

Initiation à l'algorithmique

Denis Lapoire

16 octobre 2007

Table des matières

1	Discours de la Méthode(extraits)	9
2	Quelques définitions et quelque syntaxe	11
2.1	Problèmes	11
2.2	Types abstraits	12
2.2.1	Quel type pour les entiers?	14
2.2.2	Un objet plus complexe : le tableau ou la matrice	14
2.3	Algorithme	15
2.3.1	Entête de l'algorithme	16
2.3.2	Le corps de l'algorithme	16
2.3.3	Variable	17
2.3.4	Évaluation d'expression	18
2.3.5	Initialisation	18
2.3.6	Modification de valeurs	18
2.3.7	Mise en séquence	19
2.3.8	Branchement conditionnel	19
2.3.9	Autres branchements conditionnels	19
2.3.10	Boucle	20
2.3.11	Autres boucles	22
2.3.12	Instruction <code>retourner</code>	23
2.3.13	Appel de fonctions	25
2.3.14	Passage par références	26
2.3.15	Indentation	28
2.4	Conclusion	30
3	Problèmes	31
3.1	Définition d'un problème	32
3.2	Premières solutions issues de la définition	32
3.3	Comparaison de problèmes	33
3.3.1	Augmenter les contraintes en entrée	33
3.3.2	Diminuer les contraintes en sortie	34
3.3.3	Décomposition en un ensemble équivalent de sous-problèmes	34

4	Terminaison et complexités	37
4.1	Terminaison	37
4.2	Complexité	38
4.3	Complexité d'une entrée	39
4.3.1	Complexité d'un booléen	39
4.3.2	Complexité d'un entier	39
4.3.3	Complexité d'une matrice d'entiers	40
4.4	Complexité d'un algorithme	40
4.4.1	Instruction élémentaire	40
4.4.2	Espace utilisée par l'algorithme	40
4.4.3	Complexité en temps dans le pire des cas	41
4.4.4	Complexité en temps en moyenne	42
4.4.5	Complexité en temps dans le meilleur des cas	42
4.4.6	Complexité en espace	42
4.5	Complexité d'un problème	42
4.5.1	Compromis espace-temps	43
4.6	Notations et simplifications	43
4.6.1	À une constante multiplicative près : notation Θ	43
4.6.2	Des fonctions étalons	45
4.6.3	Simple majoration : notation O	46
4.6.4	Quelques règles d'évaluation de complexité	47
4.6.5	Mise en séquence	47
4.6.6	Branchement conditionnel	48
4.7	Un peu de vocabulaire	48
4.7.1	Un algorithme linéaire	48
4.7.2	Un algorithme exponentiel	49
4.8	Étude du problème de puissance	50
4.8.1	Premiers algorithmes triviaux, simples mais lents	50
4.8.2	Analogie avec le produit	50
4.8.3	Premier algorithme rapide	51
4.8.4	Second algorithme rapide	52
4.8.5	Analyse des complexités : l'importance des hypothèses	53
5	Algorithmes "Diviser Pour régner"	57
5.1	Un premier exemple : la multiplication de deux entiers	57
5.1.1	Complexité au moins linéaire	58
5.1.2	Un algorithme existant quadratique	58
5.1.3	Une amélioration "Divide And Conquer" entre linéaire et quadratique	58
5.1.4	Conclusion	60
5.2	Évaluation de la complexité	60
5.2.1	Définition récursive de la fonction	60
5.2.2	Résolution de certaines fonctions $\mathbb{N} \rightarrow \mathbb{N}$ définies récursivement	61

5.3	Un deuxième exemple : la multiplication de deux matrices	62
5.3.1	Première méthode	62
5.3.2	Seconde méthode dite de Strassen	63
5.3.3	Conclusion	63
6	Quelques éléments de Programmation Dynamique	65
6.1	Une solution en programmation dynamique	66
6.1.1	Autre alternative	68
6.2	Bien fondé de l'approche dynamique	68
7	Algorithme Glouton	71
7.1	Un premier exemple : le rendu de monnaie	71
7.1.1	Correction	72
7.1.2	Amélioration	72
7.2	Deuxième exemple : optimisation de la location d'une salle	73
7.2.1	Première tentative	73
7.2.2	Deuxième tentative	73
7.2.3	Troisième tentative concluante	74
7.3	Conclusion	75
8	Quelques algorithmes de tri	77
8.1	Approche gloutonne	77
8.1.1	Une première implémentation	78
8.1.2	Une deuxième implémentation à l'aide d'un tas	79
8.2	Approche "Diviser pour régner"	79
8.2.1	Première solution algorithmique de TriRec	80
8.2.2	Seconde solution algorithmique de TriRec	82
8.2.3	Différentes variantes liées au choix du pivot	84
8.2.4	Le choix du premier pivot venu : le tri rapide	84
8.2.5	Le choix d'un pivot aléatoire	86
8.2.6	Le choix d'un pivot médian	86

Ce cours est dédié aux étudiants de 1ère année de l'Enseirb spécialité informatique. Son intitulé "Initiation à l'algorithmique" est d'un certain point de vue un contresens. Tous les étudiants de l'Enseirb bénéficient d'une solide culture mathématique et ont donc nécessairement de larges connaissances en algorithmes. En effet, la quasi totalité des objets mathématiques qu'ils ont rencontré depuis leur plus tendre âge ont été définis de façon calculatoire.

L'objet de ce cours est donc de s'appuyer sur ces connaissances et ce qu'elles recèlent d'intuition pour présenter différentes méthodes générales pour fournir à un problème une ou plusieurs solutions algorithmiques.

Nous insisterons sur la nécessité pour résoudre un problème de le découper en sous-problèmes auxiliaires, de résoudre ainsi le premier problème avant même d'avoir résolu ces problèmes auxiliaires. Cette approche descendante que nous suggère Descartes (Chapitre 1) est présentée dans le Chapitre 3.

Ce principe universel rappelé, nous présenterons des méthodes de résolution de problèmes qui ont pour nom :

- l'approche "Diviser pour Régner".
- la Programmation Dynamique.
- l'approche gloutonne.

Tout au long de ce cours, nous insisterons sur différentes qualités attendues des algorithmes écrits :

1. la correction naturellement qui assure que l'algorithme répond bien au problème.
2. la simplicité d'écriture qui met en valeur notamment les différentes étapes de calcul en fournissant une vision structurée et donc simple de l'algorithme.
3. la faible consommation de deux ressources que sont le temps et l'espace.
En d'autres termes, nous évaluerons la complexité en temps et en espace de ces algorithmes.

Les deux derniers principes sont très souvent contradictoires. Un algorithme très rapide en temps s'écrit de façon plus complexe. Cette opposition apparaît dans l'écriture récursive : dans de très nombreux cas, celle-ci offre une définition très simple mais utilise des ressources en espace non constant, alors qu'une version itérative plus compliquée à définir utilise moins d'espace. Pour cette raison, nous

aurons soin pour certains problèmes d'écrire plusieurs solutions algorithmiques, chacune ayant une qualité singulière.

La résolution du problème *Tri* conclura ce cours en illustrant l'intérêt des méthodes présentées plus haut. Ce cours n'est pas un cours d'algorithmique et structures de données. Il ne contiendra pas notamment les classiques notions de Pile, Liste et File, d'arbres ou de graphes, ni leurs définitions, ni à fortiori leurs implémentations. Les exemples de problèmes concernant auant que faire ce peut des objets simples comme les booléens, les entiers ou des objets déjà très familiers à tous les étudiants comme les tableaux ou matrices.

Nous pouvons conseiller quelques documents :

1. Ce document accessible à <http://www.enseirb.fr/~lapoire/1ereAnnee/InitiationAlgorithme/Cours/>. Nous rappelons que ce document est disponible sous forme papier. Il n'est donc pas utile de l'imprimer. Les remarques, corrections éventuelles sont les bienvenues et sont sollicitées à lapoire@enseirb.fr.
2. "Introduction à l'algorithmique" de Cormen and Co. Collection Dunod. Cet ouvrage est présent à la bibliothèque de l'Enseirb. Il est complet, long(plus de 1000 pages) et couvre l'ensemble des notions vues dans ce cours dans de nombreux autres cours d'Informatique de l'Enseirb. Les chapitres concernés par ce cours sont les chapitres 1, 2, 3, 4, 6, 7, 15, 16 et 28.

Chapitre 1

Discours de la Méthode(extraits)

“ Au lieu de ce grand nombre de préceptes dont la logique est composée, je crus que j’aurois assez des quatre suivants, pourvu que je prisse une ferme et constante résolution de ne manquer pas une seule fois à les observer.

Le premier était de ne recevoir jamais aucune chose pour vraie que je ne la connusse évidemment être telle ; c’est-à-dire, d’éviter soigneusement la précipitation et la prévention, et de ne comprendre rien de plus en mes jugements que ce qui se présenterait si clairement et si distinctement à mon esprit, que je n’eusse aucune occasion de le mettre en doute.

Le second, de diviser chacune des difficultés que j’examinerais, en autant de parcelles qu’il se pourrait, et qu’il serait requis pour les mieux résoudre.

Le troisième, de conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu à peu comme par degrés jusque à la connaissance des plus composés, et supposant même de l’ordre entre ceux qui ne se précèdent point naturellement les uns les autres.

Et le dernier, de faire partout des dénombrements si entiers et des revues si générales, que je fusse assuré de ne rien omettre. ”

Descartes, *Discours de la Méthode*(1637)

Chapitre 2

Quelques définitions et quelque syntaxe

Dans ce chapitre, nous présentons le pourquoi d'un algorithme : sa raison d'être est de résoudre un problème.

Nous présenterons en outre un langage de description algorithmique. Celui-ci est propre à ce cours. Il a été choisi pour des raisons de simplicité et de proximité avec d'autres langages comme le langage C.

2.1 Problèmes

La raison d'être d'un algorithme est de résoudre un problème. La plus grande attention doit être portée à la compréhension du problème, faute de quoi l'algorithme n'a aucune chance d'être correct.

Le langage utilisé pour la définition d'un problème est un langage scientifique utilisant pour des raisons de simplicité une langue naturelle (le français par exemple) ou, pour lever toute ambiguïté, un langage logique. La compréhension du problème nécessite la maîtrise de ce langage.

Un problème est une relation binaire liant deux objets ou deux séquences d'objets de types fixés. Un exemple de problème est :

problème Puissance

Entrée : un réel $x \neq 0$, un entier n

Sortie : le réel x^n

Cette relation n'est pas nécessairement fonctionnelle, à toute entrée correspond une unique sortie. Le problème suivant associe à une entrée par exemple 6 deux sorties possibles 2 et 3 dans l'exemple.

problème FacteurPremier

Entrée : un entier $n > 1$

Sortie : un facteur propre premier de n si il en existe,
0 sinon.

Problèmes typés

La définition d'un problème nécessite un typage constant des objets qui constituent les entrées et les sorties. Ainsi, la relation :

Entrée : un entier $n > 1$

Sortie : un entier facteur propre de n si il en existe,
le booléen faux sinon

n'est pas correctement typée. Une définition correcte peut être d'étendre à l'entier retourné le rôle du booléen comme ici :

Entrée : un entier $n > 1$

Sortie : un entier facteur propre de n si il en existe,
l'entier 0 sinon.

Cet artifice ici est possible car on peut faire jouer à l'entier 0 le rôle du booléen faux() puisqu'il n'est jamais un facteur propre premier. Ce type d'astuce est très souvent à éviter car il ajoute de la confusion et provoque des erreurs. Une solution robuste est la suivante qui retourne deux objets, l'un entier, l'autre booléen aux rôles bien distincts :

Entrée : un entier $n > 1$

Sortie : le booléen vrai() et un facteur premier de n si il en existe,
le booléen faux() et un entier quelconque sinon.

2.2 Types abstraits

Un type se définit par un nom, un domaine et un ensemble d'opérations typés. Les types que nous rencontrerons dans ce cours sont limités à des objets d'aspect assez familier que sont les booléens, les entiers et les tableaux ou matrices de ceux-ci. Dans un prochain cours, nous reviendrons plus largement sur cette notion de type.

Le type booléen

Le type le plus élémentaire est sans contestation possible le type de nom booléen, dont le domaine est composé des deux seuls éléments vrai et faux et défini par les cinq opérations :

1. vrai() qui retourne le booléen vrai.
2. faux() qui retourne le booléen faux.
3. \wedge qui associe aux deux booléens vrai le booléen vrai et faux sinon. Cette opération peut se noter **et**.
4. \vee qui associe aux deux booléens faux le booléen faux et vrai sinon. Cette opération peut se noter **ou**.

5. \neg qui associe à **vrai** la valeur **faux** et inversement. Cette opération peut se noter **non**.

Exercice 1 Définir les opérateurs booléens \Rightarrow , \Leftarrow et \Leftrightarrow à l'aide des opérateurs booléens ci-dessus.

Pourquoi un type abstrait

Excepté le type booléen ou plus généralement tout type à domaine fini, la définition d'un type à des fins de calculs pose des difficultés.

Supposons que l'on veuille définir un type réel et définir quelques opérateurs et comparateurs bien utiles comme par exemple $+$, $.$, inv , $\sqrt{}$, \leq . Dans le monde réel (concret), deux solutions s'offrent à nous :

1. Un premier monde concret : on représente tout réel r à l'aide d'un autre réel l'approchant.

Le type `float` dit "à virgule flottante 32 bits IEEE" code un réel r sur 4 octets en approchant tout réel par un triplet composé d'un bit de signe, d'un exposant sur 8 bits et d'une "mantisse" sur 23 bits. Les résultats sont toujours faux, selon une approximation qui dépend de la qualité des fonctions fournies.

2. Un second monde concret, le calcul formel, qui représente tout réel non à l'aide d'une expression décimale avec puissance mais à l'aide d'une expression.

Ainsi, `inv(pi())` représente l'inverse de π . Les calculs réalisés dépendent naturellement de l'état des connaissances implémentées ou non qui permettent ce calcul formel.

Ces deux solutions ont pour désavantage l'une de fournir des résultats (presque) toujours erronés, et l'autre de ne pouvoir une réponse que dans un nombre limité de cas.

Pour cette raison, pour résoudre un problème algorithmique concret, nous décomposerons celui-ci en deux sous-problèmes.

1. Un premier où l'on fournit une solution algorithmique en supposant la possession d'un type parfait dit "abstrait". Par exemple : un type réel qui stocke chaque réel dans un espace mémoire constant $\Theta(1)$, qui multiplie deux réels en temps constant $\Theta(1)$ et ce sans erreur.
2. Un second qui consiste à étudier, en l'aménageant ou non, l'algorithme précédent en considérant son exécution sur un type réel concret.

Notre enseignement ne sera concerné que par le premier aspect. Le second aspect nécessite de connaître le contexte précis de résolution des problèmes et sera abordé dans d'autres cours.

En clair, le type abstrait permet d'oublier ou de s'accommoder, dans un premier temps, de la petitesse de notre monde réel et de simplifier le problème en le situant dans un monde plus abstrait.

2.2.1 Quel type pour les entiers ?

Voici quelques types d'entiers que nous aurons à considérer :

Un type entier concret : le type `int`

Ce type fourni par le langage `C` manipule des entiers codés sur 4 (parfois 2) octets (taille constante $\Theta(1)$) et réalise les opérations courantes en temps constant. Il ne permet ainsi de ne représenter que 2^{32} entiers compris entre -2^{31} et $2^{31} - 1$.

Un type entier concret : le type `entierLong`

Ce type permet de manipuler des entiers n sur un espace mémoire non constant égal à $\log_2(n)$ bits et réalise les opérations courantes selon une complexité en temps très souvent non constante (par exemple, $\log_2(n)$ pour l'addition).

Il nous arrivera ou de considérer le type `entierLong` précédent ou le plus souvent de considérer le type abstrait suivant :

Un type entier abstrait : le type `entier`

Ce type suppose de pouvoir manipuler des entiers utilisant un espace constant de représentation au travers d'opérations ($+$, \cdot , etc) supposées s'exécuter en temps constant.

2.2.2 Un objet plus complexe : le tableau ou la matrice

Nous aurons à manipuler des ensembles ou séquences d'objets. Pour cela, nous utiliserons des tableaux. Le type abstrait `tableau` utilisé est choisi pour des raisons de simplicités. Il permet de :

1. de construire un tableau. Il faut alors fournir sa longueur, et la valeur d'initiale commune à tous les éléments.
Ainsi `constructTab(10)(100)` retourne un tableau de longueur 10 de valeurs entières toutes égales à 100. Le tableau possède pour indices les 100 entiers de 1 à 100.
2. de calculer la longueur d'un tableau.
Ainsi, l'expression `longueur(T)` a pour valeur la longueur de `T`.
3. de calculer ou de modifier la valeur d'un tableau T à un indice i fourni. Nous utiliserons alors l'expression déjà familière `T[i]`. Ainsi, `T[4] <- 77` remplace la 4^{ème} du tableau par la valeur entière 77.

Exercice 2 Quel quadruplet d'entiers retourne la fonction suivante :

```

fonction exemple():(entier,entier,entier,entier)

début
  T ← constructTableau(10)(100) ;
  T[5] ← T[5] · T[7] ;
  T[longueur(T)] ← 66 ;

  retourner (T[1],T[5],longueur(T),T[10])
fin

```

Remarque 1 L'écriture de certains algorithmes peut être parfois simplifiée en choisissant un premier indice autre que l'entier 1 par exemple 0 ou -10 ou en choisissant pour indices d'autres objets que les entiers par exemple des caractères. Si cela est nécessaire, vous pouvez considérer un autre type abstrait que le type tableau plus haut. Pour cela, vous devez :

1. justifier que ce nouveau choix crée des simplifications.
2. expliquer que ce nouveau type peut être défini. C'est le cas par exemple d'un tableau dont les indices sont les 26 caractères ordonnés selon 'a', ..., 'z'. C'est à dire d'un ensemble d'indices mis en bijection avec un intervalle d'entiers de la forme $[1, n]$ avec $n \geq 0$.

2.3 Algorithme

Un exemple d'algorithme résolvant Puissance est :

```

fonction puissance( x : réel ; n : entier) : réel

début
  res ← 1.0 ;

  faire n fois
    res ← res · x
  ;
  retourner res
fin

```

FIG. 2.1 – L'algorithme puissance résolvant Puissance

Comme nous pouvons l'observer sur l'exemple de l'algorithme puissance, un algorithme est un texte composé de deux parties :

1. une partie entête, appelée aussi prototype. Dans l'exemple :
fonction puissance(x : réel ; n : entier) : réel.
2. le corps de l'algorithme qui fournit sa définition. Dans l'exemple : début fin.

