b=a+c; (idem)
```

Cette séquence d'instructions vaut donc en tout 6 « unités » de temps. On considère que chaque opération vaut une unité de temps, même si en pratique, une affection est plus coûteuse qu'une addition, mais cet écart reste constant.

b) Pour les **instructions conditionnels**, il est souvent difficile de déterminer quelle branche de la condition est exécutée et comptée. D'où, si T(X) est le nombre d'opérations fondamentales (unités de temps) de la séquence X, alors :

T(`SI C ALORS I1 SINON I2 FIN SI`) <= T(`C`) + max{ T(`I1`), T(`I2`) }

Le temps d'une instruction conditionnelle est donc **borné par le temps de test de la condition plus le plus grand temps des deux branches de l'instruction**.

c) Pour les **instructions itératives (boucles)**, le **total des opérations fondamentales est Σ T(i)** où i est la variable de contrôle de la boucle et T(i) le nombre d'opérations fondamentales à la $i^{ème}$ itération.

d) Pour les **appels de méthodes non récursives**, il suffit de compter les nombres d'opérations fondamentales suivant les **trois règles précédentes (a, b et c)**. Si jamais, il y a des appels à d'autres fonctions, il suffit d'abord d'évaluer les nombres d'opérations fondamentales de ces autres fonctions avant de les additionner ensemble.

e) Pour les **appels de méthodes récursives**, le comptage du nombre d'opérations fondamentales se ramène en général à la **résolution d'équations de récurrence**. En effet, le nombre d'opérations fondamentales T(n) à l'appel de la fonction récursive avec un argument de taille n, s'écrit, selon la récurrence, en fonction de divers T(k), pour k<n des arguments de taille inférieure à n.

Prenons l'exemple de la fonction factorielle de la fiche Récursivité et Diviser-pour-Régner :

```
ALGO fact
Entrées: Entier n;
Sortie: Entier;
-----------------------------------------
   SI (n==0) ALORS RETOURNER 1;
   SINON RETOURNER n*fact(n-1) ;
   FIN SI
FIN ALGO
```

Si on choisit ici l'opération multiplication comme fondamentale du cas « sinon », nous avons clairement comme système d'équations de récurrence **T(0)=0 et T(n)=T(n-1)+1**.

Et en résolvant ce système très simple, nous obtenons **T(n)=n**.

Pour la résolution des systèmes d'équations de récurrence, vous pouvez mettre en application vos connaissances en mathématiques.

Remarque : nous avons ici dans un but d'illustration, volontairement omis de tenir compte du coût des appels de fonction, qui peut parfois être très coûteux. Dans ce cas, nous devons compter les appels comme des opérations fondamentales.

B3. Un autre exemple : la recherche d'un entier dans un tableau d'entiers

En langage algorithmique, nous avons quelque chose comme suit :

```
Entier L=nouveau Entier[n]; // supposons que l'on connaît n
Entier X ; // élément recherché
Entier j ; // compteur pour parcourir le tableau L