2.3.1 Entête de l'algorithme

L'entête d'un algorithme fournit 4 informations :

1. la première est la nature de l'algorithme et est précisée par le mot-clef **fonction** ou **procédure**.

Une fonction est un algorithme retournant un ou plusieurs objets typés : il utilise la fonction **retourner**. Et ce, contrairement à la procédure. Le calcul d'une procédure est sauvegardée en mémoire grâce à un effet de bord décrit plus loin.

2. le second est le nom de l'algorithme ; dans l'exemple : **puissance**.

Une grande importance doit être consacré à la définition de ce nom. Plusieurs méthodes existent quant à la définition de ce nom.

Une minimale définit le nom simplement à partir du simple problème à résoudre (exemple : fonction **puissance**). On peut aussi indiquer dans le nom, la méthode utilisée, voire le type des objets passées en entrée ou en sortie. Ces choix sont primordiaux dans l'écriture de longs programmes nécessitant l'écriture d'un grand nombre de fonctions. Les exemples que nous présenterons sont relativement simples et concis et donc ne nécessitent pas cet effort.

3. le troisième est le nom et le type des arguments en entrée ; dans l'exemple : (**x** : réel ; **n** : entier).

Les différents types sont ceux spécifiés par le problème.

4. la quatrième est le type (ou les types) de l'objet retourné par la fonction (dans l'exemple **réel**).

Les types retournés sont ceux spécifiés par le problème à résoudre.

En conclusion, excepté le nom de l'algorithme et le nom des arguments en entrée, le prototype est entièrement dépendant de la définition du problème à résoudre et s'écrit donc sans difficulté (si le problème est clairement défini!).

2.3.2 Le corps de l'algorithme

Nous allons définir ici une syntaxe permettant d'écrire tout algorithme. Cette syntaxe est propre à ce cours. Elle n'a pas vocation à permettre la compilation ou l'exécution de l'algorithme sur une machine. Son ambition est de fournir selon une syntaxe simple des algorithmes simples et non ambigus qui pourront être traduits, avec un minimum d'effort, dans un langage de programmation comme le Langage C.

2.3.3 Variable

La quasi-totalité des problèmes nécessitent pour leur résolution un espace mémoire auxiliaire sur lequel sont exécutés des calculs intermédiaires. Pour gérer de façon simple et lisible cette espace mémoire, on utilise des “variables”.

Ainsi, derrière toute variable, se trouve un espace mémoire dont on peut modifier le contenu. De façon plus abstraite et donc plus formelle, une variable possède plusieurs attributs :

1. un identificateur.

Ce nom dont elle ne change pas devra être choisi de façon judicieuse pour rendre l’algorithme compréhensible. Naturellement, il ne peut pas être choisi parmi les mots clefs `début`, `tantque`, etc.

2. un type.

La variable possède à sa création un type dont elle ne peut pas changer. Notons que dans d’autres langages, d’autres choix existent.

3. une valeur.

Dès sa création, la variable possède une valeur de, naturellement, même type. Cette valeur peut naturellement varier : d’où le nom de variable.

4. une durée de vie.

Dans un objectif de simplification, une variable existe uniquement au cours d’un appel de fonction et cesse à la fin de cet appel de fonction, donc lors de l’exécution de l’instruction `retour`.

Exemple 1 Voici un second algorithme qui nous permettra d’illustrer comment créer une variable et comment modifier la valeur d’une variable à l’aide de l’opération d’affectation \leftarrow .

```

fonction exemple1():entier
début
  i ← 4;
  r ← 4.0;
  T ← constructTab(10)(100)(1);

  i ← 5;
  j ← i;
  i ← i+5;
  T[i] ← 2·j + 2 ;

  retourner i ;
fin

```

FIG. 2.2 – Un exemple d’algorithme

2.3.4 Évaluation d'expression

Tout programme contient des expressions typées. Une expression typée est une expression bien formée vérifiant la syntaxe du type considéré.

Par exemple `2.j + 2` est une expression de type **entier**.

Puisque à tout moment de l'exécution d'un algorithme chaque variable possède une valeur, à tout moment on peut *évaluer* une expression, c'est à dire lui associer une valeur.

Dans l'exemple précédent, avant exécution de `T[i]<-2.j + 2 ;`, la variable `j` a pour valeur 5, aussi l'évaluation de l'expression `2.j + 2` fournit la valeur 12.

Ces expressions peuvent naturellement être construites à partir de fonctions. Ainsi `(exemple1()+20.exemple1())` est une expression de type **entier** construite à partir de la fonction `exemple1` qui retourne à chaque appel la valeur entière 10. L'évaluation de `(exemple1()+20.exemple1())` fournit la valeur entière 210.

2.3.5 Initialisation

Pour créer une variable, il suffit d'utiliser l'opération affectation `<-` et de préciser à droite de ce symbole sa première valeur.

Ainsi `i<-4` crée une variable de nom `i`, de type **entier** puisque 4 est de type **entier** et de valeur 4.

Ainsi `r<-4.0` crée une variable de nom `r`, de type **réel** puisque 4.0 est de type **réel** et de valeur 4.0.

Ainsi `j<-i` crée une variable de nom `j`, de type **entier** puisque `i` est de ce type et de valeur celle de `i` à savoir 4. Notons que les variables `i` et `j` sont distinctes, ainsi la prochaine modification de `i` n'entraînera pas de modification de `j`.

En ce qui concerne le type tableau, vous pouvez utiliser un constructeur de tableau. Ainsi, `T<-constructTab(10)(100)(1)` crée une variable `T`, de nom `T`, de type **tableau** et désignant un tableau de taille 10, à éléments tous égaux à 100 et de premier indice 1.

2.3.6 Modification de valeurs

Une fois la variable créée, l'utilisation de l'affectation `<-` permet de modifier la valeur de la variable. Dans le programme `exemple1`, `i<-5` remplace l'ancienne valeur 4 par la valeur 5.

En ce qui concerne les tableaux, seules les modifications élémentaires sont autorisées. Ainsi, `T[i]<- j` modifie la 5ème case du tableau `T` de valeur initiale 100 par la valeur de l'expression `2.j+2` à savoir 12.

2.3.7 Mise en séquence

La première possibilité pour structurer un ensemble d'instructions est leur mise en séquence.

La syntaxe est :

```
début
  <instruction 1>
  ;
  <instruction 2>
  ;
  ....
  <instruction n>
fin
```

L'exécution d'une telle séquence est réalisée en exécutant d'abord l'instruction <instruction 1>, puis l'instruction <instruction 2> jusqu'à l'instruction <instruction n>.

2.3.8 Branchement conditionnel

Une deuxième possibilité pour structurer un ensemble d'instructions est l'utilisation du branchement conditionnel `si alors sinon`.

La syntaxe est :

```
si <expression booléenne> alors
  <instruction 1>
sinon
  <instruction 2>
```

L'exécution d'une telle instruction consiste à évaluer l'expression <expression booléenne> (qui doit nécessairement être de type booléen!) c'est à dire lui associer ou le booléen `vrai` ou `faux` puis ensuite, si l'évaluation fournit `vrai` exécuter <instruction 1>, et sinon exécuter <instruction 2>.

Exemple 2 La fonction `maximum` de la figure 2.3 associe comme valeur à la variable `maxi` la valeur de la variable `a` si l'expression `a>b` est vraie ou sinon la valeur de la variable `b`. Clairement, `maxi` prend pour valeur le maximum des valeurs de `a` et `b`. La fonction `maximum` permet bien de calculer le maximum des deux paramètres passés en entrée.

2.3.9 Autres branchements conditionnels

Notons l'existence d'autres branchements conditionnels. Par exemple, `si alors` dont la syntaxe

```

fonction maximum(a,b : entier) : entier
début
  si a > b
    maxi ← a
  sinon
    maxi ← b
  ;
  retourner maxi
fin

```

FIG. 2.3 – Algorithme de calcul du maximum

```

si <expression booléenne> alors
  <instruction>
est équivalent à
si <expression booléenne> alors
  <instruction 1>
sinon
  ; % ne rien faire

```

2.3.10 Boucle

Une troisième possibilité pour structurer un ensemble d'instructions est l'utilisation de la boucle `tantque`.

La syntaxe est :

```

tantque <expression booléenne> faire
  <instruction>

```

Une définition formelle de cette boucle peut être faite en utilisant le branchement conditionnel `si alors` : l'instruction

```

tantque <expression booléenne> faire
  <instruction>

```

est équivalente au sous-programme (infini!) suivant :

```

si <expression booléenne> alors
début
  <instruction> ;
  si <expression booléenne> alors
  début
    <instruction> ;
    si <expression booléenne> alors
    début
      <instruction> ;

```

```

    si <expression booléenne> alors
    début
        <instruction> ;

        ETC

    fin
  fin
fin

```

Cette définition peut créer une certaine gêne : un sous-programme infini peut être difficile à concevoir. Aussi, nous illustrerons la définition du **tantque** à l'aide de l'exemple suivant :

Exemple 3 L'exécution de

```

i ← 31 ;

tantque (i>28)
  i ← i - 2 ;

<instruction suivante> ;

```

consiste à associer à *i* la valeur 31 puis à :

- évaluer l'expression *i>28* qui fournit le booléen **vrai** (car $31 > 28$) et donc à exécuter *i←i-2*; la valeur de *i* est alors 29;
- évaluer l'expression *i>28* qui fournit le booléen **vrai** (car $29 > 28$ est vrai) et donc à exécuter *i←i-2*; la valeur de *i* est alors 27;
- évaluer l'expression *i>28* qui fournit le booléen **vrai** (car $27 > 28$ est faux); l'instruction **tantque** est alors finie d'être exécutée; on exécute alors l'instruction suivante.

La structure de contrôle **tantque** est très puissante. Elle permet de mettre en séquence un nombre aussi grand qu'on le souhaite d'une même instruction, voire même d'un nombre infini, comme le montre l'exemple suivant :

Exemple 4 L'exécution de

```

i ← 31 ;

tantque (i<>28)                % <> signifie ≠
  i ← i - 2

```

consiste à associer à *i* la valeur 31 puis à :

- évaluer l'expression *i<>28* qui fournit le booléen **vrai** (car $31 \neq 28$ est vrai) et donc à exécuter *i←i-2*; la valeur de *i* est alors 29;
- évaluer l'expression *i<>28* qui fournit le booléen **vrai** (car $29 \neq 28$ est vrai) et donc à exécuter *i←i-2*; la valeur de *i* est alors 27;

- évaluer l’expression $i < 28$ qui fournit le booléen *vrai* (car $27 \neq 28$ est vrai) et donc à exécuter $i \leftarrow i - 2$; la valeur de i est alors 25;
- et ainsi de suite ...

La boucle ne finira jamais d’être exécutée. Nous entrons dans ce que nous appelons une “boucle infinie”. Le programme dans lequel serait exécutée une telle instruction ne retourne aucune valeur car ne termine pas.

Les programmes que vous devrez écrire devront terminer. La question cruciale de la terminaison sera abordée dans le chapitre suivant.

2.3.11 Autres boucles

D’autres boucles existent. Citons en trois (cette liste n’est pas exhaustive et sera enrichie en cours ou en TD) :

1. la boucle de syntaxe

```
faire <expr. entière> fois
  <instruction>
```

Elle est équivalente à :

```
indice ← <expr. entière> ;
tantque indice > 0 faire
  début
    <instruction> ;
    indice ← indice - 1
  fin
```

2. la boucle de syntaxe

```
pour <variable entière> de <expr. entière1> à <expr. entière2>
  <instruction>
```

Elle est équivalente à :

```
<variable entière> ← <expr. entière1> ;
tantque (<variable entière> ≤ <expr. entière2>) faire
  début
    <instruction> ;
    indice ← indice + 1
  fin
```

3. la boucle de syntaxe

```
faire
  <instruction>
jusqu’à
  <expression booléenne>
```

Elle est équivalente à :

```

début
  <instruction>;

  tantque non(<expression booléenne>) faire
    <instruction>
fin

```

Par leur définition même, ces trois boucles n'augmentent pas la pouvoir expressif du langage algorithmique : les boucles *tantque* suffisent. Quel est donc leur intérêt ? Elles permettent d'écrire des algorithmes de compréhension plus simple. Propriété que nous attendons de tous les algorithmes que vous écrirez.

2.3.12 Instruction retourner

L'exécution de toute fonction doit se terminer par l'exécution de la fonction `retourner` ayant pour argument un de type égal au type retourné par la fonction. Lors de l'exécution de l'instruction `retourner <expression>`, l'expression est évaluée, cette valeur est alors retournée, l'exécution de la fonction est finie.

Ainsi, la fonction

```
fonction suivant(n : entier):entier
```

```
  retourner n+1
```

permet de calculer l'entier qui succède à l'entier n .

Certaines syntaxes imposent que toute fonction ne possède qu'une instruction `retourner` et que celle ci soit à la fin. Pour des raisons de simplicité, non de syntaxe mais d'écriture, nous accepterons le contraire, si cela se justifie comme dans l'exemple suivant :

Exemple 5 Considérons le problème qui consiste à décider si un entier a est présent dans un tableau d'entiers :

problème Recherche

Entrée : un entier a , un tableau d'entiers T

Sortie : vrai si il existe i tel que $i \in [1, \text{longueur}(T)]$ et $a=T[i]$

faux sinon

Une première solution est fournie par l'algorithme `recherche1`.

L'algorithme `recherche2` de la Figure 2.4 fournit une seconde solution ; elle permet de quitter la boucle dès que l'on a trouvé le bon indice et utilise pour cela une boucle *tantque*.

Les deux solutions ont chacune un avantage : la première est de définition plus simple alors que la seconde est (un peu) plus rapide mais de définition plus complexe. Une solution aussi simple que la première et aussi rapide que la seconde est fourni par l'algorithme `recherche3` (Figure 2.5) ; il utilise `retourner` à l'intérieur de la boucle *tantque*.

```
fonction recherche1(a : entier ; T : tableau d'entiers) : booléen
```

```
début
```

```
  res ← faux() ;
```

```
  pour i de 1 à longueur(T)
```

```
    si (a = T[i]) alors
```

```
      res ← vrai()
```

```
  ;
```

```
  retourner res
```

```
fin
```

```
fonction recherche2(a : entier ; T : tableau d'entiers) : booléen
```

```
début
```

```
  i ← 1 ;
```

```
  tant que (a <> T[i] ET i < longueur(T)) faire
```

```
    i ← i + 1
```

```
  ;
```

```
  retourner ( a = T[i])
```

```
fin
```

FIG. 2.4 – Un second algorithme de recherche

Exercice 3 Ecrire un algorithme solution de Recherche obtenu à partir de `recherche2` en remplaçant la boucle `tantque` par une boucle `JusquA`. Vous pourrez dans un premier temps réécrire l'algorithme de façon automatique en utilisant la règle de réécriture définissant la boucle `JusquA` à partir de la boucle `tantque`. Dans un second temps, vous modifierez l'algorithme ainsi obtenu en le rendant plus simple et plus compréhensible.

Exercice 4 Expliquer pourquoi la mise en séquences des instructions `retourner a` ; `retourner b` est équivalente à l'instruction `retourner a`. Equivalent signifie ici que l'on peut supprimer dans tout algorithme l'instruction `retourner b` sans modifier une quelconque exécution de celui-ci.

Exercice 5 Une *permutation* est ici un tableau indicés de 1 à n contenant une et une seule fois chacun des n entiers de 1 à n . La permutation *inverse* d'une permutation T de longueur n est le tableau U de même longueur vérifiant pour tout $i \in [1, n]$ l'égalité $T[U[i]] = i$.

1. Fournir un exemple d'une permutation de longueur 6 et de son inverse.
2. Fournir une seconde définition équivalente de l'inverse d'une permutation.


```

fonction recherche3(a : entier ; T : tableau d'entiers) : booléen
début
  pour i de 1 à longueur(T)
    si (a = T[i]) alors
      retourner vrai()
  ;
  retourner faux()
fin

```

FIG. 2.5 – Un troisième algorithme de recherche

3. Ecrire un algorithme résolvant ce problème.

2.3.13 Appel de fonctions

Nous pouvons bien entendu utiliser une fonction déjà définie pour en définir une seconde. Dans l'exemple de la Figure 2.6, la fonction `tata` utilise pour sa définition la fonction `toto`.

```

Exemple 6 fonction toto(a:entier):entier
  début
    a ← a + 3 ;
    retourner a + 3
  fin

fonction tata(a:entier) : entier
  début                                     % instant t10
    b ← toto(a) ;
                                           % instant t20
    c ← toto(toto(a)) ;
                                           % instant t30
    retourner c
  fin

```

FIG. 2.6 – Algorithmes `toto` et `tata`

Dans l'exemple de la fonction `toto`, en supposant que le paramètre `a` prenne pour valeur l'argument 10, l'état des variables aux instants `t1`, `t2` et `t3` sera :

```

instant t10  variables existantes : a
             valeurs de a         : 10
instant t20  variables existantes : a , b
             valeurs de a et b    : 10, 13

```

```
instant t30   variables existantes : a , b , c
              valeurs de a, b et c : 10, 13, 16
```

Remarquons dès à présent que la modification de la variable `a` à l'intérieur de la fonction `toto` (instruction `a <- a + 3 ;`) n'a pas entraîné de modification de valeur de la variable `a` du programme `tata` : à l'instant `t20`, la variable `a` conservé la valeur 10 qu'elle avait à l'instant `t10`.

2.3.14 Passage par références

Quand on définit un langage de programmation, il existe deux façons principales de définir la sémantique d'un appel de fonction. Le choix que nous ferons est celui de passage par références (par opposition au passage par valeurs) : les objets complexes tels les tableaux sont passés par références. Lors de l'exécution de l'algorithme, l'appel à une fonction consiste non pas à recopier tous les éléments du tableau passé en argument mais à simplement fournir la référence (ou l'adresse) du tableau. Ce choix a deux conséquences :

1. un gain en temps et en espace.

Contrairement à un passage par valeurs qui nécessite une copie de l'objet et donc une complexité en temps et en espace linéaire en la taille de l'objet, le passage par références nécessite de mémoriser qu'une référence et donc a un coût en temps et en espace constant.

2. un effet de bord.