(1)  j=0 ;
(2)  TANT QUE ( (j<n) ET (L[j] != X) )
(3)       j=j+1;
     FIN TANT QUE
(4)  SI (j>=n) ALORS j=-1 FIN SI //j==-1 si X n'est pas dans L
```

Dans l'analyse de cet algorithme, on doit compter le nombre d'itérations et le nombre d'opérations fondamentales par itération.

Comme **opérations fondamentales**, nous ne pouvons prendre que la **comparaison de X avec les éléments du tableau**. En effet, tout le reste concerne soit la programmation, soit l'implémentation de la liste par un tableau. Avec une implémentation de la liste avec une liste chaînée, la ligne (3) et le test j<n de la ligne (2) n'interviendraient plus de cette manière.

Deux cas se présentent donc :

- Le nombre d'itérations est égal à n, et j= -1, si X n'est pas dans la liste.
- Ce même nombre est égal à j, rang de la première occurrence de X, si X est dans la liste.

Cet exemple montre la complexité dépend de trois points essentiels :
1. **le choix des opérations fondamentales,**
2. **la taille des données (ici n),**
3. **pour une taille fixée, les différentes données possibles (ici j varie).**

Remarque : La taille d'un problème dépend souvent du problème, par exemple : dans le calcul matriciel, c'est la taille des matrices ; dans les listes, c'est le nombre d'éléments ; etc.

B3. Autres critères d'évaluation

Nous avons privilégié jusque là la complexité temporelle, mais ce n'est pas le seul critère qui entre en ligne de compte pendant la phase d'analyse d'un algorithme, il y a aussi **l'occupation espace mémoire**, la **simplicité de l'algorithme** ou encore l'**adéquation d'un algorithme à certaines données.**

On peut aussi définir des mesures pour se rendre compte du critère occupation mémoire.

Très souvent, un algorithme plus rapide utilise plus de mémoire, mais si la mémoire centrale est limitée, il convient de trouver un compromis espace-temps. Pour cela, prenez l'exemple de la suite de Fibonacci, avec une implémentation récursive et une implémentation itérative utilisant un tableau stockant les résultats intermédiaires (voir la fiche Récursivité et Diviser-pour-Régner).

Par ailleurs, la simplicité d'un programme permet d'économiser souvent le " temps humain " (maintenance, compréhension), ceci est surtout intéressant dans les cas où le programme développé est utilisé peu de fois. On peut alors se permettre dans ce cas là d'avoir un programme moins performant mais plus rapide à développer.

Enfin, pour les algorithmes numériques où la précision et la stabilité sont aussi des critères importants.

Il convient donc de **bien prendre en compte le contexte de développement et d'utilisation du programme pendant la phase d'analyse**.

C. Complexité au pire cas

C1. Définition

Soit $coût_A(d)$ la complexité en temps de l'algorithme A sur la donnée d. La complexité dans le **pire des cas** est donnée par

$$Max_A(n) = \max\{ cout_A(d), d \in D_n \}$$

où D_n est le domaine de réalisation de la donnée d.

Cette complexité dans le pire cas donne une borne supérieure sur l'efficacité réelle de l'algorithme A.

C2. Deux exemples d'application

C2.1. Multiplication de matrices carrées

Soient $A=(a_{ij})$ et $B=(b_{ij})$ deux matrices n x n à coefficients dans R, on peut donc calculer la matrice produit C=A x B selon la formule classique $c_{ij} = \Sigma_{k=1..n}\ a_{ik}.b_{kj}$ et l'algorithme s'écrit pour les matrices d'entiers comme suit :

```
Entier[][] A=nouveau Entier[n][m];
Entier[][] B=nouveau Entier[m][n]) ;
Entier[][] C=nouveau Entier[n][n] ;

Entier i, j, k ; // Indices de boucles

POUR i=0 à n FAIRE
   POUR j=0 à n FAIRE
      C[i][j]=0 ;
      POUR k=0 à n FAIRE
          C[i][j]=C[i][j]+A[i][k]*B[k][j]
      FIN POUR
   FIN POUR
FIN POUR
```

Il est facile de voir que peu importe les données en entrée, les trois boucles sont exécutées. En considérant que le calcul des c_{ij} comme l'opération fondamentale, la complexité au pire cas devient

$\text{Max}(n)= \Sigma_{i=1..n}\ \Sigma_{j=1..n}$ (1 opération + $\Sigma_{k=1..n}$ 3 opérations) = n^2+3n^3 opérations (unités de temps)

C2.2. La recherche d'un élément dans une liste

Dans cet exemple, il est facile de déterminer la complexité au pire cas :
Max(n)=n (l'élément X n'est pas dans la liste et il faut tester jusqu'à la fin de la liste pour le savoir).

D. Ordres de grandeur : comparaisons d'algorithmes

Comme nous avons pu constater, les **complexités sont exprimées en fonction de la taille des données**. Or pour un problème donné, si la taille des données est faible, le choix d'un algorithme par rapport à un autre importe souvent peu. En revanche, lorsque **la taille des données devient grande**, il convient de **bien connaître la croissance des fonctions de complexité.** C'est comme les **études aux limites des fonctions mathématiques.**

D1. Approximations des complexités

Souvent, il suffit d'une approximation simple de la complexité pour savoir si un algorithme est plus efficace qu'un autre :

1) Dans les approximations, il n'est **pas utile de tenir compte des constantes additives**. Par exemple, il est secondaire de savoir si un algorithme fait du n opérations, du n+2 ou encore du n+10 opérations pour un n grand (i.e. pour n tendant vers l'infini).
2) **De même pour les constantes multiplicatives**, mais là, il convient d'être **plus prudent** au moment de l'approximation.

   Par exemple :
   - A1 a une complexité Max1(n)=n^2 et A2 avec Max2(n)=2n, A2 est donc meilleur que A1 pour tous les n>2. C'est sans discussion.
   - Mais si A1 a Max1(n)=$3n^2$ et A2 a Max2(n)=25n, alors A2 est donc meilleur que A1 que si n>8.

   Toutefois, peu importent les constantes multiplicatives, si M1(n)=k1.n^2 et M2(n)=k2.n, alors l'algorithme A2 est toujours meilleur que A1 à partir d'un certain rang pour n, car la fonction n^2 croît beaucoup plus vite que n : lim $_{n\to infini}$ $n/n^2 = 0$.

   On dit alors que **l'ordre de grandeur asymptotique** de n^2 est strictement plus grand que celui de n.

   Remarque : Toutes les notions mathématiques liées aux limites des fonctions s'appliquent ici.

D2. Ordre de grandeur asymptotique et notation « Landau »

Supposons que l'on ait à comparer A1 et A2 de complexités respectives de Max1(n) et Max2(n), si l'ordre de grandeur asymptotique de A1 est strictement plus grand celui de A2, on peut conclure immédiatement que A1 est meilleur que A2 pour n grand. En revanche, si Max1(n) et Max2(n) ont le même ordre de grandeur asymptotique, il convient de faire une analyse plus fine pour comparer A1 et A2.

Pour comparer les ordres de grandeur asymptotique, on a l'habitude d'utiliser la notation " Landau " en mathématiques.

Définition pour le pire cas :
Soient f et g deux fonctions de N dans R^+, **f=O(g)** si et seulement si il existe c dans R^+, il existe n0 dans N tel que quelque soit **n > n0, f(n) <= c.g(n)**.

Ainsi f=O(g) veut dire que f est **dominée asymptotiquement** par g.

Par exemples : 2n=O(n^2), mais 2n=O(n) aussi ! Il suffit de bien choisir la constante n0.

Remarques :
- Lorsqu'une complexité est donnée en terme de O(g), il convient de donner la fonction g la plus " serrée " possible. Donc 2n=O(n) est plus juste.
- En d'autres termes, O() est une borne asymptotique supérieure.

D3. Rappel des opérations en notation « Landau »

On suppose que n est la taille du problème, alors on a ces opérations pour des algorithmes ayant une complexité temporelle polynomiale en la taille du problème :
- $O(a)=O(1)$ si a est une constante par rapport à n.
- $O(a.n+b)=O(n)$ où a et b sont des constantes par rapport à n.
- $O(a.n^2+b.n+c)=O(n^2)$, cela se généralise à n^k.
- $O(n^{k1}).O(n^{k2})=O(n^{k1+k2})$ où k1 et k2 sont aussi des constantes par rapport à n.
- $O(a.n+b)+O(c.n+d)=O((a+c).n+(b+d))=O(n)$.
- Etc.

Remarque importante : Certains algorithmes ont des complexités temporelles exponentielles en $O(2^n)$ par exemple ! Vous pouvez imaginer ce que cela peut donner comme temps de calcul théorique lorsque nous augmentons simplement d'une unité la taille du problème en passant par exemple de 2^{50} à 2^{51}. Le temps de calcul double simplement à chaque unité de taille supplémentaire !
Prenons l'algorithme qui consiste à énumérer toutes les solutions possibles d'un vecteur binaire (0/1) de taille n. Il a typiquement cette complexité en $O(2^n)$. Pour peu que nous ayons un vecteur de taille 50 à énumérer, et supposons que nous ayons à notre disposons un ordinateur ultra puissant qui est capable d'énumérer 2^{20} solutions à la seconde, l'algorithme d'énumération nécessitera tout de même 2^{30} secondes (soit 34,048 années !!!!!) avant d'accomplir sa tâche et de s'arrêter.
Ceci vous montre la limite d'un algorithme lorsqu'il a une forte complexité (degré élevé du polynôme ou exponentielle, voire pire !). Malheureusement, pour beaucoup de problèmes d'optimisation combinatoire en production, localisation et transport, nous n'avons pas encore trouvé à ce jour des algorithmes polynomiaux pour les résoudre…

E. Trois algorithmes de tri classique

Les tris se retrouvent dans pratiquement 80% des applications informatiques. C'est pourquoi les algorithmes de tri ont long temps été et ils continuent à être au centre d'études de complexités temporelle et spatiale. Voir « Algorithmes de tri ou Tris » dans les livres d'algorithmique cités à la fin de cette fiche pour avoir une vision plus exhaustive. Nous présentons ici uniquement 3 : tri par insertion, tri rapide (quicksort en anglais) et tri fusion (merge sort en anglais).

E1. Tri par insertion

Ce tri que vous avez en 1er semestre consiste à simuler ce que peut faire un joueur de carte lorsque les cartes viennent juste de lui être distribuées. Ce joueur va alors, prendre les cartes devant lui, une à une. A la 1ère carte, il va juste la mettre dans sa main ; puis vienne la 2ème carte, il va l'insérer dans sa main devant ou après la première carte selon la valeur de cette

carte ; il va en faire de même avec la 3ème carte par rapport aux deux 1ères qu'il a déjà triées dans sa main ; et ainsi de suite. A la fin de processus d'« insertion » de cartes, le jouer aura un jeu de cartes trié dans sa main.

Cet algorithme de tri, illustré sur un tableau d'entiers à trier peut donc être écrit comme suit :