Conséquence immédiate de sa définition, l'espace mémoire modifié par la fonction appelé est celui initial. Si l'on écrit dans cet espace, après retour de la fonction appelé, les modifications sont maintenues.

Ainsi, dans l'exemple suivant :

```
procedure doublerT(T:tableau d'entiers)
```

```
  pour i de 1 à longueur(T) faire
    T(i) ← 2 · T[i]
```

```
fonction exempleT(taille):entier
```

```
  T ← constructTab(taille)(1) ;
      % instant t0
  doublerT(T) ;
      % instant t1
  doublerT(T) ;
      % instant t2
  retourner T[taille]
```

la fonction `exempleT` retourne l'entier 4. La valeur du tableau `T` d'une taille 5 est aux instants `t0`, `t1` et `t2` successivement égale à :

(1,1,1,1,1) ; (2,2,2,2,2) ; (4,4,4,4,4)

Ce mécanisme ne concerne que des objets complexes tels les tableaux ou les matrices. Comme cela a été dit précédemment, des objets simples comme les objets de type booléen ou entier sont passés par valeurs. Ainsi, dans l'exemple suivant :

```

procedure doubler(t:entier)

    t ← 2 · t

fonction exemple(taille : entier):entier

    t ← 1 ;
           % instant t0
    doubler(t) ;
           % instant t1
    doubler(t) ;
           % instant t2
    retourner t

```

la fonction `exemple` retourne l'entier 1. La valeur de l'entier t est aux instants t_0, t_1 et t_2 successivement égale à : 1 ; 1 ; 1.

Remarque 2 Dans l'écriture de certains algorithmes, on peut vouloir insister sur le fait que le tableau est passé par référence. Une notation courante est la notation **ES** signifiant Entrée/Sortie. Le prototype de la procédure `doublerT` serait alors `procedure doublerT(ES T:tableau d'entiers)`.

Les étudiants sont libres de choisir ou non cette notation. Cela n'a aucune conséquence sur le sens de l'algorithme : les objets de type complexe sont toujours supposés être passés par référence.

Remarque 3 On pourrait dans certains cas vouloir passer un objet de type non complexe, un entier par exemple, par référence. La notation **ES** pourrait alors être utilisée. Mais cela nécessiterait de devoir manipuler des références d'entier et donc d'enrichir notre langage algorithmique et de le compliquer.

Dans le cas de la fonction `exemple`, si l'on souhaite doubler la valeur de t , la notation **ES** n'est pas la solution préférable. Il suffit de définir l'algorithme `doubler` comme une fonction qui retourne un entier et de remplacer l'instruction `doubler(t)` par l'instruction `t ← doubler(t)`.

Exercice 6 Expliquer pourquoi l'instruction `doublerT(T)` est équivalente à l'instruction `T ← doubler2T(T)` où `doubler2T` est une fonction de même définition que la procédure `doublerT` excepté une dernière ligne égale à `retourner T`.

Exercice 7 Ecrire une procédure qui résoud le problème suivant :

`combinaisonSuivante`

Entrée : un tableau de booléen T

Sortie : le booléen vrai et le tableau successeur de T si il existe le booléen faux et un tableau de booléen sinon
(le tableau est retourné par effet de bord)

Le tableau successeur d'un tableau de booléens T est un tableau de booléens de même taille obtenu par simple incrémentation (en confondant 0 et faux ainsi que 1 et vrai) : le tableau (0,1,0,1,1) a pour tableau successeur (0,1,1,0,0). Le tableau (1,1,1,1,1) n'a pas de successeur.

Récurtivité

Observons que tout ce qui n'est pas interdit étant autorisé, la définition d'une fonction peut contenir des appels à elle-même. L'exemple trop célèbre du problème Factoriel admet ainsi pour solution :

```
fonction factoriel(a:entier):entier
```

```
    si a = 1 alors
        retourner 0
    sinon
        retourner a · factoriel(n-1)
```

La compréhension de cet algorithme présente deux difficultés :

1. sa correction.

Cette difficulté en fait n'en est pas une. Sa correction est la conséquence immédiate de la définition mathématique même de $a!$: $a!$ peut être défini comme étant l'entier égal à 1 si $a = 1$ et égal à $a \cdot (a - 1)!$ sinon.

2. sa complexité(en temps et en espace).

Ce point plus délicat exige de connaître précisément le modèle de calcul, c'est à dire la façon dont sont gérés les différents appels récursifs, au sein de ce que appelons la *pile d'appel*.

Ces points seront appelés ultérieurement.

2.3.15 Indentation

Afin de gagner en lisibilité, on doit utiliser les sauts de ligne et les espaces blancs de façon à ce que l'écriture de votre programme mette en valeur sa structure intrinsèque. Ainsi, l'algorithme recherche2 écrit correctement plus haut ne peut pas s'écrire ainsi :

```
fonction recherche2(n : entier ; T : tableau d'entiers) : booléen
début res ← faux() ; i ← 1 ;
tant que (res=faux ET i ≤ longueur(T)) début si (n = T[i])
alors res ← vrai() ; i ← i + 1 ;
fin retourner res ; fin
```

Cette écriture est formellement correcte mais est illisible : un compilateur la comprendrait mais pas un humain.

On autorise souvent d'utiliser cette indentation pour masquer les parenthèses (début, fin) : en effet, il y a redondance entre un texte bien indenté et ces parenthèses. Ainsi, l'algorithme `recherche2` peut s'écrire de la façon décrite par la Figure 2.7.

```

fonction recherche2(n : entier ; T : tableau d'entiers) : booléen

    res ← faux() ;
    i ← 1 ;

    tant que (non(res) ET i ≤ longueur(T))
        si (n = T[i]) alors
            res ← vrai() ;
            i ← i + 1 ;

    retourner res ;

```

FIG. 2.7 – Écriture bien indentée

Utilisez cette tolérance avec précaution. En effet, une erreur d'indentation et votre programme est radicalement différent. Ainsi, le programme `recherche4` (Figure 2.8) est totalement différent, faux et d'ailleurs ne termine pas.

```

fonction recherche4(n : entier ; T : tableau d'entiers) : booléen

    res ← faux() ;
    i ← 1 ;

    tant que (non(res) ET i ≤ longueur(T))
        si (n = T[i]) alors
            res ← vrai() ;
            i ← i + 1 ;

    retourner res ;

```

FIG. 2.8 – Problème d'indentation

Exercice 8 Trouver une instance pour lequel `recherche4` ne termine pas. Écrire avec des parenthèses `début`, `fin` ce même algorithme `recherche4`.

2.4 Conclusion

Nous avons présenté dans ce chapitre de façon une syntaxe des programmes à écrire. Cette syntaxe est simple. Vous êtes en mesure de “comprendre” (au sens syntaxique) n’importe quel programme. Malheureusement, contrairement à une langue étrangère où la compréhension d’une syntaxe et d’un dictionnaire caractérise une bonne maîtrise de la langue, cette connaissance là n’est qu’un préambule à votre initiation aux algorithmes.

Nous verrons dans les prochains chapitres quelques méthodes très générales pouvant permettre de trouver la solution algorithmique à un problème donné (Chapitres 3, 5, 6 et 7). Nous verrons aussi comment écrire des algorithmes en respectant les propriétés essentielles d’un bon algorithme :

sa lisibilité

Obtenu notamment par de simples règles d’indentation, par des choix de noms de variables et de sous-fonctions, par une bonne décomposition du problème en sous-problèmes auxiliaires.

sa correction

La notion de correction sera définie dans le Chapitre 3. Nous veillerons naturellement à ce que l’ensemble des algorithmes présentés et étudiés soient corrects. Cependant les outils permettant de prouver formellement la correction d’un algorithme seront présentés dans les cours Analyse d’Algorithmes et Structures Arborescentes. Naturellement ces outils seront partiellement en TD.

sa faible complexité en temps et/ou en espace

Les Chapitres 4, 5, 6 et 7 aborderont ces notions.

Chapitre 3

Problèmes

Un problème est une relation binaire reliant des entrées ou instances à des sorties. Un problème en général n'est pas fonctionnel au sens où une entrée peut admettre plusieurs sorties possibles. Un algorithme est un texte. Il définit au travers d'un modèle du calcul (une machine) une autre relation binaire reliant des entrées et des sorties. Nous dirons qu'un tel algorithme résout le problème (ou de façon similaire, est une solution algorithmique, est correct vis à vis du problème) si pour toute instance du problème, toute exécution de l'algorithme fournit une sortie spécifiée par le problème.

Résoudre un problème algorithmique est difficile. Parfois, de telles solutions algorithmiques n'existent pas. Même parmi les problèmes admettant des solutions, il n'existe pas de méthode (d'algorithme) fournissant une solution à coup sûr. C'est pour cela que le génie humain peut exprimer tout son art et tenter de résoudre quelques uns de ces problèmes. Pour cela, nous vous invitons à procéder avec méthode (voir Discours du même nom).

L'algorithme présente nécessairement différentes étapes de calcul. Ces étapes peuvent se situer à des niveaux différents. Il est alors important de savoir les hiérarchiser. Nous procéderons selon une méthode descendante en découpant le problème en sous problèmes et en résolvant indépendamment ces différents morceaux. A un certain moment de cette découpe, apparaîtront des problèmes déjà rencontrés par ailleurs ou à l'évidence générale très simple à résoudre. Les solutions algorithmiques de ces "bas morceaux" n'auront pas nécessité pas à être écrites.

3.1 Définition d'un problème

Considérons le problème suivant :

problème FacteurProprePremier

Entrée : un entier $a > 1$

Sortie : un facteur propre premier de a si il en existe,
0 sinon.

La première tâche est la compréhension du problème. Souvent le problème est mal formalisé, présente une définition ambiguë voire incohérente. Ce premier travail nécessite de définir formellement le vocabulaire, les notions utilisées au degré suffisant.

Supposant acquises la notion d'entier, le comparateur $>$ ainsi que les constantes 0 et 1 il nous faut rappeler ce qu'est un facteur, un facteur propre ainsi qu'un entier premier. Cela nécessite l'évocation de l'opération \cdot considérée acquise.

1. Un entier a est facteur d'un entier b non nul si il existe c tel que : $b = a \cdot c$; il est facteur propre de b si il est distinct de 1 et de b .
2. Un entier premier est un entier admettant exactement deux facteurs (distincts) 1 et lui même.

3.2 Premières solutions issues de la définition

La formalisation d'un problème selon un langage précis et complètement défini est non seulement nécessaire à l'obtention de solutions algorithmiques mais fournit (très souvent) mécaniquement des solutions. Dans notre exemple, l'ensemble des définitions précédentes fournit comme premier algorithme :

```
fonction facteurProprePremier(a:entier):entier
```

```

  pour i de 0 à  $+\infty$  faire
    si estFacteurPropre(i,a)  $\wedge$  estPremier(i) alors
      retourner i

  retourner 0
```

Cet algorithme est correct, à ceci près qu'il ne termine pas dans le cas d'entiers premiers. Nous dirons qu'il est *partiellement correct* : il est trivialement vrai que toute sortie retournée est acceptée par le problème.

Pour corriger cet algorithme, il suffit de mentionner une propriété évidente : tout facteur propre d'un entier a appartient à l'intervalle $[2, a - 1]$. Une seconde propriété tout aussi évidente est que le plus petit facteur propre d'un entier a est premier. La traduction algorithmique de ces deux propriétés est l'algorithme suivant (totalement) correct :


```

fonction facteurProprePremier0(a:entier):entier

    pour i de 2 à a-1 faire
        si estDiviseur(i,a) alors
            retourner i

    retourner 0

```

Cette solution bien que concise ne fait que traduire dans un formalisme algorithmique le problème lui-même et quelques unes des propriétés basiques des objets mentionnés. Cet algorithme est en fait très lent (nous verrons dans le chapitre prochain que cet algorithme est en fait exponentiel). Peut-on faire mieux ?

Nous verrons dans la section suivante nous progresserons dans la compréhension du problème en le décomposant et en extrayant de celui-ci une composante difficile.

3.3 Comparaison de problèmes

Dans cette section, nous montrons comment extraire d'un problème des sous-problèmes plus simples. Ce qui permet d'une part de progresser d'un problème plus simple vers des problèmes plus difficiles et d'autre part de décomposer un problème en un ensemble équivalent de sous-problèmes. Il s'agit d'appliquer humblement les principes second et troisième du *Discours de la Méthode*.

Si vous rencontrez des difficultés à résoudre un problème, extrayez-en un plus simple et essayez de le résoudre. Nous dirons qu'un problème P est *aussi simple* d'un point de vue logique qu'un problème Q , si tout algorithme solution de Q est une solution de P . Si il est différent, il sera dit plus simple.

Si l'on vous demande de résoudre algorithmiquement un problème Q , pensez en en extraire un problème plus simple P . En effet, la résolution de P est un passage obligé pour la résolution de Q : toute solution de Q étant une solution de P , si vous ne savez pas résoudre P , vous ne savez pas résoudre non plus Q !

3.3.1 Augmenter les contraintes en entrée

Une façon d'en extraire un plus simple est de limiter les instances. Définissons un entier *composite* si il est distinct de 0, de 1 et si il n'est pas premier. Ainsi, si l'on se restreint aux entiers *composites*, le problème *FacteurProprePremier* se réduit en :

```

problème FacteurPremierRestreint
Entrée : un entier  $a > 1$  qui n'est pas premier
Sortie : un facteur premier de  $a$ 

```

3.3.2 Diminuer les contraintes en sortie

Une seconde façon de simplifier un problème est au lieu de restreindre les entrées, de relâcher les sorties c'est à dire de réduire les contraintes portant sur les sorties. Ainsi, un problème plus simple car extrait de `FacteurPremierRestreint` en remplaçant “facteur propre premier” par “facteur propre” est :

problème `FacteurPropre`

Entrée : un entier $a > 1$ qui n'est pas premier

Sortie : un facteur propre de a

En outre, un problème plus simple que `FacteurProprePremier` est obtenu en remplaçant “un facteur propre premier de n ” par un entier > 1 . On obtient clairement le problème suivant :

problème `EstComposite~`

Entrée : un entier $a > 1$

Sortie : un entier > 0 si il existe un facteur propre premier de a
0 sinon.

Ce dernier problème est appelé `EstComposite~` car il est très proche du problème qui décide si un entier est composite :

problème `EstComposite`

Entrée : un entier $a > 1$

Sortie : vrai si il existe un facteur propre premier de a
faux sinon.

Il est facile d'observer sur cet exemple, que toute solution algorithmique `facteurProprePremier` de `FacteurProprePremier` est une solution de chacun des problèmes :

1. `FacteurPremierRestreint`
2. `FacteurPropre`
3. `EstComposite~`

3.3.3 Décomposition en un ensemble équivalent de sous-problèmes

Dans la section précédente, nous avons montré que les problèmes `FacteurPremierRestreint` et `EstComposite` sont plus simples que `FacteurProprePremier`. L'algorithme suivant prouve que `FacteurProprePremier` se décompose exactement en deux premiers problèmes.

```
fonction facteurProprePremier(a:entier):entier
  si estComposite(a)
    retourner facteurPremierRestreint(a) ;
  sinon
    retourner 0
```

Observons que si l'on sait résoudre `FacteurPropre`, on sait résoudre `FacteurPremierRestreint` à la condition que l'on sache résoudre le problème `EstComposite`.

```

fonction facteurPremierRestreint(a:entier):entier

    faire
        a ← facteurPropre(a)
    jusqu'à
        non(estComposite(a))

;

retourner a

```

Conclusion méthodologique

Cet exemple illustre qu'un problème initial `FacteurProprePremier` peut se décomposer en deux problèmes `FacteurPropre` et `EstComposite`. Pour résoudre le premier il est nécessaire et même suffisant de résoudre les deux autres.

En introduction, nous avons indiqué que la solution `facteurProprePremier0` était une mauvaise solution car très lente : il peut utiliser jusqu'à \sqrt{a} instructions élémentaires.

Est-ce que l'algorithme `facteurProprePremier` est meilleur? Tout dépend bien entendu des fonctions auxiliaires `estComposite` et `facteurPropre`. De premières solutions immédiates seraient :

```

fonction estComposite(a:entier):booléen

    pour i de 2 à a-1 faire
        si estDiviseur(i,a)
            retourner faux

    retourner vrai

fonction facteurPropre(a:entier):entier

    pour i de 2 à a-1 faire
        si estDiviseur(i,a)
            retourner i

```

Ces solutions nécessitent chacune \sqrt{a} opérations ; si elles étaient retenues, entraîneraient un nombre d'opérations pour `facteurProprePremier` égal à $2 \cdot \sqrt{a}$.

Si l'on interroge la communauté scientifique, nous obtenons deux réponses :

1. la première réponse est une bonne nouvelle : le problème `EstComposite` peut être résolu à l'aide d'un algorithme `estComposite` nécessitant $(\log(a))^{12}$ opérations élémentaires.
Cet algorithme, découvert récemment, est trop compliqué pour être présenté dans ce cours.
2. la deuxième réponse est une mauvaise nouvelle : on ne sait pas résoudre `FacteurPropre` avec des algorithmes réellement plus efficaces que `facteurPropre`.

Sur un exemple, nous avons décomposé un problème `FacteurProprePremier` en deux problèmes `EstComposite` et `FacteurPropre`, nous avons montré que la résolution du premier problème était équivalent à la résolution des deux autres problèmes.

Ayant appris que le deuxième problème pouvait être résolu "rapidement", nous en avons déduit que la difficulté du premier résidait dans la difficulté du troisième.

Cette approche de décomposition doit être privilégiée systématiquement. Elle met en valeur des liens existant avec d'autres problèmes, éventuellement déjà étudiés ou déjà résolus. En outre, elle permet de répartir le travail :

- auprès de différentes équipes.

Chacune ayant pour objectif de résoudre l'un des sous problèmes.

- dans le temps.

Si à l'avenir, une meilleure solution apparaissait en ce qui concerne `FacteurPropre`, elle pourrait être introduite immédiatement dans votre algorithme.

Chapitre 4

Terminaison et complexités

À un même problème, différentes solutions algorithmiques peuvent être proposées. Nous avons vu dans le chapitre précédent l'existence d'algorithmes qui ne terminent pas, c'est à dire qui sur certaines entrées peuvent ne jamais retourner de résultat car entrant dans des boucles infinies. La première qualité attendue d'un algorithme est bien sûr sa terminaison.

Un second critère permet de les comparer et ainsi d'en distinguer de meilleures que d'autres. Ce critère est la faible utilisation de deux ressources :

1. le temps
2. l'espace.