```
ALGO triInsertion
Entrées : Entier[] tab ; // on donne un tableau d'entiers à trier, les entiers sont déjà dans tab
Sortie : Entier[] ; // le tableau trié
-----------------------------------------------------------------------
// Le tri consiste à maintenir triés les i-1 premiers éléments du tableau
// A chaque itération sur i, l'élément tab[i] est inséré parmi les éléments triés tab[1]…tab[i-1]
// On recherche la position où l'insérer, et on décale les éléments suivants
  Entier i ; // Indice de boucle permettant le parcours de tous les éléments à trier du tableau
  Entier j ; // Indice de la position d'insertion de tab[i]
  POUR i=1 à tab.longueur-1 FAIRE  // on va examiner les entiers de tab un par un
       j=0 ;  // on va recherche la position dans le tableau où l'on peut insérer tab[i]
      TANT QUE ((j<i) ET (tab[j]<=tab[i]))
                  j=j+1,
      FAIN TANT QUE  // à la fin de cette boucle, j indique la position d'insertion
      Insertion(tab, i, j) ;  // on insère tab[i] en position j
  FIN POUR
  RETOURNER tab ;
FIN ALGO

ALGO Insertion
Entrées : Entier[] t ; Entier a, b ;  // a indice de l'élément à insérer, b la position d'insertion
Sortie : Rien ;
-----------------------------------------------------------------------
  Entier temp ; // variable temporaire
  Entier k ;  // indice de boucle de décalage