4.1 Terminaison

L'une des qualités attendus d'un programme est qu'il termine, c'est à dire qu'il n'admette aucune instance pour laquelle l'exécution rentre dans une boucle infinie.

On touche ici à l'un des problèmes difficiles en informatique. Par exemple, la communauté scientifique n'a pas réussi à prouver que l'algorithme suivant :

fonction `syracuse(a:entierLong) : mot`

```
tantque a > 1 faire
  si a est pair alors
    a ←  $\frac{a}{2}$  ;
  sinon
    a ←  $\frac{3 \cdot a + 1}{2}$  ;

retourner ''fini''
```

terminait sur chaque entrée n . Nous avons bien-sûr exécuté l'algorithme sur un grand nombre d'entiers et observé que le calcul terminait chaque fois. Mais nous n'avons aucune certitude en ce qui concerne tous les entiers.

Cet exemple traduit le fait que nous n'avons pas de méthode pour décider si un algorithme termine. Pire, nous avons prouvé qu'il n'existe pas de méthode universelle pour décider si un algorithme termine. En clair, le problème de la terminaison :

Terminaison

Entrée : un algorithme A

Sortie : le booléen indiquant si A termine ou non

est un problème indécidable, c'est à dire incalculable : on prouve qu'il n'existe aucun algorithme le résolvant.

Exemple 7 Il existe d'autres exemples de problèmes indécidables. L'un des plus célèbres est le dixième problème de Hilbert, qui s'interrogeait en 1900 sur l'existence d'un algorithme décidant l'existence d'une racine entière pour une équation polynomiale à coefficients entiers. Nous savons aujourd'hui qu'il n'en existe pas : ce problème est indécidable.

Ne prenez pas prétexte de l'indécidabilité de la terminaison, pour produire des algorithmes ne terminant pas. Avec un minimum de méthodologie, il est possible lorsque vous écrivez un algorithme de vous assurer de façon formelle qu'il termine.

4.2 Complexité

Nous l'avons déjà dit, un grand nombre de problèmes fournissent dans la définition mathématique des objets mentionnés une solution algorithmique.

problème Puissance

Entrée : un réel x , un entier a

Sortie : le réel x^a

Si on utilise la définition de x^n vue en troisième ou en quatrième, à savoir x multiplié par lui même n fois, on obtient l'algorithme suivant :

fonction puissance1(x:réel ; a : entier) : réel

```
res ← 1.0 ;
```

```
faire a fois
```

```
  res ← res · x ;
```

```
retourner res
```

Si vous préférez la relation récursive apprise en seconde x^a est égal à 1 si a vaut 0 et $x \cdot x^{a-1}$ sinon. Vous en déduisez l'algorithme suivant :

```

fonction puissance2(x:réel ; a : entier) : réel

    si (a=0) alors
        res ← 1.0 ;
    sinon
        res ← x.puissance2(x,a-1) ;

    retourner res

```

Ces deux algorithmes sont tous deux corrects mais ne sont pas équivalents, le second utilisant davantage de ressources espace que le premier. Les deux ressources que nous considérerons ici sont le temps et l'espace.

4.3 Complexité d'une entrée

Avant de définir ce qu'est la complexité d'un algorithme, il nous faut définir la complexité d'une entrée (instance) manipulée par cet algorithme.

Définition 1 La *complexité* (ou *taille*) d'une entrée est le nombre d'octets nécessaires à sa représentation.

Comme nous le verrons par la suite, la complexité d'un objet (entrée, algorithme ou problème) est définie à une constante multiplicative près.

4.3.1 Complexité d'un booléen

Ainsi, par exemple, un booléen nécessite pour sa représentation un octet (en fait un bit suffit). Nous dirons qu'il est de complexité ou taille constante.

Complexité d'une matrice de booléen

Une matrice n lignes, m colonnes de booléens est de complexité $n \cdot m$.

4.3.2 Complexité d'un entier

Comme l'a indiqué la Section 2.2.1, deux hypothèses sont possibles :

1. La taille de l'entier est fixe. Il s'agit du type `entier` ou `int`. Cette hypothèse sera la plus fréquente. Qu'il s'agisse d'indices ou d'entiers apparaissant dans des tableaux et des matrices.
2. La taille de l'entier n est égal à son logarithme. Il s'agit du type `entierLong`. Cette hypothèse sera retenue dans le cas d'algorithmes destinés à la manipulation de longs entiers, plusieurs centaines de bits.

4.3.3 Complexité d'une matrice d'entiers

Quand nous considérerons une matrice d'entiers n lignes m colonnes, nous supposerons que ceux-ci sont de taille constante. Ainsi, la complexité de la matrice est considérée égale à $n \cdot m$.

4.4 Complexité d'un algorithme

Un algorithme est un objet qui à partir de toute entrée permet d'exécuter des instructions en consommant deux ressources :

1. du temps. Le temps sera évalué en considérant le nombre d'instructions élémentaires devant être exécutées : une instruction est élémentaire si elle peut être exécutée en un temps fixe.
2. de l'espace. L'espace est le nombre d'octets utilisé par l'exécution de l'algorithme. Il ne tient pas compte de l'espace utilisé par les objets fournis en entrée.

4.4.1 Instruction élémentaire

Le caractère élémentaire d'une instruction dépend des hypothèses initiales.

Ainsi, si l'on décide de manipuler des entiers à taille fixe (sur 4 octets par exemple), l'addition d'entiers (ainsi d'ailleurs que toutes les opérations) doit être considérée comme élémentaire.

À l'opposé si l'on considère des entiers pouvant être de grande taille (plusieurs centaines d'octets et davantage), l'addition ne peut pas être considérée comme élémentaire car elle nécessite autant de manipulations de bits qu'il y en a de présents.

4.4.2 Espace utilisée par l'algorithme

Dans les algorithmes itératifs, l'espace utilisé peut se résumer à celui nécessaire pour représenter les nouvelles variables utilisées.

```
fonction factIter(a:entierLong):entierLong
```

```
    res ← 1 ;
```

```
    tantque (a>1) faire
```

```
        res ← res · a ;
```

```
        a ← a - 1 ;
```

```
    retourner res
```


Ainsi la complexité en espace de cet algorithme est constant. La complexité en temps est (un multiple de) n .

Dans les algorithmes récurifs, il faut considérer en outre l'espace requis pour gérer l'ensemble des appels récurifs. Cette gestion est organisée au travers de ce que l'on appelle une *pile d'appel* dont la taille est égale au nombre d'appels récurifs. Ainsi, par exemple l'algorithme :

```
fonction factRec(a:entierLong):entierLong
    si a = 0
        retourner 1
    sinon
        retourner a · factRec(a-1)
```

nécessite une pile d'appel de hauteur n . En supposant que la multiplication soit élémentaire (en temps constant), la complexité en espace est un multiple de n ainsi que celle en temps.

4.4.3 Complexité en temps dans le pire des cas

La *complexité en temps dans le pire des cas* d'un algorithme A est la fonction qui à tout entier n associe le nombre d'instructions élémentaires maximal exécutées par A sur des entrées de complexité n . Ainsi, l'algorithme :

```
fonction decalage(a:entierLong):entierLong
    si a = 0 alors retourner 0
    tant que estPair(a) faire
        a ←  $\frac{a}{2}$ 
    retourner a
```

qui consiste à supprimer les bits nuls de poids faible a une complexité en temps (si l'on suppose l'évaluation de $\frac{a}{2}$ élémentaire, ce qui est toujours le cas) :

1. constante pour tout entier impair.
2. égale à $\log(a)$ pour toute puissance de 2.

Ainsi, sa complexité en temps dans le pire des cas est $\log(a)$.

Avertissement

Quand nous évoquerons la complexité d'un algorithme, nous considérerons la complexité dans le pire des cas. Cependant il en existe d'autres la complexité en moyenne et celle dans la meilleur des cas.

4.4.4 Complexité en temps en moyenne

La définition diffère de celle dans le pire des cas en considérant non le nombre maximal d'instructions élémentaires mais la moyenne du nombre d'instructions élémentaires sur l'ensemble des entrées de taille n .

Calculer en moyenne est un exercice souvent plus délicat que dans le pire des cas. Dans le cas de l'algorithme `decalage`, cette complexité en moyenne est constante. En mesurant le nombre d'exécutions de `a <- a/2`, on observe sur les entiers de taille 3 (représenté ci-dessous en binaire) :

```
001 : nbre d'instruction 0
010 : nbre d'instruction 1
011 : nbre d'instruction 0
100 : nbre d'instruction 2
101 : nbre d'instruction 0
110 : nbre d'instruction 1
111 : nbre d'instruction 0
```

Ainsi, le nombre moyen d'exécutions de $\frac{a}{2}$ est sur des entiers de taille 3 égal à $\frac{4}{7}$.

Exercice 9 Prouver que la complexité en moyenne de `decalage` est constant.

4.4.5 Complexité en temps dans le meilleur des cas

La définition diffère de celle dans le pire des cas en considérant non le nombre maximal d'instructions élémentaires mais le nombre minimal d'instructions élémentaires sur l'ensemble des entrées de taille n .

Dans l'exemple de `decalage`, le meilleur des cas est naturellement atteint par les entiers impairs. Ainsi, la complexité en temps dans le meilleur des cas de `decalage` est constante.

4.4.6 Complexité en espace

Pour définir la complexité en espace, il suffit de remplacer le terme "nombre d'instructions élémentaires exécutées par A " par "nombre d'octets utilisées lors de l'exécution de A ". À l'image de la complexité en temps, il existe notamment trois complexités en espace :

1. dans le pire des cas.
2. en moyenne
3. dans le meilleur des cas.

4.5 Complexité d'un problème

Une idée naturelle est d'évaluer la complexité en temps d'un problème (de même pour l'espace). Puisque un problème admet plusieurs solutions algorith-

miques, on peut les comparer sous le critère de leur complexité en temps et considérer la plus faible que nous définissons comme la complexité du problème.

4.5.1 Compromis espace-temps

Il n'existe pas parfois de meilleur algorithme.

Nous verrons plus loin, qu'un même problème ayant en entrée des entrées de taille n peut par exemple admettre :

- une solution algorithmique de complexité en temps dans le pire des cas en n^2 et une complexité en espace constante.
- une solution algorithmique de complexité en temps dans le pire des cas en n et une complexité en espace n .

Ces deux solutions étant incomparables, l'une ne peut être considérée comme meilleure que l'autre.

4.6 Notations et simplifications

Le calcul des complexités est "complexe". Pour les mêmes raisons qu'en ce qui concerne la terminaison, il n'existe pas de méthode générale (ou algorithme) pour calculer la complexité d'un algorithme et encore moins celle d'un problème.

Nous montrerons comment évaluer la fonction complexité d'un algorithme en évaluant non pas précisément cette fonction mais la classe à laquelle elle appartient.

4.6.1 À une constante multiplicative près : notation Θ

Les fonctions complexité que nous considérons dès à présent sont supposées à valeurs entières strictement positives.

Supposons que nous utilisions des entiers manipulés au travers d'additions et de multiplications considérées comme instructions élémentaires. Supposons que nous ayons deux algorithmes. Le premier sur une entrée de taille n provoque $10 \cdot n$ additions et n multiplications. Le second n additions et $2 \cdot n$ multiplications. Le premier a pour complexité en temps $n \mapsto 11 \cdot n$. Le second $n \mapsto 3 \cdot n$.

Lequel est optimal ? En fait tout dépend des complexités réelles de l'addition et de la multiplication, c'est à dire du nombre de cycles d'horloges nécessité par le processeur pour additionner et multiplier deux entiers. Deux stratégies s'offrent à nous :

1. ou on comptabilise pour chacune des opérations le nombre d'exécutions. On obtient alors tant de comparaisons, tant d'affectations, tant d'additions, tant de multiplications. Cette précision est parfois nécessaire dans l'étude d'algorithmes embarquées ou à temps réel mais est coûteuse.

2. ou on considère toutes ces opérations élémentaires comme équivalentes. C'est le choix que nous ferons. En conséquence de quoi, il est absurde de préférer $n \mapsto 3 \cdot n$ à $n \mapsto 11 \cdot n$.

En conséquence de quoi, deux fonctions f et g égales à une constante multiplicative près devront être considérées comme équivalentes.

La conséquence de ceci, deux fonctions f et g pour lesquelles il existe des réels $0 < \alpha$ et $0 < \beta$ tels que :

$$\forall n \alpha \cdot f(n) \leq g(n) \leq \beta \cdot f(n)$$

et dont tels que $\frac{1}{\beta} \cdot g(n) \leq f(n) \leq \frac{1}{\alpha} \cdot g(n)$ devront être considérées comme équivalentes. Conséquence de ce choix, deux simplifications interviennent :

À une constante additive près

Soit $f : \mathbb{N}^* \rightarrow \mathbb{N}^*$ une fonction et k un entier.

La fonction $k + f$ vérifie $\frac{1}{k+1} \cdot k + f \leq f \leq k + f$. Donc ces deux fonctions seront considérées comme équivalentes.

Comportement asymptotique

Soit $f : \mathbb{N}^* \rightarrow \mathbb{N}^*$ une fonction et N un entier. Soit g la fonction qui à tout entier $n \in [1, N]$ associe 1 et à tout autre entier associe $f(n)$. Soit $M := \max_{i \in [1, N]} f(i)$. Il est facile d'observer que :

$$\forall n g(n) \leq f(n) \leq M \cdot g(n)$$

En d'autres termes, deux fonctions étant égales asymptotiquement doivent être considérées comme équivalentes.

Notation Θ

Soit $f : \mathbb{N}^* \rightarrow \mathbb{N}^*$ une fonction. Nous noterons $\Theta(f)$ l'ensemble des fonctions $g : \mathbb{N}^* \rightarrow \mathbb{N}^*$ telles qu'il existe un entier N , deux réels α et β tels que pour tout entier $n > N$ on ait :

$$\alpha \cdot f(n) \leq g(n) \leq \beta \cdot f(n)$$

Du fait que $f \in \Theta(g)$ induit une relation d'équivalence (Exercice 11). Pour cette raison, la notation $f \in \Theta(g)$ est remplacée par la notation $f = \Theta(g)$.

Conclusion

Quand nous cherchons à calculer une fonction complexité f (dans le pire des cas), nous nous intéressons à une approximation de la fonction f équivalente à f qui vérifie $f \in \Theta(g)$. Ce calcul ainsi simplifié nécessite cependant deux démonstrations distinctes :

majoration la preuve que pour tout entier n , pour toute entrée E de taille n , la complexité observée lors de l'exécution sur E est majorée par $g(n)$ (à une constante multiplicative près fixée).

minoration la preuve que pour tout entier n , il existe au moins une entrée E de taille n pour laquelle la complexité observée lors de l'exécution sur E est minorée par $g(n)$ (à une constante multiplicative près fixée).

Exercice 10 Démontrer pour toutes fonctions $f, g : \mathbb{N}^* \rightarrow \mathbb{N}^*$ et tout réel $\alpha > 0$:

1. $\Theta(\alpha \cdot f) = \Theta(f)$
2. $\Theta(f + g) = \Theta(\max(f, g))$

Exercice 11 Démontrer que la relation $g \in \Theta(f)$ induit une relation d'équivalence (réflexive, symétrique, transitive), c'est à dire que pour toute fonction f, g et h on a :

1. $f \in \Theta(f)$.
2. $f \in \Theta(g)$ entraîne $g \in \Theta(f)$.
3. $f \in \Theta(g)$ et $g \in \Theta(h)$ entraîne $f \in \Theta(h)$.

4.6.2 Des fonctions étalons

Certaines fonctions $\mathbb{N}^* \rightarrow \mathbb{N}^*$ servent d'étalons et fournissent une hiérarchie simple de cet ensemble indénombrable de classes. Pour des raisons de simplicité, les fonctions sont notées plus simplement : ainsi la fonction $n \mapsto n^2$ est notée plus simplement n^2 . Quelques exemple des fonction :

1. la fonction 1 dite constante.
2. la fonction logarithmique $\log(n)$.
3. la fonction linéaire n .
4. la fonction $n \cdot \log n$.
5. les fonctions polynomiales n, n^2, \dots, n^k avec k fixé.
6. des fonctions superpolynomiales comme $2^{\sqrt{n}}$.
7. la fonction exponentielle 2^n .
8. des fonction superexponentielles telles que $n^n, 2^{2^n}$.

Les algorithmes que nous souhaitons faire exécuter sur machine doivent être de complexité en temps et en espace au pire polynomiale et ce selon une petite puissance (≤ 4) : un algorithme nécessitant 2^n opérations élémentaires nécessite dans le cas où $n = 10000$ (par exemple une petite matrice carrée de taille $100 \cdot 100$) un temps qui se compte en milliard d'années !

Exercice 12 Considérant qu'une opération élémentaire se fasse en 10^{-9} seconde (un milliardième de seconde), rappelant qu'une année comporte à peu près 3.10^9 seconde. Calculer le temps nécessaire à l'exécution d'un algorithme sur une entrée de taille 100, 1000 ou 10000 de complexité :

1. n ,
2. n^2 ,
3. n^6 ,
4. n^{100}
5. 2^n ,
6. 2^{2^n} .

4.6.3 Simple majoration : notation O

Considérons la fonction :

```
fonction toto(n : entierLong) : entierLong
```

```

i ← n ;

tantque i > 0 faire
    si testTpsConstant(i) alors
        retourner i
    ;
    i ← i-1
;
retourner i

```

Supposons que `testTpsConstant` se fasse en temps constant. Que peut-on dire de la complexité en temps (dans le pire des cas) de cet algorithme ? Tout dépend du test que réalise `testTpsConstant`. La complexité en temps de `toto` est, par exemple, :

1. $\Theta(1)$ si `testTpsConstant` teste la parité.
2. $\Theta(\log(n))$ si `testTpsConstant` teste la primalité : la proportion de nombres premiers est $\frac{1}{\log(n)}$.
3. $\Theta(n)$ si `testTpsConstant` teste le fait d'être puissance de 2.

Dans des situations comparables du fait de notre incapacité à évaluer finement la complexité, nous nous contenterons de majorer la fonction complexité. Dans l'exemple de la fonction `toto`, nous dirons que sa complexité en temps (dans le pire des cas) appartient à la classe $O(n)$, c'est à dire que sa fonction complexité temps est majorée (à une constante multiplicative près) par la fonction $n \mapsto n$.

Ainsi, pour toute fonction $f : \mathbb{N}^* \rightarrow \mathbb{N}^*$ la classe $O(f)$ est l'ensemble des fonctions g telles qu'il existe un entier N et un réel $\alpha > 0$ tels que pour tout entier $n > N$ on ait :

$$g(n) < \alpha \cdot f(n)$$

Pour des raisons de simplicité et de similarité avec la notation $g = \Theta(f)$, l'appartenance $g \in O(f)$ est noté $g = O(f)$.

À titre d'exercice, quelques petites propriétés :

Exercice 13 Démontrer pour toutes fonctions $f, g : \mathbb{N}^* \rightarrow \mathbb{N}^*$ et tout réel $\alpha > 0$:

1. $O(\alpha \cdot f) = O(f)$.
2. $O(f + g) = O(\max(f, g))$.
3. $\Theta(f) \subset O(f)$.

Exercice 14 Démontrer pour toutes fonctions $f, g : \mathbb{N}^* \rightarrow \mathbb{N}^*$:

1. $f = \Theta(g) \Leftrightarrow \Theta(f) = \Theta(g)$.
2. $f = O(g) \Rightarrow O(f) \subseteq O(g)$.
3. $f = O(g) \not\Rightarrow O(f) = O(g)$.

O induit une relation d'ordre et non d'équivalence

Observons que si $n \in O(n^2)$ (noté $n = O(n^2)$), nous avons $n^2 \notin O(n)$ (noté $n^2 \neq O(n)$). La relation induite par O n'est pas symétrique, elle est simplement d'ordre :

Exercice 15 Démontrer que la relation $g \in O(f)$ induit une relation d'ordre large (réflexive, transitive), c'est à dire que pour toute fonction f, g et h on a :

1. $f \in O(f)$.
2. $f \in O(g)$ et $g \in O(h)$ entraîne $f \in O(h)$.

4.6.4 Quelques règles d'évaluation de complexité

Pour chacun des exercices suivants, et pour chacun des entiers $i \in [1, 4]$, nous notons $\#fi$ la complexité en temps dans le pire des cas de la fonction fi .

4.6.5 Mise en séquence

Exercice 16 Considérer l'algorithme suivant :

fonction $f1(i, j : \text{entierLong}) : \text{entierLong}$

```

    i ← f2(i) ;
    j ← f3(j) ;

```

```

    retourner i ;

```

Démontrer que :

1. $\#f1 = O(\#f2 + \#f3)$.
2. $\#f1 = \Theta(\#f2 + \#f3)$.

Exercice 17 Considérer l'algorithme suivant :

```
fonction f1(i : entierLong) : entierLong
```

```
    i ← f2(i) ;
    retourner f3(i) ;
```

Démontrer que :

1. $\#f1 \neq O(\#f2 + \#f3)$ et donc $\#f1 \neq \Theta(\#f2 + \#f3)$.
Vous considérez des entiers de taille (complexité non constante). Vous choisissez une fonction $f2$ qui augmente significativement la taille de i par exemple une fonction exponentielle.
2. Exprimer selon une notation O la fonction $\#f1$ en fonction de $\#f2$, $\#f3$ et $f2$. La fonction $f2$ sera supposée croissante.

4.6.6 Branchement conditionnel

Exercice 18 Considérer l'algorithme suivant :

```
fonction f1(n:entierLong) : entierLong
```

```
    si f2(n) alors
        retourner f3(n)
    sinon
        retourner f4(n)
```

En supposant que n n'est pas modifié par l'appel de $f1$ ou $f2$, démontrer que :

1. $\#f1 = O(\#f2 + \#f3 + \#f4)$.
2. $\#f1 = \Theta(\#f2 + \#f3 + \#f4)$ n'est pas toujours vrai. Fournir un exemple.

4.7 Un peu de vocabulaire

Quand on précise la complexité d'un algorithme ou d'un problème, il faut définir en fonction de quelle quantité celle-ci est exprimée. Nous pouvons employer l'expression "tel algorithme est de complexité linéaire". À défaut de toute précision, ceci signifie linéaire en fonction de la complexité (la taille) de l'entrée x .

4.7.1 Un algorithme linéaire

Considérons une solution au problème de la comparaison de deux matrices carrées de même taille de la Figure 4.1. Cet algorithme est de complexité en temps $\Theta(l^2)$ quadratique en fonction de l mais doit être considéré comme un simple algorithme linéaire (en la taille des entrées A et B égale à $n = l^2$) qui compare bit à bit les représentations machines des objets. La fonction complexité est en fait $\Theta(n)$.


```

fonction égal(A,B: matrice d'entiers): booléen

    l ← nbColonnes(A) ;

    pour i de 1 à l faire
        pour j de 1 à l faire
            si A[i,j] ≠ B[i,j] alors
                retourner faux()

    retourner vrai();

```

FIG. 4.1 – Comparaison de deux matrices

4.7.2 Un algorithme exponentiel

Considérons l'algorithme suivant.

```

fonction factIter(k:entierLong):entierLong

    res ← 1 ;

    tantque (k>1) faire
        res ← res · k ;
        k ← k - 1 ;

    retourner res

```

Supposons que le produit de deux entiers est de complexité constante. Cet algorithme est de complexité en temps $\Theta(k)$ linéaire en fonction de l'entrée k mais doit être considéré comme exponentiel (en la taille de l'entrée k égale à $n = \log_2(k)$). La fonction complexité est en fait : $n \mapsto 2^n$.

Exercice 19 Qualifier chacune des complexité en temps et en espace de l'algorithme d'addition de deux matrices carrées. S'agit t- il de complexité constante ? logarithmique ? linéaire ? quadratique ? exponentielle ? autre ?

Même question pour le produit de deux matrices carrées ?

Exercice 20 Considérons l'un des premiers algorithmes que vous exécutiez dans votre jeune âge (l'entier est fourni par l'enseignant : la machine c'est vous!) :

```

procédure punition(k: entierLong)

    pour i de 1 à k faire
        écrire (''J'apprendrais mes leçons'')

```

Même question que l'exercice précédent.

4.8 Étude du problème de puissance

4.8.1 Premiers algorithmes triviaux, simples mais lents

Considérons le problème de puissance d'un élément x selon une puissance k .

problème Puissance

Entrée : un élément x , un entier k

Sortie : x^k

Une première définition de la puissance d'un élément x selon un entier k est le "dessin" : $x^k = \overbrace{x \cdot \dots \cdot x}^{k \text{ fois}}$. Elle se traduit immédiatement en l'algorithme suivant :

fonction puissance1(x :élément ; k : entier) : élément

```
res ← elementNeutreProduit() ;
```

```
faire  $k$  fois
  res ← res · x ;
```

```
retourner res
```

Une deuxième définition plus formelle et est récursive : $x^k = \begin{cases} 1 & \text{si } k = 0 \\ x \cdot x^{k-1} & \text{sinon} \end{cases}$

Elle se traduit tout aussi immédiatement en :

fonction puissance2(x :élément ; k : entier) : élément

```
si (k=0) alors
  retourner elementNeutreProduit() ;
sinon
  retourner x.puissance2(x,k-1) ;
```

4.8.2 Analogie avec le produit

Les deux résolutions précédentes sont aussi naïve que celle du problème `Produit` par l'algorithme `produit1` ci-dessous.

problème `Produit`

Entrée : un élément x , un entier k

Sortie : $x \cdot k$

FIG. 4.2 – Le problème `Produit`

Une bien meilleure solution apprise dans nos très jeunes années est :

```

fonction produit1(x:élément ; k : entier) : élément

    res ← elementNeutreAddition() ;

    faire k fois
        res ← res + x ;

    retourner res

```

FIG. 4.3 – L’algorithme naïf et long produit1

```

fonction produit3(x:élément ; k : entier) : élément

    res ← elementNeutreAddition() ;
    l ← nbreBits(k) ;

    pour i de 1 à l faire
        si iemeBitFaible(k,i) = 1 alors
            res ← res + x ;
            x ← double(x) ;

    retourner res

```

FIG. 4.4 – Un algorithme appris dès l’age de 6 ans

4.8.3 Premier algorithme rapide

Pour la simple et triviale observation que la définition de l’opération mathématique puissance est celle de du produit en remplaçant $+$ par \cdot , nous tirons de l’algorithme produit3 la solution suivante au problème de la puissance :

```

fonction puissance3(x:élément ; k : entier) : élément

    res ← elementNeutreProduit() ;
    l ← nbreBits(k) ;

    pour i de 1 à l faire
        si iemeBitFaible(k,i) = 1 alors
            res ← res · x ;
            x ← carre(x) ;

    retourner res

```

Exercice 21 Assurez-vous sur un exemple de la correction de puissance3. Puis prouvez formellement et directement celle-ci.

4.8.4 Second algorithme rapide

Une solution rapide au problème de la puissance peut être obtenue en considérant la propriété vérifiée par tout élément x et tout entier n :

$$x^{2 \cdot k} = (x^2)^k$$

Puis, en la complétant de façon à obtenir une propriété non caractéristique c'est à dire une définition équivalente de l'opération puissance :

$$x^k = \begin{cases} 1 & \text{si } k = 0 \\ (x^{\frac{k}{2}})^2 & \text{si } k \text{ est pair} \\ (x^{\frac{k-1}{2}})^2 \cdot x & \text{sinon} \end{cases}$$

Et enfin, en la traduisant selon le formalisme algorithmique considérée :

fonction puissance4(x:élément ; k : entier) : élément

```

si k=0 alors
    retourner elementNeutreProduit()
sinon si estPair(k) alors
    retourner carre(puissance4(x, k/2))
sinon
    retourner carre(puissance4(x, (k-1)/2)) · x

```

Dérécursivation

A partir d'un algorithme récursif, il est toujours possible d'extraire un algorithme itératif. Nous pouvons ici observer le détail des calculs réalisés par `puissance4` et en déduire une solution itérative. L'étude d'un exemple suffit à avoir une vision complète. Considérons un élément x et l'entier $k = 22 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$ de représentation binaire (bit de poids faible à droite) égale à 10110. Le calcul sur l'entrée $(x, 22)$ entraîne successivement l'appel sur les entrées $(x, 11)$, $(x, 5)$, $(x, 2)$, $(x, 1)$ et $(x, 0)$. L'ensemble des calculs peut se représenter simplement à l'aide de l'expression suivante où 1 représente l'élément neutre du produit et où $()^2$ désigne l'opérateur `carre` :

$$((((1)^2 \cdot x)^2)^2 \cdot x)^2 \cdot x)^2$$

Cette expression permet de visualiser l'organisation du calcul. Les 5 opérations $)^2 \cdot x$, $)^2$, $)^2 \cdot x$, $)^2 \cdot x$ et $)^2$ correspondent aux 5 bits 10110 de la représentation binaire de k . L'algorithme itératif réalisant les mêmes calculs est :

Exercice 22 Considérer la propriété $x^{2 \cdot n} = x^{2^n}$.

1. Extraire de cette propriété une caractérisation de la puissance.
2. Ecrire un algorithme récursif.

```

fonction puissance5(x:élément ; k : entier) : élément

    l ← nombreBits(k) ;
    res ← elementNeutreProduit() ;

    pour i de 1 à l faire
        res ← carre(res) ;

        si ièmeBitFort(k,i)=1 alors
            res ← res · x ;

    retourner res

```

FIG. 4.5 – Algorithme extrait d'un algorithme récursif

3. Fournir la séquence des calculs réalisés par cet algorithme sur l'entrée $(x, 22)$?
4. Peut-on extraire de cet algorithme récursif un algorithme itératif? Lequel?

Exercice 23 Démontrez la correction de l'algorithme `puissance4`.

Exercice 24 En utilisant l'analogie puissance-produit, réécrire l'algorithme `puissance5` en un algorithme réalisant le produit.

4.8.5 Analyse des complexités : l'importance des hypothèses

Cette section a pour objet d'étudier la complexité en temps (dans le pire des cas) des algorithmes puissances vus précédemment en fonction du type choisi `élément` et `entier`. Nous nous contenterons d'étudier la complexité en temps dans le pire des cas des deux algorithmes `puissance1` et `puissance3`.

Afin de simplifier l'étude, nous supposons que :

1. les opérations `nombreBits`, `elementNeutreProduit`, `ièmeBitPaible`, `retourner` sont des opérations de complexité en temps constant, noté $\Theta(1)$.
2. l'opération produit noté \cdot et donc l'opération `carre` est de complexité en temps linéaire en la taille des éléments opérés.

Ce qui signifie que $\log(a) + \log(b)$ opérations suffisent pour faire le produit de deux entiers longs a et b ou que n^2 opérations suffisent à faire le produit de deux matrices carrées $n \cdot n$. Cette hypothèse est (presque) avérée pour les entiers, un peu moins pour les matrices (voir prochain chapitre). Cette simplification, un peu grossière, nous permet de nous concentrer sur l'essentiel de notre propos qui est de distinguer non pas le polynôme x^1 du polynôme $x^{1.1}$ mais de distinguer le polynôme x^1 de l'exponentielle 2^x .

Méfiez vous des conclusions hâtives

Ainsi, la complexité en temps de `puissance1` est la somme des complexité de k produits successifs et celle de `puissance1` la somme des complexités de $l := \log(k)$ produits successifs (à une constante 2 multiplicative près).

Une lecture rapide du propos précédent peut vous amener à conclure que le second algorithme est bien plus rapide que le premier. Il peut n'en être rien. Tout dépend de la complexité du dernier produit. En effet, le dernier produit peut être aussi complexe que tous ceux qui l'on précédé, c'est à dire aussi complexe que l'algorithme lui-même.

En fait deux cas principaux se présentent selon que le résultat du produit de deux éléments conserve la taille des éléments initiaux ou prend pour taille la somme des tailles de ces éléments initiaux.

Si conservation de la taille, `puissance1` est exponentiel, `puissance1` est quadratique.

Si il y a conservation de taille, tous les produits ont la même complexité, celle notée `taille(x)`, du premier élément x passé en argument. Ainsi, la complexité en temps dans le pire des cas de :

– `puissance1` est $\Theta(k \cdot \text{taille}(x))$.

En clair, cet algorithme est exponentiel puisque k est une fonction exponentielle de la taille $\log(k)$ de k .

– `puissance3` est $\Theta(\log(k) \cdot \text{taille}(x))$.

En clair, cet algorithme est quadratique puisque $\log(k)$ est la taille de k . Voir exemples ci dessous.

Si somme des taille, `puissance1` et `puissance1` sont exponentiels.

Si la taille du produit est la somme des tailles des opérandes, la complexité en temps dans le pire des cas de :

– `puissance1` est $\Theta(k^2 \cdot \text{taille}(x))$.

En effet, l'algorithme réalise k produits sur une opérande droite de taille `taille(x)` et une opérande gauche de taille successivement égal à 1, `taille(x)`, $2 \cdot \text{taille}(x)$, $\dots (k-1) \cdot \text{taille}(x)$. La somme fournit très exactement $(k(k+1)/2 - 1) \cdot \text{taille}(x)$ que l'on approche selon $k^2 \cdot \text{taille}(x)$.

– `puissance3` est $\Theta(k \cdot \text{taille}(x))$.

En effet, l'algorithme réalise $\log(k)$ carrés sur une opérande de taille successivement égal à $1 \cdot \text{taille}(x)$, $2 \cdot \text{taille}(x)$, $4 \cdot \text{taille}(x)$, $\dots 2^{\log(k)} \cdot \text{taille}(x)$. Sachant que $\sum_{0 \leq i < \log(k)} 2^i = 2^{\log(k)+1} - 1 = \Theta(k)$, on en déduit que la complexité en temps de l'algorithme est celle du dernier produit exécuté c'est à dire $\Theta(k \cdot \text{taille}(x))$.

Nous présentons ici différents types `element` et interprétons le résultat dans le contexte de ce choix.

le type entierLong : augmentation de la taille

Un entier x est ici de taille $\log(x)$. Les deux algorithmes sont exponentiels. Même en supposant qu'un produit sur des entiers de 8 bits se fasse en 10^9 seconde, le calcul de `puissance1(10,10000000000000000000)` nécessite $(10^{20})^2 = 10^{40}$ opérations élémentaires c'est à dire 10^{23} années de calculs, ce qui fait beaucoup. Le second plus rapide nécessite tout de même 10^{20} opérations élémentaires soit 100 années de calcul, ce qui n'est pas rien !

Le type int : conservation de la taille

Le type `int` est de taille constante, souvent 4 octets. Le produit est de complexité constante et ne modifie pas la taille des entiers multipliés.

En fait, l'analyse et théorique et pratique de la complexité n'a aucun sens puisque pour de très très petits entiers, on dépasse la quantité maximale autorisée : 10^{10} est déjà supérieur à 2^{32} et donc nécessite plus de 4 octets.

Le type float : conservation de la taille

Le type `float` est de taille constante, 4 octets. Le produit est de complexité constante et ne modifie pas la taille des entiers multipliés. Contrairement aux `int`, un `float` étant représenté grace notamment à un exposant prenant 256 valeurs. On peut calculer la puissance d'un `float` selon un petit entier `char`.

D'un point de vue pratique, le début (?) d'un comportement asymptotique peut être observé pour ces 256 valeurs.

Le type matrice carrée de réels : conservation de la taille

Si l'on considère une matrice $n \cdot n$ carrée de réels, ces derniers étant supposés de taille constante. Le produit clairement conserve la taille, toujours égale à $n \cdot n$. L'algorithme `puissance1` est de complexité exponentielle $\Theta(k \cdot n^2)$. L'algorithme `puissance3` est de complexité $\Theta(\log(k) \cdot n^2)$.

Remarque 4 Cette complexité peut être qualifiée de quadratique puisque la taille de l'entrée est $\log(k) + n^2$. La complexité $\log(k) \cdot n^2$ est majorée par le carré $(\log(k) + n^2)^2$. On peut atteindre cette complexité quadratique, il suffit de choisir un entier k de taille $\log(k) = n^2$, et obtenir une complexité $\log(k) \cdot n^2$ égale au carré $(\log(k) + n^2)^2$ de la taille de l'entrée (à la constante multiplicative $\frac{1}{4}$ près).

Exercice 25 Poursuivre l'étude de complexité en considérant au choix :

1. les autres algorithmes puissances `puissance2`, `puissance4`, `puissance5`, `produit1`, `produit3`.
2. la complexité en espace.
3. la complexité en moyenne, dans le meilleur des cas.

4. en choisissant la puissance définie par le produit modulo un entier n .
Le produit modulo un entier n de deux entiers a et b est l'entier noté $a \cdot_n b$ égal au reste du produit $a \cdot b$ par la division euclidienne par n . Exemple :
 $7 \cdot_9 5 = 8$.