  temp=tab[a] ;  // sauvegarder tab[a] dans temp
  POUR k=a à b+1 FAIRE  // boucle de décalage vers la « droite » du tableau
          tab[k]=tab[k-1] ;
  FIN POUR
  tab[b]=temp;  // on met l'élément à la bonne position
FIN ALGO
```

Remarque : si ces algorithmes sont des méthodes d'une classe de tableau d'entiers triés, le tableau d'entiers ferait alors partie des attributs de la classe, et il n'y aurait plus besoin de le passer/récupérer en entrée/sortie des algorithmes.

E1.1. Exemple d'exécution

Supposons que nous avons le tableau [101, 115, 30, 63, 47, 20].
Itération 1 de i : [101,…] est trié au départ, et on regarde où on peut insérer 115.
Itération 2 de i : [101, 115, …] est trié, et on regarde où on peut insérer 30.
Itération 3 de i : [30, 101, 115, …] est trié, et on regarde où on peut insérer 63.
Itération 4 de i : [30, 63, 101, 115, …] est trié, et on regarde où on peut insérer 47.
Itération 5 de i : [30, 47, 63, 101, 115, …] est trié, et on regarde où on peut insérer 20.
Itération 6 de i : [20, 30, 47, 63, 101, 115] est trié, et l'algorithme se termine.

E1.2. Analyse de complexité temporelle au pire cas

Si nous regardons l'algorithme triInsertion et Insertion, nous pouvons constater deux opérations fondamentales : le test de comparaison tab[j]<=tab[i], et le décalage des éléments tab[k]=tab[k-1].

Sur le test de comparaison dans triInsertion, avec l'exemple d'exécution, nous pouvons constater que si le nombre d'éléments du tableau est de n, alors la boucle i va de 1 à n-1 et pour chaque itération de la boucle sur i, nous avons au pire cas i comparaisons de la boucle TANT QUE (**ATTENTION : la valeur de i est incrémentée de 1 à chaque boucle de i**).
De même, l'algorithme Insertion a une boucle de décalages allant de i à j+1 en décrémentant. Dans le pire cas, l'insertion se fait toujours en début du tableau, i.e. j vaut 0. Donc la boucle k effectue i décalages (tab[k]=tab[k-1]) dans le pire cas.