Exercice 26 Évaluer la complexité en espace des différents algorithmes dans chacune des deux nouvelles hypothèses.

Exercice 27 Expliquer pourquoi dans le cadre de l'étude de la puissance supposer des entiers de taille fixe n'a pas beaucoup d'intérêt et ce contrairement aux réels (de taille fixe eux aussi).

Exercice 28 Qualifier la complexité de chacun des algorithmes dans chacune des hypothèses : est-elle constante, linéaire, polynomiale, exponentielle ?

Chapitre 5

Algorithmes “Diviser Pour régner”

L’approche “Diviser Pour Régner” (Divide and Conquer) est une méthode qui permet de résoudre un problème en fournissant un algorithme récursif. Cette méthode n’offre naturellement aucune garantie : il n’existe pas de méthode (ou algorithme) permettant à partir d’un problème d’obtenir à coup sûr une solution algorithmique.

La structure générale d’un algorithme “Diviser Pour Régner” A et permettant d’associer à une entrée x une solution s comporte 3 parties :

1. la décomposition. Celle-ci consiste à décomposer l’entrée x en a nouvelles entrées x_1, \dots, x_a .
2. a appels récursifs. Ceux-ci consistent à appliquer récursivement la fonction A sur chacune des nouvelles entrées x, \dots, x_a et à retourner les a solutions s_1, \dots, s_a .
3. la recombinaison. Celle-ci consiste à recomposer à partir des solutions partielles s_1, \dots, s_a la solution s associée à x .

5.1 Un premier exemple : la multiplication de deux entiers

Supposons que nous manipulions des entiers de très grande taille (plusieurs centaines d’octets) et que nous souhaitions les multiplier :

Problème ProduitEntiers

Entrée : a, b : entier

Sortie : $a \cdot b$

La taille n désigne la taille $\max(\log(a), \log(b))$ maximale des entiers a et b .

Ici, nous considérons comme opérations constantes, les seules opérations de lecture d’un bit, de modification d’un bit, d’accès au bit suivant, de suppression

du bit de poids faible (division par 2 noté $x \gg 1$) ou d’insertion d’un nouveau bit 0 de poids faible (multiplication par 2 noté $x \ll 1$). Les opérations d’addition, de soustraction seront de complexité en temps linéaires $\Theta(n)$.

5.1.1 Complexité au moins linéaire

Clairement, pour multiplier deux entiers a et b , chaque bit compte : la modification d’un seul bit modifie le résultat. Aussi, doit-on lire chacun des bits de a et chacun des bits de b . En conséquence de quoi, tout algorithme résolvant **Produit** est de complexité en temps au moins linéaire $\Theta(n)$.

5.1.2 Un algorithme existant quadratique

L’algorithme ancien vu **produit3** dans le chapitre précédent (Figure 4.4) est de complexité en temps $\Theta(n^2)$. Peut-on améliorer celui-ci en utilisant l’approche “Divide And Conquer” et ainsi se rapprocher de $\theta(n)$?

5.1.3 Une amélioration “Divide And Conquer” entre linéaire et quadratique

Quitte à incrémenter n , nous pouvons supposer que n est un entier pair. Décomposons a sous la forme $a = a_1 \cdot 2^{\frac{n}{2}} + a_2$: a_2 est composé des $\frac{n}{2}$ bits de poids faibles de a ; a_1 est composé des $\frac{n}{2}$ bits de poids forts de a . Décomposons b sous la forme $b = b_1 \cdot 2^{\frac{n}{2}} + b_2$. L’égalité évidente :

$$a \cdot b = (a_1 \cdot b_1) \cdot 2^n + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 2^{\frac{n}{2}} + a_2 \cdot b_2$$

entraîne l’égalité

$$a \cdot b = (a_1 + a_2)(b_1 + b_2) \cdot 2^{\frac{n}{2}} + a_1 \cdot b_1 \cdot (2^n - 2^{\frac{n}{2}}) + a_2 \cdot b_2 \cdot (1 - 2^{\frac{n}{2}})$$

Apparaît dans cette dernière égalité, un algorithme de multiplication d’entiers binaire plus efficace que l’algorithme traditionnel. Que nous dit cette égalité, pour multiplier a par b , il faut en fait :

1. décomposer l’entrée (a, b) en trois nouvelles entrées : $(a_1 + a_2, b_1 + b_2)$, (a_1, b_1) et (a_2, b_2) .
2. appliquer récursivement le produit sur chacune de ces entrées. Notons s_1 , s_2 et s_3 les produits associés.
3. recomposer le produit attendu s à partir des ces 3 solutions partielles s_1 , s_2 et s_3 de la façon suivante :

$$s \leftarrow s_1 \cdot 2^{\frac{n}{2}} + s_2 \cdot 2^n - s_2 \cdot 2^{\frac{n}{2}} + s_3 - s_3 \cdot 2^{\frac{n}{2}}$$

L’algorithme peut s’écrire :

```

fonction produit(a,b: entier):entier

    n ← taille(a,b) ;

    si n= 1
        si (a=1 ET b=1)
            retourner 1
        sinon
            retourner 0

    p ← n >> 1 ; % p= $\lfloor \frac{n}{2} \rfloor$ 

    (a1,a2) ← décomposition(a,n) ;
    (b1,b2) ← décomposition(b,n) ;

    s1 ← produit(addition(a1,a2),addition(b1,b2)) ;
    s2 ← produit(a1,b1) ;
    s3 ← produit(a2,b2) ;

    res ← s1<<p ; % signifie : res ← s·2p
    res ← addition(res, s2<<n) ;
    res ← soustraction(res, s2<<p) ;
    res ← addition(res, s3) ;
    res ← soustraction(res, s3<<p) ;

    retourner res

```

Évaluation de la complexité

Notons $f : \mathbb{N} \rightarrow \mathbb{N}^*$ la fonction de complexité en temps. Pour simplifier l'exposé, la taille d'un couple est considéré le maximum des tailles des deux entiers. Observant que pour multiplier deux entiers a et b de taille chacun au max n , il est nécessaire de :

1. de décomposer ces entiers et produire les trois couples d'entiers. Chacune des opérations (décomposition, taille, addition, décalage, affectation) est de complexité $O(n)$. La complexité de la décomposition est $\Theta(n)$.
2. de réaliser trois appels récursifs sur ces trois couples de taille chacun $\frac{n}{2}$.
3. de recomposer la solution finale à partir des trois solutions partielles. Cette recomposition nécessite 2 additions, 2 soustractions, 4 décalages droits : chacune de ces opération est de complexité en temps $O(n)$. La complexité de la recomposition est $\Theta(n)$.

En clair, la fonction complexité $f(n)$ est défini récursivement par $f(1) = 1$ et

$$f(n) = n + 3f\left(\frac{n}{2}\right)$$

Nous verrons comment résoudre un tel système d'équations. Nous pouvons ici le résoudre à la main. La solution est $f(n) = n + 3\left(\frac{n}{2} + 3\left(\frac{n}{2^2} + \dots\right)\right) = n\left(\left(\frac{3}{2}\right)^0 + \left(\frac{3}{2}\right)^1 + \dots\right) \approx n \cdot \left(\frac{3}{2}\right)^{\ln_2(n)}$ qui est égal à $n \cdot n^{\ln(\frac{3}{2})/\ln(2)}$. Observant que $1 + \ln(\frac{3}{2})/\ln(2)$ est égal à $\ln(3)/\ln(2) = \ln_2(3)$, nous avons :

$$f(n) = n^{\ln_2(3)} \approx n^{1,58}$$

5.1.4 Conclusion

Cet algorithme est donc meilleur que le premier algorithme de complexité $\Theta(n^2)$. De nouvelles améliorations “Divide and conquer” sont possibles qui permettent pour tout réel $\epsilon > 0$ de fournir un algorithme solution de **Produit** de complexité en temps $\Theta(n^{1+\epsilon})$.

D'autre part des techniques utilisant les transformées de Fourier permettent en temps linéaire $\Theta(n)$ de résoudre ce même problème. Ainsi, le produit est de même complexité qu'une addition ou qu'une comparaison.

5.2 Évaluation de la complexité

Évaluer la complexité d'un algorithme se fait en deux temps.

5.2.1 Définition récursive de la fonction

Cette première peut être immédiate. Il est nécessaire de “projeter” la définition récursive de l'algorithme en une définition récursive de la fonction complexité en temps. Nous pouvons alors obtenir par exemple l'équation :

$$f(n) = f\left(\frac{n-3}{4}\right) + 6 \cdot f\left(\frac{2n-1}{8}\right) + 3 \cdot n + \sqrt{n} + 879$$

Cette équation doit être débarrassée des termes marginaux, des constantes multiplicatives n'apparaissant pas dans le terme récursif. Nous obtenons alors pour équation :

$$g(n) = 7 \cdot g\left(\frac{n}{4}\right) + g$$

Exercice 29 Démontrer que les fonctions f et g vérifiant les deux systèmes d'équation précédent vérifient $f = \Theta(g)$ et donc $g = \Theta(f)$.

Exercice 30 Toute solution à une équation de la forme :

$$g(n) = a \cdot g\left(\frac{n}{b}\right) + \Theta(n^k)$$

appartient à la classe $\Theta(h)$ définie par la solution h de l'équation :

$$h(n) = a \cdot h\left(\frac{n}{b}\right) + n^k$$

Exercice 31 Toute fonction vérifiant une inéquation de la forme :

$$g(n) \leq a \cdot g\left(\frac{n}{b}\right) + \Theta(n^k)$$

appartient à la classe $O(h)$ définie par la solution h de l'équation :

$$h(n) = a \cdot h\left(\frac{n}{b}\right) + n^k$$

5.2.2 Résolution de certaines fonctions $\mathbb{N} \rightarrow \mathbb{N}$ définies récursivement

L'équation :

$$g(n) = a \cdot g\left(\frac{n}{b}\right) + n^k$$

admet pour solution :

$$g(n) = \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^{\ln_b(a)} \cdot \ln(n)) & \text{si } a = b^k \text{ c.a.d si } k = \ln_b(a) \\ \Theta(n^{\ln_b(a)}) & \text{sinon \% c.a.d si } a > b^k \end{cases}$$

preuve

Soit g la fonction définie par : $g(n) = a \cdot g\left(\frac{n}{b}\right) + n^k$. Pour tout entier n on a :

- $g(n) = n^k + a \cdot \left(g\left(\frac{n}{b}\right)\right)$.
- $g(n) = n^k + a \cdot \left(\left(\frac{n}{b}\right)^k + a \cdot g\left(\frac{n}{b^2}\right)\right)$.
- $g(n) = n^k \left(1 + \frac{a}{b^k} + \left(\frac{a}{b^k}\right)^2 + \dots + \left(\frac{a}{b^k}\right)^l\right)$ où l est l'entier vérifiant $\frac{n}{b^l} = 1$ c.a.d $\ln_b(n) = l$.

Or toute somme de la forme $1+r+\dots+r^l$ avec $l \in \mathbb{N}$ est égale à $\begin{cases} l+1 & \text{si } r = 1 \\ (r^{l+1} - 1)/(r - 1) & \text{sinon} \end{cases}$

et donc est égale à : $\begin{cases} \Theta(1) & \text{si } r < 1 \\ \Theta(l) & \text{si } r = 1 \\ \Theta(r^l) & \text{sinon} \end{cases}$. On en déduit :

1. si $a = b^k$, $g(n) = \Theta(n^{\ln_b(a)} \cdot \ln(n))$ puisque $k = \ln_b(a)$.
2. si $a > b^k$, $g(n) = \Theta(n^k \cdot \left(\frac{a}{b^k}\right)^{\ln_b(n)})$. Observant que $\left(\frac{a}{b^k}\right)^{\ln_b(n)}$ est égal à $\left(\left(\frac{a}{b^k}\right)^{\ln_{\frac{a}{b^k}}(n)}\right)^{\ln(\frac{a}{b^k})/\ln(b)}$ c'est à dire à $n^{\ln_b(a)-k}$, on obtient $g(n) = \Theta(n^{\ln_b(a)})$.

3. si $a < b^k$, $g(n) = \Theta(n^k)$.

Exercice 32 Pour réaliser le produit de deux entiers, une autre équation vérifiée par $a \cdot b$ est :

$$a \cdot b = (a_1 \cdot b_1) \cdot 2^n + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 2^{\frac{n}{2}} + a_2 \cdot b_2$$

Cette équation fournit un algorithme récursif.

1. Écrire cet algorithme en utilisant les fonctions utilisées dans `produit2`.
2. Écrire la définition récursive de la fonction complexité en temps.
3. Résoudre cette équation et calculer de fait cette fonction complexité.
4. Calculer la fonction complexité en espace.
5. Comparer ces différentes complexités avec celles des algorithmes solution de `Produit`.
6. Conclure en comparant cet algorithme aux autres solutions.

5.3 Un deuxième exemple : la multiplication de deux matrices

Nous pouvons nous inspirer du produit de deux entiers, pour réaliser le produit de deux matrices carrées :

problème `ProduitMat`

Entrée : deux matrices `X Y` carrées de même taille

Sortie : le produit matriciel de `X` et `Y`

Notons `Z` la matrice produite. Décomposant chacune des matrices carrées `X`, `Y` et `Z` de même taille supposée paire en des matrices `A, A, B, C, D, E, F, G, H, I, J, K, L`, de la façon suivante :

$$\begin{array}{r} \quad | \text{ A B } | \quad \quad | \text{ E F } | \quad \quad | \text{ I J } | \\ \text{ X = } | \text{ C D } | \quad \text{ Y = } | \text{ G H } | \quad \text{ Z = } | \text{ K L } | \end{array}$$

5.3.1 Première méthode

La première méthode découle des équations évidentes :

$$\text{I} = \text{AE} + \text{BG}$$

$$\text{J} = \text{AF} + \text{BH}$$

$$\text{K} = \text{CE} + \text{DG}$$

$$\text{L} = \text{CF} + \text{DH}$$

Exercice 33 En vous inspirant de la section traitant du produit de deux entiers :

1. Écrire un algorithme résolvant `ProduitMat` utilisant la méthode décrite plus haut.

2. Vérifier que l'équation vérifiée par la fonction complexité en temps est :
 $f(n) = 8f(\frac{n}{4}) + n$ où n est taille de la matrice (9 pour une matrice $3 \cdot 3$).
3. Vérifier que la solution de cette équation est $\Theta(n^{\frac{3}{2}})$.
4. Calculer la complexité en espace.
5. Conclure en comparant cet algorithme avec d'autres solutions.

5.3.2 Seconde méthode dite de Strassen

Reprenant les notations de la section précédentes, la seconde méthode découle des égalités suivantes :

$$\begin{array}{ll} P1 = A(G-H) & P5 = (A+D)(E-H) \\ P2 = (A+B)H & P6 = (B-D)(F+H) \\ P3 = (C+D)E & P7 = (A-C)(E+G) \\ P4 = D(F-E) & \end{array}$$

$$\begin{array}{l} I = P5 + P4 - P2 + P6 \\ J = P1 + P2 \\ K = P3 + P4 \\ L = P5 + P1 - P3 - P7 \end{array}$$

Exercice 34 1. Vérifier la correction des équations proposées.

2. En vous inspirant de la section traitant du produit de deux entiers, écrire un algorithme résolvant `ProduitMat` utilisant la méthode ci-dessus.
3. Vérifier que l'équation vérifiée par la fonction complexité en temps est :
 $f(n) = 7 \cdot f(\frac{n}{4}) + n$ où n est taille de la matrice (9 pour une matrice $3 \cdot 3$).
4. Vérifier que la solution de cette équation est $\Theta(n^{\ln_4 7}) \approx \Theta(n^{1,403})$.
5. Calculer la complexité en espace.
6. Conclure en comparant cet algorithme avec d'autres solutions.

5.3.3 Conclusion

Une solution algorithmique améliore l'algorithme de Strassen ($\Theta(n^{1,403})$) et fournit un algorithme de complexité $\Theta(n^{1,38})$ (Coppersmith & Winograd). Pour des raisons évidentes, le produit de deux matrices nécessite de lire chacun des éléments de la matrice et nécessite au moins $\Theta(n)$ opérations. Peut-on à l'image du produit de deux entiers (ou de deux vecteurs) atteindre cette limite $\Theta(n)$ ou sinon, s'en rapprocher davantage ? La question est ouverte.

Chapitre 6

Quelques éléments de Programmation Dynamique

Nous verrons dans ce chapitre comment gérer efficacement la mémoire de façon à éviter la répétition qui peut être exponentielle de certaines mêmes calculs.

Nous traiterons l'exemple d'un algorithme de complexité exponentielle transformé en un algorithme polynomiale.

Considérons l'algorithme suivant :

```
fonction toto(s:séquence d'entiers):entier

    si longueur(s) = 1 alors
        retourner premierElement(s)
    sinon
        retourner toto(extract1(s)) + toto(extract2(s)) ;
```

où `extract1` et `extract2` extraient deux sous-séquences de `s`.

Notons n la longueur de la séquence s . Supposons pour simplifier que la complexité de `extract1` et `extract2` est $\Theta(1)$. Dans le cas où `extract1(s)` et `extract2(s)` sont systématiquement de longueur $\leq \frac{n}{b}$ avec $b > 1$, nous nous trouvons face à un algorithme "Divide and Conquer" de complexité en temps polynomial car solution de $g(n) = 1 + 2 \cdot g(\frac{n}{b})$.

À contrario, Supposons par exemple que `extract1(s)` (resp. `extract2(s)`) retire un unique élément de `s` par exemple le premier (resp. le dernier). La fonction complexité en temps vérifie l'équation :

$$g(n) = 1 + 2 \cdot g(n - 1)$$

et est égale à $\Theta(2^n)$! L'algorithme est exponentiel et est impraticable (le traitement d'une séquence de 100 éléments nécessite un siècle et ce même si 1 milliard d'opérations élémentaires sont exécutées par seconde).

L'observation des calculs effectués par `toto` nous montre que l'appel sur la séquence (1,2,3,4,5,6) entraîne l'exécution de `toto` sur les séquences (1,2,3,4,5)

et (2,3,4,5,6). Ce qui provoquera récursivement 4 appels sur les séquences (1,2,3,4), (2,3,4,5), (2,3,4,5) et (3,4,5,6). Nous voyons ici que par deux fois `toto` est exécuté sur la séquence (2,3,4,5). Si on détaille le nombre de fois où les appels sont exécutés sur chacune des sous séquences nous obtenons le dessin suivant :

```

(1,2,3,4,5,6):1

(1,2,3,4,5):1    (2,3,4,5,6):1

(1,2,3,4): 1    (2,3,4,5):2    (3,4,5,6):1

(1,2,3):1    (2,3,4):3    (3,4,5):3    (4,5,6):1

(1,2):1    (2,3):4    (3,4):6    (4,5):4    (5,6):1

(1):1    (2):5    (3):10    (4):10    (5):5    (6):1

```

Il est facile d'observer que le nombre d'appels sur les séquences de longueur l est exponentiellement proportionnel à son opposé soit 2^{6-l} : 2⁵ appels sur les séquences de longueurs 1 ont été exécutés alors que 6 en tout auraient suffi.

L'idée de la programmation dynamique repose sur l'idée de réaliser un compromis espace-temps, c'est à dire d'éviter de répéter les mêmes calculs quitte, pour éviter ces répétitions, à utiliser une mémoire auxiliaire pour se rappeler des calculs effectués.

6.1 Une solution en programmation dynamique

Pour mémoriser les calculs, nous avons besoin de deux informations :

1. la première `déjàCalc` indique pour toute séquence si le calcul de `toto` a déjà été effectué.
2. la seconde `valeurs` indique pour toute séquence pour laquelle le calcul de `toto` a été effectué, la valeur de ce calcul.

Supposons que `extract1(s)` supprime le premier élément de la séquence `s` alors que `extract2(s)` supprime le dernier. Ici, toute sous-séquence extraite `t` d'une séquence initiale `s` peut être représenté par un couple d'entiers (i, j) formé du rang du premier élément de `t` relativement à `s` et du rang du dernier élément de `t` relativement à `s`. L'algorithme devient alors :

```

fonction totoDynamique(s:séquence): entier

    n ← longueur(s) ;
    déjàCalc ← matriceCarrée(n,faux()) ;

```

```

valeurs ← matriceCarrée(n,0) ;

(déjàCalc,valeurs) ← totoDynamRec(déjàCalc,valeurs,1,1) ;

retourner valeurs[1][n] ;

fonction totoDynamRec(déjàCalc,valeurs: matrice,i,j:entiers)
    : matrice × matrice

    si déjàCalc[i][j]
        retourner (déjàCalc,valeurs)

    sinon
        si i=j
            x ← ièmeElément(s,i) ;
        sinon
            (déjàCalc,valeurs) ← totoDynamRec(déjàCalc,valeurs,i,j-1) ;
            (déjàCalc,valeurs) ← totoDynamRec(déjàCalc,valeurs,i+1,j) ;
            x ← valeurs[i][j-1] + valeurs[i+1][j] ;

        déjàCalc[i][j] ← vrai() ;
        valeurs[i][j] ← x ;

    retourner (déjàCalc,valeurs)

```

Évaluation de la complexité de totoDynamique

Notons n la longueur de la liste s . La complexité en espace est lié à la taille des deux matrices `déjàCalc` et `valeurs` soit $\Theta(n^2)$ et à la taille maximal de la pile d'appel $\Theta(n)$ soit un total de $\Theta(n^2)$.

La complexité en temps est égal à $\Theta(n^2)$. Cette complexité est due :

- aux hypothèse faites en introduction qui supposaient les complexités de `extract1` et `extract2` constantes.
- à la complexité de la construction des deux matrices `déjàCalc` et `valeurs`.
- au fait que l'instruction `x<-valeurs[i][j-1]+valeurs[i+1][j]` est exécutée une unique fois pour tout couple (i, j) avec $i < j$. Ainsi, le nombre de fois où la fonction `totoDynamiqueRec` est appelé est exactement 1 plus deux fois le nombre de couples (i, j) avec $1 \leq i < j \leq n$ c'est à dire $1+(n-1) \cdot n = \Theta(n^2)$.

En conséquence, nous avons un algorithme quadratique $\Theta(n^2)$ alternative effective à un algorithme `toto` impraticable car de complexité exponentielle $\Theta(2^n)$.

6.1.1 Autre alternative

Dans l'exemple précédent, nous pouvons abandonner l'écriture récursive et dynamique et obtenir un meilleur algorithme qui organise itérativement le remplissage de la matrice `valeurs` en prenant des couples (i, j) de façon à faire croître $j - i$. Cette connaissance nous permet ainsi de faire l'économie de la matrice `déjàCalc` et de la pile d'appel induite par tout algorithme récursif. L'algorithme est :

```

fonction totoItératif(s:séquence): entier

    l ← longueur(s) ;
    valeurs ← matriceCarrée(l,0) ;

    pour i de 1 à l faire
        pour j de 1 à l faire

            si i=j alors
                valeurs[i][i] = ièmeElément(s,i) ;
            sinon
                valeurs[i][j] ← valeurs[i][j-1] + valeurs[i+1][j] ;

    retourner valeurs[1][n]

```

6.2 Bien fondé de l'approche dynamique

L'existence d'un algorithme itératif aussi simple que `totoItératif` est liée à notre capacité à prédire quelles sont les sous-séquences de `s` sur lesquelles le calcul doit être effectué (ici toutes les sous-séquences de la forme (s_i, \dots, s_j) avec $s = (s_1, \dots, s_n)$) et dans quel ordre elles doivent être évaluées (ici selon $j - i$ croissant). Le caractère polynomial de `totoItératif` était possible car l'ensemble de ces sous-séquences était de cardinalité polynomiale ici $\Theta(n^2)$.

Parfois cette connaissance est absente et seule une approche dynamique illustrée par `totoDynamique` est possible.

Supposons que `extract1` et `extract2` extraient de toute séquence `s` des sous-séquences selon des algorithmes complexes. Par exemple :

```

fonction extract1(s:séquence):séquence % notation s = (s1, ..., sn)

    si s1 + s2 < s3 + 3 alors
        retourner (s1, s3, ..., sn)
    sinon si s2 ≥ s4 alors
        retourner (s2, s4, ..., sn)
    sinon si etc

```

À cause d'une telle complexité, il est possible à priori que toute sous-séquence de s ait une évaluation nécessaire à celle de s . Or le nombre de sous-séquences de s est exponentiel $\Theta(2^n)$: par exemple la séquence $(1, 2, 3)$ de longueur 3 admet exactement $7 = 2^3 - 1$ sous-séquences : $()$, (1) , (2) , (3) , $(1, 2)$, $(1, 3)$, $(2, 3)$.

Il n'est alors pas possible d'évaluer toutes les sous séquences de s comme l'a réalisé `totoItératif`. Il nous faut évaluer que les sous-séquences réellement nécessaires, c'est à dire celles rencontrées lors du calcul récursif.

L'approche est celle utilisée par `totoDynamique` à la différence près qu'il faut reconsidérer la définition de `déjàCalc` et de `valeurs`. Sans rentrer dans les détails, nous pouvons supposer :

- `déjàCalc` est un ensemble de séquence dans lequel nous pouvons ajouter toute nouvelle séquence (fonction `ajouter`) et dans lequel nous pouvons décider de l'appartenance d'une séquence à cet ensemble (fonction `appartient`).
- `valeurs` est un ensemble de couples (t, v) formés d'une séquence t et d'un entier v dans lequel nous pouvons extraire pour une séquence donné t la valeur entière associée v dans l'ensemble `valeurs` (fonction `calcValeur`).

L'algorithme devient alors :

```

fonction totoDynamique2(s:séquence): entier

    (déjàCalc,valeurs) ← totoDynamRec2(ensVide(),ensVide(),s) ;

    retourner calcValeur(valeurs,s) ;

fonction totoDynamRecG(déjàCalc,valeurs: ensemble,s : séquence)
    : ensemble × ensemble

    si appartient(s,déjàCalc)
        retourner calcValeur(valeurs,s) ;

    sinon
        si longueur(s)=1
            x ← uniqueElément(s) ;
        sinon
            t ← extract1(s) ;
            u ← extract2(s) ;

            (déjàCalc,valeurs) ← totoDynamRec2(déjàCalc,valeurs,t) ;
            (déjàCalc,valeurs) ← totoDynamRec2(déjàCalc,valeurs,u) ;

            x ← calcValeur(valeurs,t) + calcValeur(valeurs,u) ;

        déjàCalc ← ajouter(déjàCalc,s) ;

```

```
valeurs ← ajouter(valeurs, (s, x)) ;
```

```
retourner (déjàCalc, valeurs)
```

La complexité de cet algorithme dépendra naturellement de ;

- de la longueur de la séquence initiale s .
- la cardinalité N de l'ensemble des sous-séquences dont l'évaluation est nécessaire à la séquence initiale s .
- de la complexité des opérations `calcValeur`, `appartient`, `ajouter`.

Une implémentation très naïve des ensembles `déjàCalc` et `valeurs` permet une complexité des primitives en $\Theta(N \cdot n)$, qui offre une complexité générale pour `totoDynamique2` égale à $\Theta(n \cdot N^2)$. Une meilleure implémentation (dans certains cas) offre des complexités pour les primitives égales à $\Theta(n)$ et donc une complexité générale égale à $\Theta(n \cdot N)$.

La complexité est principalement lié au nombre N de sous-séquences nécessairement évaluables pour évaluer s . Si ce nombre est polynomial, la programmation dynamique offre un algorithme (de complexité en temps) polynomial. Si ce nombre N est exponentiel, il n'existe, sauf cas singulier, aucune solution algorithmique en temps polynomial qu'elle soit dynamique ou autre.

Chapitre 7

Algorithme Glouton

L'approche "Divide and Conquer" construit une solution après avoir découpé l'entrée initiale en plusieurs sous-entrées et après avoir calculé les solutions partielles à chaque nouvelle sous-entrée. La construction de la solution se fait en fait après avoir parcouru toute l'entrée.

L'approche gloutonne diffère de celle-ci en construisant une partie de la solution de façon définitive.

7.1 Un premier exemple : le rendu de monnaie

Soit E l'ensemble des monnaies en euro. Le problème de la boulangère est de rendre une valeur S en euro en utilisant un nombre minimal de pièces.

problème RenduDeMonnaie

Entrée : S : entier, E : ensemble d'entiers

Sortie : une séquence d'entiers de E de somme S et de longueur minimale

L'algorithme s'écrit de façon récursive de la façon suivante :

```
fonction renduRec(S:entier ; E: ensemble d'entiers): séquence d'entiers
```

```
  si S=0
```

```
    retourner ()
```

```
  sinon
```

```
    calculer x la plus grande valeur de E inférieure ou égale à S ;
```

```
    retourner ajouter(x, renduRec(S-x))
```

ou ainsi de façon itérative :

```
fonction renduIter(S:entier ; E: ensemble d'entiers): séquence d'entiers
```

```
  solution ← séquenceVide() ;
```

```

tantque S ≠ 0 faire
    calculer x la plus grande valeur de E inférieure ou égale à S ;
    solution ← ajouter(x,solution) ;
    S ← S - x ;
retourner solution

```

Cet exemple illustre ce qu'est un algorithme glouton en ébauchant la solution sans prendre connaissance de la totalité de l'entrée : en effet, pour rendre la valeur de 17 centimes, seul le premier chiffre compte (le chiffre 1) et détermine de rendre une pièce de 10 centimes si S vaut $\{1, 2, 5, 10, 20, 50\}$.

7.1.1 Correction

La correction de cet algorithme est lié au choix de l'ensemble E .

Si l'ensemble des pièces européennes avait été $\{1, 4, 6\}$ l'algorithme aurait été incorrect : il associe à la valeur 8 la séquence $(6, 1, 1)$ qui n'est pas optimale, la solution optimale étant $(4, 4)$.

7.1.2 Amélioration

Pour des raisons de complexités en temps et en espace, une autre représentation de la séquence solution est préférable. En effet si E est égal à $\{1, 2, 5, 10\}$, la solution associée à la valeur 1000000 (resp. 10^{100}) est la séquence composée de 10000 (resp. 10^{99}) éléments égaux à 10. Cet algorithme est donc nécessairement exponentiel.

Une autre représentation peut se faire dans l'exemple $E = \{1, 2, 4, 10\}$ par un tableau indicé de 1 à 4 et indiquant le nombre de pièces correspondant. L'algorithme est :

```

fonction rendu(S:entier ; E: tableau d'entiers): tableau d'entiers
    solution ← tableau(4)(0) ;
    pour i de 1 à 4 faire
        solution[i] ← ⌊ $\frac{S}{E[i]}$ ⌋ ;
        S ← S - x ;
    retourner solution

```


7.2 Deuxième exemple : optimisation de la location d'une salle

Supposer que vous proposiez une salle à la location et ne pouvant être loué que pour un événement à la fois vous souhaitez maximiser le nombre d'événements; chaque événement e ayant une date de début d_e et une date de fin d_f .

Exemple 8 Dans le cas de trois événements $(1, 10)$, $(8, 13)$, $(12, 20)$, une solution optimale est de retenir les deux événements $(1, 10)$ et $(12, 20)$.

7.2.1 Première tentative

Un premier algorithme consiste à retenir en priorité les événements compatibles avec les événements déjà retenus qui sont de durée minimale :

fonction `reserv1(E:ensemble d'événements) : ensemble d'événements`

```
reste ← E ;
```

```
solution ← ensembleVide() ;
```

```
tantque vrai() faire
```

```
    tenter de choisir un événement e de date de durée minimale
        compatible avec solution ;
```

```
    si choix impossible
        retourner solution ;
```

```
    enlever e à reste ;
```

```
    ajouter e à solution ;
```

Cet algorithme est incorrect car il associe à l'ensemble des événements $\{(1, 10), (8, 13), (12, 20)\}$ le singleton $\{(8, 13)\}$ qui n'est pas optimal.

7.2.2 Deuxième tentative

Un second algorithme consiste à retenir en priorité les événements compatibles avec les événements déjà retenus qui ont une date de début minimale :

```

fonction reserv2(E:ensemble d'événements) : ensemble d'événements

    reste ← E ;

    solution ← ensembleVide() ;

    tantque vrai() faire

        tenter de choisir un événement e de date de début minimale
            compatible avec solution ;

        si choix impossible
            retourner solution ;

        enlever e à reste ;

        ajouter e à solution ;

```

Cet algorithme est incorrect car il associe à l'ensemble des événements $\{(1, 100), (2, 2), (3, 3), \dots, (10, 10)\}$ le singleton $\{(1, 100)\}$ qui n'est pas optimal la solution optimale étant $\{(2, 2), (3, 3), \dots, (10, 10)\}$.

7.2.3 Troisième tentative concluante

Un troisième algorithme consiste à retenir en priorité les événements compatibles avec les événements déjà retenus qui ont une date de fin maximale :

```

fonction reserv3(E:ensemble d'événements) : ensemble d'événements

    reste ← E ;

    solution ← ensembleVide() ;

    tantque vrai() faire

        tenter de choisir un événement e de date de début maximale
            compatible avec reste ;

        si choix impossible
            retourner solution ;

        enlever e à reste ;

        ajouter e à solution ;

```

On observe que cet algorithme associe à l'ensemble $\{(1, 10), (8, 13), (12, 20)\}$ la solution optimale $\{(1, 10), (12, 20)\}$ et à l'ensemble $\{(1, 100), (2, 2), (3, 3), \dots, (10, 10)\}$ la solution optimale étant $\{(2, 2), (3, 3), \dots, (10, 10)\}$.

Cet algorithme est correct, a une complexité en temps linéaire. Ces assertions sont laissés en exercice :

Exercice 35 Considérons l'algorithme `reserv3`. Supposons que l'ensemble des n événements soit représenté par deux tableaux d'entiers indicés de 1 à n et indiquant l'un la date de début (tableau `dateDeb`) et l'autre la date de fin (tableau `dateFin`).

1. Réécrire l'algorithme `reserv3` en utilisant et en observant la propriété suivante : un événement `e` qui n'a pas été retenu dans le contexte d'une solution courante `solution` ne peut être retenu après ; il peut ainsi être supprimé de l'ensemble `reste`. Pour cela vous devrez :
 - Indiquer comment représenter l'ensemble `reste`.
 - Indiquer comment représenter l'ensemble `solution`.
2. Évaluer la complexité de cette implémentation.
3. Prouver par récurrence la correction de l'algorithme `reserv3`.

7.3 Conclusion

Un grand nombre d'algorithmes optimaux du point de vue de la complexité en temps sont d'origine gloutonne, en particulier ceux fonctionnant en temps linéaires.

De nombreuses théorisations de cette approche gloutonne existent. La plus générale étant celle utilisant des matroïdes.

Chapitre 8

Quelques algorithmes de tri

Un problème classique en informatique est celui du tri :

problème Tri

Entrée : un ensemble d'entiers E

Sortie : une séquence triée composée de chacun des éléments de E

Notons que nous aurions pu supposer que l'ensemble en entrée était une partie d'un univers d'éléments muni d'un ordre total. Ce cas particulier fait aux entiers n'altère pas le caractère général des algorithmes que nous allons considérer.

Nous verrons comment à partir des deux méthodes vues dans les chapitres précédents résoudre un tel problème.

8.1 Approche gloutonne

Une approche gloutonne consiste à construire la séquence finale en en définissant un premier élément : clairement, il suffit d'extraire de l'ensemble E son plus petit élément x et de continuer. On déduit un premier algorithme récursif :

```
fonction triGloutonRécursif(E: ensemble):séquence
```

```
    si estVide(E) alors
```

```
        retourner sequenceVide()
```

```
    sinon
```

```
        (e,E) ← extraireMinimum(E) ;
```

```
        retourner ajouterEnDebut(e,triGloutonRécursif(E))
```

Cet algorithme se traduit immédiatement en l'algorithme itératif suivant :

```

fonction triGloutonItératif(E: ensemble):séquence

    Solution ← séquenceVide();

    tantque E n'est pas vide faire
        (e,E) ← extraireMaximum(E) ;
        Solution ← ajouterEnFin(e,Solution)
    ;

    retourner Solution

```

Cet algorithme très générique n'indique pas la façon dont sont représentés les ensembles ou les séquences.