En somme, nous avons à la boucle i qui emboîte deux boucles, celle de j et celle de k. Or, celles de j et k ont une complexité temporelle en O(i) dans le pire cas, soit 2.O(i)=O(2.i)=O(i). Donc à chaque itération de la boucle i, nous avons O(i) opérations fondamentales (comparaisons et décalages) ; et i va de 1 à n-1. D'où la complexité au pire cas est égale à

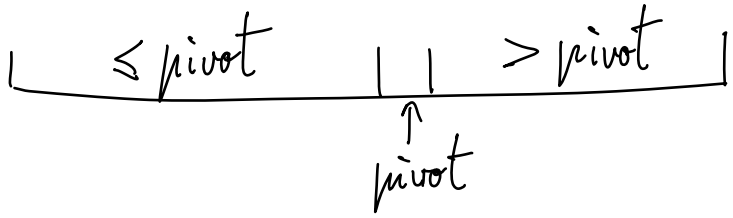
$$\text{Max(n)}=\sum_{i=1}^{n-1} O(i) = O(n.(n-1)/2) = O(n^2).$$

L'algorithme tri par insertion est donc au pire cas en $O(n^2)$.

Remarque : La complexité spatiale est évidente ici, nous n'avons utilisé qu'un tableau de n entiers, donc O(n) en espace.

E2. Tri rapide (Quicksort en anglais)

L'idée principale du tri rapide est fondée sur la notion de Diviser-pour-régner vue dans la fiche Récursivité et Diviser-pour-Régner. Le Diviser-pour-régner ici est la dichotomie de l'ensemble des éléments à chaque étape du tri. Mais plutôt que de partitionner l'ensemble en deux parties de même cardinalité, ici nous allons utiliser un **élément pivot**.



Dans le tri rapide, le pivot est **choisi arbitrairement comme étant le premier élément de l'ensemble à trier**, puis on place les autres éléments à gauche (<=pivot) et à droite (>pivot) du pivot.

Pour un tableau d'entiers à trier, l'algorithme récursif du tri rapide s'écrit :

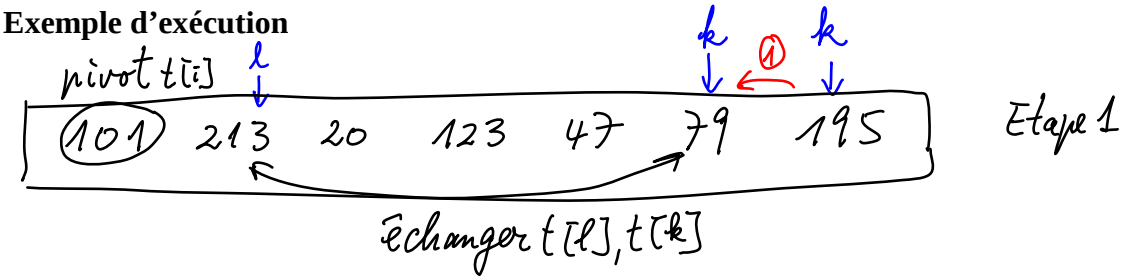
```
ALGO triRapide  // L'appel initial du tri rapide est : tab=triRapide(tab, 0, n-1) ;
Entrées : Entier[] tab ;  // tableau de n entiers non triés
          Entier i, j ;  // indices du début et fin des éléments du tableau à trier
Sortie : Entier[] ; // tableau de n entiers triés
```

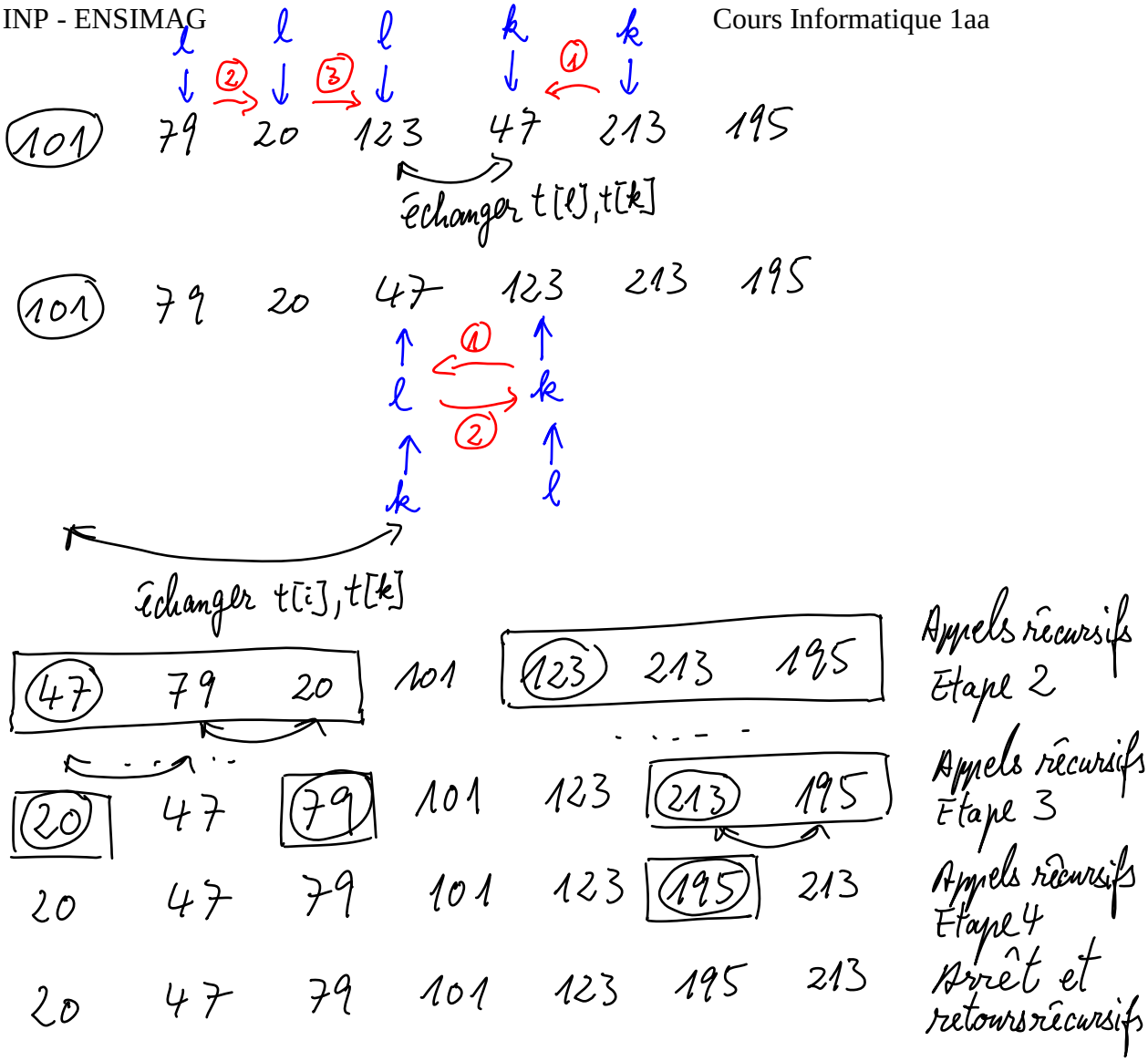
```
 Entier k ; // position du pivot dans le tableau tab
 SI (i<j) ALORS // on n'est pas encore arrivée à un seul entier à trier
        k=Placer(tab, i, j) ; // partitionner tab et placer tab[i] en k
        // puis trier récursivement les sous-tableaux gauche et droit du pivot tab[k]
        tab=triRapide(tab, i, k-1) ;   // trie partie gauche
        tab=triRapide(tab, k+1, j) ;  // trie partie droite
 FIN SI
 RETOURNER tab ;
FIN ALGO
```