8.1.1 Une première implémentation

Une première solution consiste à supposer que l'ensemble E est représenté à l'aide d'un tableau indicé de 1 à un indice n . L'ensemble courant sera représenté par ce même tableau et l'indice `indDeb` indiquant l'indice du premier élément (l'ensemble E étant formé des éléments indicés par les indices de `indDeb` à n). Ces choix ont pour conséquences immédiates l'algorithme suivant :

```

procédure triGlouton1(T:tableau d'éléments)
    pour indDeb de 1 à longueur(T)-1 faire
        début
            indMin ← indiceElementMin(T,indDeb) ;

            échanger(T,indDeb,indMin) ;
        fin

fonction indiceElementMin(T:tableau d'éléments ; indDeb : indice):indice

    indMin ← indDeb ;

    pour i de indDeb+1 à n faire
        si T[i] < T[indMin] alors
            indMin ← i ;

    retourner indMin

procédure échanger(T:tableau d'éléments ; i , j :éléments)
    début
        élémentAux ← T[i] ;
        T[i]← T[j] ;

```

```

    T[j] ← élémentAux ;
fin

```

L’algorithme possède quelques qualités :

1. il est de compréhension simple.
2. il est de complexité en espace constant. Excepté 3 ou 4 variables, nous n’avons pas besoin d’espace supplémentaire. Pour cela nous avons écrit, l’algorithme sous la forme d’une procédure. La solution qui est le tableau fourni en entrée est retournée par effet de bord.

Il possède une faiblesse. Sa complexité en temps est $\Theta(n^2)$ et n’est pas optimale.

8.1.2 Une deuxième implémentation à l’aide d’un tas

Il existe une seconde implémentation qui permet d’extraire le minimum d’un ensemble en $\Theta(\log(n))$ instructions. Puisque trier un ensemble consiste en l’extraction successive de ses n plus petits éléments, nous obtenons ainsi un algorithme de complexité en temps en

$$\Theta(n \log(n))$$

Malheureusement, cet algorithme nécessite de manipuler des arbres binaires qui ne rentrent pas dans le cadre de ce cours. Heureusement, ils font partie du cours prochain. Nous aurons donc l’occasion de présenter cet algorithme optimal en temps.

8.2 Approche “Diviser pour régner”

La méthode “Diviser pour régner” consiste à “décomposer” l’instance en deux (ou plusieurs) nouvelles instances, à appliquer récursivement l’algorithme de tri sur ces deux nouvelles sous-instances et à recomposer à partir des deux résultats partiels le résultat général.

L’algorithme de découpe vient très naturellement : il s’agit de partitionner un ensemble en deux parties. Un algorithme assez générique est alors :

```

fonction tri(E:ensemble):séquence

```

```

    (F,G) ← partition(E) ;

```

```

    retourner fusionner(tri(F),tri(G))

```

Notons dès à présent les propriétés souhaitées des algorithmes de partition et de fusion. Notons n la cardinalité de l’ensemble E . En supposant que l’on arrive à réaliser les calculs de partition et de fusion en au plus n opérations élémentaires

et à partitionner E en deux parties de cardinalité égales, la fonction complexité en temps (dans le pire des cas) f de `tri` vérifie :

$$f(n) = n + 2 \cdot f\left(\frac{n}{2}\right)$$

La fonction f est donc égale à $\Theta(n \ln(n))$.

Le résultat est alors optimal : nous ne connaissons pas d'algorithme de tri de complexité en temps inférieur à $\theta(n \ln(n))$.

Observons dès à présent, l'enjeu de décomposer E en deux parties de cardinalité proche. Si cette contrainte n'est pas respectée, et si l'on autorise de partitionner E en deux parties une ayant 1 élément et l'autre $n - 1$ éléments. L'équation devient :

$$f(n) \leq n + f(n - 1)$$

La fonction f est donc égale à $O(n^2)$.

Lors des différents algorithmes étudiés, nous représenterons les ensembles ainsi que les listes à l'aide de tableaux. D'autres algorithmes existent qui utilisent d'autres façons de représenter les ensembles à partir de listes, files ou arbres.

L'algorithme général de tri a pour définition :

```
fonction tri(T:tableau) :tableau
```

```
    triRec(T,1,longueur(T)) ;
```

et utilise une fonction auxiliaire `triRec` qui résout récursivement le problème suivant :

problème `TriRec`

Entrée : un tableau T , deux indices $1 \leq i \leq j \leq \text{longueur}(T)$

Sortie : un tableau U contenant les mêmes éléments que T tel que

$U[i..j]$ est trié

$U[i..j]$ contient les mêmes éléments que $T[i..j]$

(U est retourné par effet de bord)

8.2.1 Première solution algorithmique de `TriRec`

Le premier algorithme est basé sur une façon naturelle de décomposer un tableau : il suffit de calculer l'indice du milieu ($\lceil \frac{i+k}{2} \rceil$) sans déplacer les éléments.

procédure triRec1(T : tableau ; i,k : indices)

```

si i < k alors
  début
    j ← ⌈ $\frac{i+k}{2}$ ⌋ ;
    triRec1(T,i,j-1) ;
    triRec1(T,j,k) ;
    fusion(T,i,j,k) ;
  fin

```

La difficulté principale est ici de résoudre le problème :

problème Fusion

Entrée : un tableau T, trois indices $i \leq j \leq k$ tels que les sous-tableaux T[i..j-1] et T[j..k] sont triés.

Sortie : un tableau U contenant les mêmes éléments que T tel que

$$U[1..i-1] = T[1..i-1]$$

$$U[i..k] \text{ est trié}$$

$$U[k+1..n] = T[k+1..n]$$

(U est retourné par effet de bord)

L'algorithme fusion (Figure 8.1) utilise deux tableaux auxiliaires gauche et droit dans lesquels nous réalisons une copie des sous-tableaux T[i..j-1] et T[j..k]. Ces copies sont réalisées par la fonction copie.

Exercice 36 Écrire un algorithme itératif inspiré de triRec1. L'idée étant de fusionner les sous-tableaux de longueur 2, puis ceux de longueur 4, puis ceux de longueur 8. Évaluer la complexité en temps et en espace de celui-ci. Comparer ces complexités à celles de triRec1. Conclure.

Évaluation de la complexité

La complexité en temps de fusion est égal à $k-i+1$ la longueur du tableau symboliquement passé en argument. Ainsi, l'équation fournissant la complexité en temps de triRec1 est :

$$g(n) = 2 \cdot g\left(\frac{n}{2}\right) + n$$

La complexité en temps de triRec1 est ainsi optimale et est égale à :

$$\Theta(n \cdot \log(n))$$

La faiblesse de cet algorithme est l'opération copie qui utilise une mémoire auxiliaire de taille nécessairement la longueur du tableau copié.

Exercice 37 Expliquer pourquoi la complexité en espace de triRec1 est $\Theta(n)$.

Exercice 38 Expliquer pourquoi la complexité en espace de triRec1 serait $\Theta(n \log(n))$ si on supprimait les deux instructions liberer(gauche) et liberer(droit).

```

fonction fusion(T : tableau ; i0,j0,k0 : indices) : tableau
  début
    gauche ← copie(T,i0,j0-1) ;
    droit  ← copie(T,j0,k0) ;

    i ← 1 ;
    j ← 1 ;

    pour indice de i0 à k0 faire
      si j > longueur(droit) alors
        T[indice] ← gauche[i] ;
        i ← i + 1 ;
      sinon si i > longueur(gauche) alors
        T[indice] ← droit[j] ;
        j ← j + 1 ;
      sinon si gauche[i]<droit[j] alors
        T[indice] ← gauche[i] ;
        i ← i + 1 ;
      sinon
        T[indice] ← droit[j] ;
        j ← j + 1 ;

    libérer(gauche) ;
    libérer(droit) ;
  fin

```

FIG. 8.1 – L’algorithme fusion

8.2.2 Seconde solution algorithmique de TriRec

Le précédent algorithme consistait à :

1. en temps constant, décomposer un tableau, par simple calcul d’un indice.
2. en temps linéaire recomposer un tableau trié à partir des sous-tableaux tries.

Le second algorithme présenté ici consiste à :

1. en temps linéaire, décomposer le tableau.
2. en temps constant recomposer un tableau trié à partir des sous-tableaux triés.

Pour recomposer en temps constant un tableau trié à l’aide de deux sous-tableaux triés, une seule solution est possible : faire en sorte que les éléments du sous-tableau de gauche soient inférieurs ou égaux aux éléments du sous-tableau de droite.

Pour réaliser cela, on choisit dans le tableau initial un élément appelé *pivot* autour duquel cette décomposition se fera. Ces deux tâches sont dévolues à deux fonctions résolvant les deux problèmes suivants :

problème ChoixPivot

Entrée : un tableau T, deux indices $1 \leq i \leq j \leq \text{longueur}(T)$

Sortie : un indice indPivot appartenant à $[i, j]$

problème Distribuer

Entrée : un tableau T, trois indices $i, j, \text{indPivot}$ tels que :

$1 \leq i \leq \text{indPivot} \leq j \leq \text{longueur}(T)$

Sortie : un tableau U et un indice k tels que:

U contient les mêmes éléments que T

$U[1..i-1] = T[1..i-1]$

$U[j+1..n] = T[j+1..n]$

$U[k] = T[\text{indPivot}]$

$\forall a \in [i, k-1] \ U[a] \leq U[k]$

$\forall a \in [k+1, j] \ U[k] \leq U[a]$

(le tableau U est retourné par effet de bord)

La Figure 8.2 fournit la définition de l’algorithme récursif.

```
fonction triRec2(T:tableau; i,j : indices):tableau
```

```
  si i < j alors
    début
      indPivot ← choixPivot(T,i,j) ;
      k ← distribuer(T,i,j,indPivot) ;

      triRec2(T,i,k-1) ;
      triRec2(T,k+1,j) ;
    fin
```

FIG. 8.2 – Algorithme de tri récursif

Une solution algorithmique pour Distribuer

Une solution algorithmique au problème Distribuer est fournie par la Figure 8.3. Il est facile d’observer que la complexité en temps de distribuer est linéaire en la longueur du tableau T considéré (c’est à dire $p := j_0 - i_0 + 1$) soit $\Theta(j_0 - i_0)$.

```

fonction distribuer(T:tableau ; i0,j0, indPivot : indice)
    : indice

    si i0 = j0 alors
        retourner indPivot ;

    pivot ← T[indPivot] ;
    échanger(T,indPivot,j0) ;

    i ← i0 - 1 ;

    pour j de i0 à j0-1 faire
        si T[j] ≤ pivot alors
            i ← i + 1 ;
            échanger(T,i,j) ;

    échanger(T,i+1,j0) ;

    retourner i+1 ;

```

FIG. 8.3 – L’algorithme distribuer

8.2.3 Différentes variantes liées au choix du pivot

L’algorithme général `triRec2` admet plusieurs variantes qui dépendent de la façon de choisir le pivot. Rappelons que si la complexité en temps du choix du pivot (c’est le cas) n’excède pas celle de `distribuer` (en $\Theta(n)$ où n est la longueur du tableau symboliquement fourni en entrée) l’inéquation fournissant la complexité en temps dans le pire des cas est :

$$f(n) \leq n + f(n - 1)$$

Auquel cas, la complexité est majorée par n^2 . Si l’on souhaite obtenir une complexité faible, il nous faut partitionner l’ensemble en deux parties égales ce qui nous permet d’obtenir pour équation :

$$f(n) = n + 2 \cdot f\left(\frac{n}{2}\right)$$

et d’obtenir ainsi pour complexité $n \cdot \ln(n)$.

8.2.4 Le choix du premier pivot venu : le tri rapide

Le problème `ChoixPivot` admet pour première solution :

```

fonction choixPivot1(T:tableau ; i,j : indices) : indice
    retourner i

```

Point faible

Cette solution de complexité en temps $\Theta(1)$ a le désavantage d’offrir une complexité en temps dans le pire des cas pour `triRec2` égale à $\Theta(n^2)$.

Exercice 39 Pour tout entier n , notons T_n le tableau trié $(1, 2, \dots, n)$.

1. Démontrer que pour tout n et tous indices i et j , `distribuer(Tn,i,j,i)` retourne le couple (T_n, i) .
2. En déduire que l’équation définissant le nombre d’instructions élémentaires exécutées sur de tels tableaux est :

$$g(n) = 1 + g(n - 1) + n$$

3. En déduire que la complexité (dans le pire des cas) de `triRec2` est $\Theta(n^2)$.

Exercice 40 Quel est le nombre d’instructions élémentaires exécutées sur un tableau à éléments tous égaux ?

Point fort

Le premier point fort est la simplicité de l’algorithme ! Le second est que la complexité en temps en moyenne, notée $f(n)$, est $\Theta(n \log(n))$.

Pour calculer cette moyenne, on restreint l’étude à des tableaux dont les éléments définissent un intervalle de la forme $[1, n]$. La preuve est l’objet de l’exercice suivant :

Exercice 41 1. Démontrer que le nombre de tels tableaux de longueur n est exactement $n!$.

2. Démontrer que l’intégrale de $x \ln(x)$ est égale à $\frac{1}{2} \cdot x^2 \ln(x) - \frac{1}{4} \cdot x^2$. En déduire que $\sum_{i \in [1, n]} p \ln(p) = \Theta(n^2 \ln(n))$.
3. Démontrer que $f(n)$ vérifie l’équation :

$$f(n) = n + \frac{1}{n} \cdot \sum_{p \in [1, n-1]} (f(p) + f(n - p - 1))$$

Pour cela, utiliser le fait que l’ensemble des tableaux de longueur n se partitionne en n ensembles de mêmes cardinalités, ceux dont le premier élément est 1, deux dont le premier élément est 2 etc.

4. Résoudre cette équation et conclure.

Algorithme dit “tri rapide” : méfiez-vous de son nom !

Cet algorithme a pour nom, l’algorithme de tri rapide. Ce nom est un faux ami : en effet cet algorithme est de complexité en temps dans le pire des cas $\Theta(n^2)$, il est donc de ce point de vue bien moins rapide que le tri par tas ou le tri médian.

Cependant, il offre du point de vue de la complexité en moyenne un résultat optimal en $\Theta(n \log(n))$ équivalent à celui du tri par tas ou du tri médian (voir plus loin). Mieux une évaluation fine du nombre d’instructions élémentaires montre qu’il est plus rapide que le tri par tas : sa complexité de la forme $\alpha \cdot n \cdot \log(n)$ présente une constante multiplicative α bien inférieure à celle du tri médian ou du tri par tas. Pour cette raison bien concrète, il est préféré aux deux autres tris. Il est en fait possible de fournir une variante améliorée du tri rapide. C’est l’objet du point suivant.

8.2.5 Le choix d’un pivot aléatoire

Nous pouvons améliorer l’algorithme suivant en choisissant pour indice un indice pris aléatoirement dans l’intervalle $[i, j]$:

```
fonction choixPivot2(T:tableau ; i,j : indices) : indice
```

```
    retourner aléatoire(i,j) ;
```

La complexité en temps dans le pire des cas est conservée : il s’agit de $\Theta(n^2)$. La complexité en moyenne est conservée : il s’agit de $\Theta(n \ln(n))$. L’avantage est qu’ici pour tout tableau fourni en entrée (à valeurs deux à deux distinctes) l’espérance du nombre d’instructions est $\Theta(n \ln(n))$. Ainsi, pour tout tableau, la malchance d’avoir un traitement en $\Theta(n^2)$ instructions est faible.

Ces algorithmes aléatoires n’étant pas étudiés dans ce cours, nous ne retiendrons que cette idée : quand nous avons à choisir entre plusieurs options équivalentes, une bonne stratégie algorithmique est le choix aléatoire.

8.2.6 Le choix d’un pivot médian

Soit E un ensemble à n éléments, nous appelons *médian* le i ème plus petit élément avec $i := \lfloor \frac{n}{2} \rfloor$.

Nous avons vu précédemment, que l’on obtient une complexité en temps dans le pire des cas en $n \cdot \log(n)$ en partitionnant l’ensemble E initial en deux parties de cardinalité proche (cette condition est en fait non seulement suffisante mais aussi nécessaire voir Exercice 43). En d’autres termes il faut choisir pour pivot le médian de l’ensemble E .

Le calcul du médian est réalisé en résolvant le problème plus général suivant :
problème Sélection

Entrée : un ensemble E , un entier $i \in [1, \text{card}(E)]$

Sortie : le i -ième plus petit élément de E

En supposant que E représenté à l'aide d'un tableau T de longueur n , le calcul du i -ième plus petit élément de T se fait de la façon suivante :

1. en considérant comme bloc chacun des sous-tableaux de longueur 5 associés aux intervalles d'indice $[1, 5]$, $[6, 10]$, $[11, 16]$ etc, on trie chacun des blocs.
2. en considérant chacun des médians des sous blocs (aux indices 3, 8, 13 etc), on calcule le médian x (et son indice indx) de ces médians en utilisant de façon récursive **sélection**.
3. on exécute **distribuer** en prenant pour pivot ce médian. On obtient ainsi un tableau T partitionné autour du médian des médians x à l'indice noté k , avec aux indices de 1 à $k - 1$ des valeurs $\leq x$ et aux indices de $k + 1$ à n des valeurs $\geq x$.
4. – si $i = k$ on retourne x ,
– sinon si $i < k$ on calcule récursivement **sélection** sur l'entrée $(T[1, k-1], i)$
– sinon on calcule récursivement **sélection** sur l'entrée $(T[k+1, n], i-k)$.

Exercice 42 1. Démontrer que k appartient à l'intervalle $[\frac{3n}{10}, \frac{7n}{10}]$.

2. Démontrer que la fonction complexité en temps f de **sélection** vérifie l'inéquation :

$$f(n) \leq n + f\left(\frac{n}{5}\right) + f\left(\frac{7n}{10}\right)$$

3. Résoudre cette inéquation en démontrant que $\Theta(n \ln(n))$ en est solution.
4. Quelle serait la complexité de l'algorithme, si au lieu de considérer des blocs de 5, nous avons considéré des blocs de longueur :
 - (a) 3.
 - (b) 7.
5. Écrire l'algorithme.

Exercice 43 Dans l'algorithme **triRec2**, supposons qu'il existe un réel fixé $0 < \alpha \leq \frac{1}{2}$ tel que l'entier $\frac{k-i}{j-i+1}$ appartienne à l'intervalle $[\alpha, 1 - \alpha]$.

1. Que vaut α quand le pivot choisi est le médian ?
2. Écrire l'équation vérifiée par la fonction complexité en fonction de α ?
3. Résoudre cette équation.
4. Que pensez-vous de l'intérêt d'un tel algorithme lorsque $\alpha = \frac{1}{10}$, $\alpha = \frac{1}{10^{10}}$, $\alpha = \frac{1}{10^{10^{10}}}$?