```
ALGO Placer
Entrées : Entier[] t ; Entier i, j ;
Sortie : Entier ;  // Position k du pivot
```

```
 Entier k, l ;
 Entier temp ;
 l=i+1; k=j;
 TANT QUE (l<=k)
        // t[i] est arbitrairement considéré comme l'élément pivot
        // et on va échanger les éléments plus grands avec les éléments plus petite que t[i]
        TANT QUE (t[k]>t[i])  // rechercher le 1er t[k]<=t[i]
                k=k-1 ;
        FIN TANT QUE
        TANT QUE (t[l]<=t[i]) // rechercher le 1er t[l]>t[i]
                l=l+1 ;
        FIN TANT QUE
        SI (l<k) ALORS
                // échanger t[l] qui est plus grand que t[i] avec t[k] qui est plus petit que t[i]
                temp=t[l] ; t[l]=t[k] ; t[k]=temp ;
                l=l+1 ; k=k-1 ;
        FIN SI
        // Placer le pivot t[i] au « milieu » des deux parties du tableau t[k]
        temp=t[i] ; t[i]=t[k] ; t[k]=t[i] ;
 FIN TANT QUE
FIN ALGO
```

E2.1. Exemple d'exécution





Le tableau est trié !

E2.2. Analyse de complexité temporelle au pire cas

Pour l'algorithme Placer sur un tableau de taille i :

- Le nombre de comparaisons avec le nombre pivot : on compare i-1 éléments avec le pivot, d'où une complexité en O(i).
- Le nombre d'échanges est au pire 3.E(i/2)=O(i/2)=O(i) transferts, lorsque la valeur du pivot est « au milieu » de celles des autres nombres.

Pour l'algorithme triRapide, au pire cas, l'appel s'effectue sur une liste en ordre croissant déjà triée ! Nous obtenons alors les appels suivants :

```
triRapide(tab, 0, n-1) ;
triRapide(tab, 0, -1) ; triRapide(tab, 1, n-1) ;  // 1ers appels récursifs
                         triRapide(tab, 1, 0) ; triRapide(tab, 2, n-1) ; // 2èmes appels récursifs
                                                 …
                                                 triRapide(tab, n-2, n-3) ; triRapide(tab, n-1, n-1) ;
```

En nombre de comparaisons :

- Il n'y a aucune comparaison effectuée pour tous les appels triRapide(tab, i, i-1) (cas de base de l'appel récursif).
- Pour les triRapide(tab, i, n-1), on fait n+1 comparaisons, puis n, n-1, n-2, …, 3.
  D'où Max(n)= $\sum_{i=1}^{n-1} n+2-i = \frac{(n+2)(n+1)}{2} - 3 = O(n^2)$.

En nombre d'échanges : au pire cas, le pivot est toujours au milieu des autres éléments.

Max(n)= $\frac{n-1}{2} + (\frac{n-2}{2\times 2}).2 + (\frac{n-3}{2^3}).2.2 + ... = \sum_{i=1}^{n-1} \frac{(n-i)}{2} = \frac{1}{2}.\frac{n(n-1)}{2} = O(n^2)$.

L'algorithme tri rapide est donc au pire cas en $O(n^2)$.
Ici, la complexité spatiale est aussi en O(n).

Remarque : Il est démontré que le tri rapide possède une complexité temporelle en moyenne en $O(n.\log_2 n)$. La complexité en moyenne de ce tri est donc meilleure que celle du pire cas.

E3. Tri fusion (Merge sort en anglais)

A l'instar du tri rapide. Le tri fusion fait appel aussi à la notion de Diviser-pour-régner et la récursivité. En revanche, la dichotomie s'effectue vraiment au milieu de l'ensemble des éléments et non via un élément pivot.

Voici l'algorithme de tri fusion :

```
ALGO triFusion  // l'algo trie récursivement les parties gauche et droite du tableau
Entrées : Entier[] tab ;
Sortie : Entier[] ;
-----------------------------------------------------------------------------
 Entier[] Tgauche, Tdroit ; // deux tableaux de taille n1=n/2 et n2=n-n1, création dans Copier
 Entier n=tab.longueur ;

 SI (n>1) ALORS
       Tgauche=Copier(tab, 0, n/2-1) ;
       Tdroit    =Copier(tab, n/2, n-1) ;
       Tgauche=triFusion(Tgauche) ;
       Tdroit=triFusion(Tdroit) ;
       tab=Fusionner(Tgauche, Tdroit) ;
 FIN SI
 RETOURNER tab ;
FIN ALGO
```

ALGO Copier // Recopie les éléments de t entre les indices a et b dans un nouveau tableau
Entrées : Entier[] t ; Entier a, b ; // un tableau et deux indices
Sortie : Entier[] ; // un tableau des éléments de T compris entre les indices a et b

```
  Entier i, j ;
  Entier[] T ;
  T=nouveau Entier[b-a+1] ;

  j=0 ;
  POUR i=a à b FAIRE
        T[j]=t[i] ; j=j+1 ;
  FIN POUR
  RETOURNER T ;
FIN ALGO
```

ALGO Fusionner // fusionne 2 tableaux triés dans un nouveau tableau qui est ainsi trié
Entrées : Entier[] t1, t2 ; // de taille n1 et n2 respectivement
Sortie : Entier[] ; // tableau contenant les éléments de t1 et t2 triés

```
  Entier n1, n2 ;  // tailles de t1 et t2
  Entier i, i1, i2 ;  // indices sur T, t1 et t2
  Entier[] T ;

  n1=t1.longueur ; n2=t2.longueur ;
  T=nouveau Entier[n1+n2] ;
   i=0 ; i1=0 ; i2=0 ;
  TANT QUE ((i1<n1) ET (i2<n2))
        SI (t1[i1]<t2[i2]) ALORS
              T[i]=t1[i1]; i1=i1+1;
        SINON
              T[i]=t2[i2]; i2=i2+1;
        FIN SI
        i=i+1;
  FIN TANT QUE
  // Un des deux tableaux est totalement recopié dans T

  TANT QUE (i1<n1)
        T[i]=t1[i1] ;  // T est complété par les éléments de t1
        i1=i1+1 ; i=i+1 ;
  FIN TANT QUE
  TANT QUE (i2<n2)
        T[i]=t2[i2] ;  // T est complété par les éléments de t2
        i2=i2+1 ; i=i+1 ;
  FIN TANT QUE
  RETOURNER T;
FIN ALGO
```

F. Exercices

1. Dans les structures de données de type liste, quelles sont les complexités temporelles au pire cas pour le parcours d'une liste à n éléments, pour une liste avec tableau puis pour une liste avec des éléments chaînés ?
2. Même question pour l'opération d'insertion en tête de liste d'un élément, avec une liste implémentée avec un tableau, puis avec des éléments chaînés.
3. Exécuter l'algorithme triFusion sur le même tableau d'entiers que celui du tri rapide, et tracer toutes les étapes intermédiaires du triFusion.
4. Faites une étude des complexités temporelles au pire cas des algorithmes Copier, Fusionner et triFusion.
5. Comparer la complexité temporelle au pire cas de triFusion avec celle de triRapide. Qu'en concluez-vous ?

Quelques références bibliographiques utilisées

Livre (une des Bibles de l'algorithmique, lisez les 10-20 premières pages de chaque chapitre):
Introduction to Algorithms (Introduction à l'algorithmique).
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.
Edition: McGraw-Hill.

Livre (une autre Bible de l'algorithmique, mais pas facile à lire) :
Foundations of Computer Science
Alfred V. Aho, Jeffrey D. Ullman.
Téléchargeable en ligne
http://infolab.stanford.edu/~ullman/focs.html

Livre (ludique et facile d'approche):
Initiation à l'informatique
Henri-Pierre Charles
Edition : Eyrolles

Livre (vous avez tout sur Java et plus):
Core Java : Volume I – Fundamentals, 8th Edition
(Au cœur de Java : Volume 1 - Notions fondamentales, 8ème Ed.)
Cay S. Horstmann, Gary Cornell.
Edition: Sun Microsystems Press

Livre :
Programming with Java (Programmation en Java)
John R. Hubbard.
Edition: Schaum's Outlines

Et tout autre livre sur l'algorithmique et les structures de données…