Initiation à l'algorithmique

Denis Lapoire

12 octobre 2006

Table des matières

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 algorithme. 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 cous, 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 ambiguiité, un langage logique. La compréhension du problème nécessite la maîtrise de ce langage.

Un problème est une relation binaire liant deux objets. Un exemple de problème est :

```
problème Puissance
Entrée : un réel x ≠ 0, un entier n
Sortie : le réel x^n
```

Cette relation n'est pas nécessairement fonctionnelle, comme par exemple le problème suivant :

```
problème FacteurPremier
Entrée : un entier n > 1
Sortie : un facteur propre premier de n si il en existe,
         0 sinon.
```

2.2 Types

Les objets manipulés ici sont typés. Un type est en fait un ensemble d'opérations permettant de manipuler ces objets.

Le type le plus élémentaire est sans contestation possible le type booléen que l'on peut définir à l'aide de 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`.

D'autres types seront utilisés, comme le type entier. Les opérations fournies seront, sauf mention contraire, les opérations arithmétiques classiques, addition, multiplication, division euclidienne, etc .... Parfois, nous nous interrogerons sur la façon optimale d'implémenter ces fonctions : auquel cas, nous supposerons par exemple que seule l'addition est fournie et qu'il nous faut redéfinir par exemple la multiplication. Nous fournirons alors une description détaillée de l'opération additive fournie. Sa complexité en temps dépend des hypothèses (voir Section 4.3).

Un autre type est le type tableau. Les opérations sur ce type permettent de :

1. de construire un tableau. Il faut alors fournir sa longueur, une valeur d'initialiser chacun des éléments et son premier indice (si il n'est pas fourni, on supposera qu'il s'agit de 1).
   Ainsi `constructTab(10)(100)(1)` retourne un tableau de longueur 10 de valeurs toutes égales à 100 et de premier indice 1.
2. de calculer sa longueur. La fonction `longueur` retourne la longueur d'un tableau.
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]`.

2.3 Algorithme

Un exemple d'algorithme résolvant `Puissance` est :

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`.

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

début
    res ← 1 ;

    faire n fois
        res ← res · x ;

    retourner res
fin
```

FIG. 2.1 – L'algorithme `puissance` résolvant `Puissance`

2.3.1 Entête de l'algorithme

L'entête d'un algorithme fournit 4 informations :

1. la première est la nature de l'algorithme, fonction ou procédure ; dans l'exemple : `fonction`.

   Celle-ci est précisée par le mot-clef `fonction` ou `procédure`. Pour des raisons de simplicité, nous utiliserons quasi exclusivement des fonctions qui contrairement aux procédures retournent en fin de calcul un ou plusieurs objets typés. Ainsi, le résultat du calcul effectué devra être retourné (instruction `retourner` présentée plus bas).
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 exécuter 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 considérée.

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 ce type et de valeur 4.

Ainsi `r<-4.0` crée une variable de nom `r`, de type `réel` puisque 4.0 est de ce type 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'entrainera 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 $\leftarrow$ 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

```
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

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

```
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
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 `<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 ;

<instruction suivante> ;
```

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<-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 *tanque* 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 add(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 n est présent dans un tableau d'entiers :

```
problème Recherche
Entrée : un entier a, un tableau d'entier T
Sortie : vrai si il existe 1≤ i ≤ longueur(T) tel que a=T[i]
         faux sinon
```

Une première solution peut être :

```
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
```

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`.

Exercice 1 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 `tanque`. Dans un second temps, vous modifierez l'algorithme ainsi obtenu en le rendant plus simple et plus compréhensible.

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

début
    i ← 1 ;

    tant que (a<>T[i] ET i < longueur(T))
    début
        i ← i + 1 ;
    fin

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

FIG. 2.4 – Un second algorithme de recherche

```
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

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`.

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
              valeur de a : 10
instant t20   variables existantes : a , b
              valeur de a : 10
              valeur de b : 13
instant t30   variables existantes : a , b , c
              valeur de a : 10
              valeur de b : 13
```

Exemple 6 `fonction toto(a:entier):entier`

```
début
    a ← a + 3

    retourner a
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

```
valeur de c : 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 entrainé 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`.

Récursivité

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

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

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 différentes notions seront traités dans de prochains chapitres ainsi que dans d'autres cours dispensés à l'Enseirb. Lire la conclusion de ce chapitre à ce sujet.

2.3.14 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 2 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

Résoudre un problème consiste à écrire un algorithme. Nous dirons qu'un algorithme `A` résout un problème `P` si pour toute instance (entrée) x de `P`, la sortie y retournée par l'algorithme `A`, lorsque celui-ci termine, est une des sorties spécifiées par le problème `P`.

L'algorithme présente nécessairement différentes étapes de calcul. Ces étapes peuvent se situer à des niveaux différents. Il est alors important de savoir hiérarchiser ces niveaux.

Ainsi, par exemple, si au cours d'un algorithme il faut calculer le maximum de deux entiers `a` et `b` et affecter le résultat à une variable `c`, il est incongru d'écrire le code :

```
si a < b
    c ← b
sinon
    c ← a
```

Il faut écrire `c <- maximum(a,b)`, quitte à redéfinir la fonction `maximum`. On gagne en lisibilité, on évite d'éventuels erreurs, mais surtout on n'encombre pas la définition de l'algorithme général de calculs subalternes.

Ainsi, pour résoudre un problème, on écrit pas une seule fonction, mais de fait une fonction générale qui fait elle-même appel à d'autres fonctions auxiliaires, solutions elles-mêmes d'autres problèmes auxiliaires.

3.1 Un premier exemple

Si l'on souhaite résoudre le problème suivant :

```
problème ÉlémentMax
Entrée : un tableau T d'entiers non vide
Sortie : l'entier maximum des entiers de T
```

À l'algorithme :

```
fonction élémentMax1(T:tableau d'entiers):entier

    res ← T[1] ;

    pour i de 2 à longueur(T)
        si T[i] > res
            res ← T[i] ;

    retourner res
```

nous préférerons l'algorithme suivant :

```
fonction élémentMax2(T:tableau d'entiers):entier

    res ← T[1] ;

    pour i de 2 à longueur(T)
        res ← maximum(res,T[i]) ;

    retourner res
```

Il nous faudra alors préciser le problème résolu par `maximum` à savoir :

```
problème Maximum
Entrée : deux entiers a et b
Sortie : le maximum de a et b
```

et éventuellement rappeler la définition d'un algorithme résolvant ce problème. Cette définition a déjà été faite et est donc désormais inutile.

3.2 Un second exemple

Considérons le problème suivant :

```
problème FacteurProprePremier
Entrée : un entier a > 1
Sortie : un facteur propre premier de a si il en existe,
         0 sinon.
```

Une première solution consisterait à écrire :

```
fonction facteurProprePremier0(a:entier):entier

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

    retourner 0
```

Cette solution bien que concise est une mauvaise solution. Elle ne met pas en valeur les différents sous-problèmes associés à celui-ci et présente un algorithme très mauvais car très lent : nous verrons dans le chapitre que cet algorithme est en fait exponentiel.

Cette solution n'étant pas décomposée à l'aide de fonctions auxiliaires, vous ne savez pas réécrire cet algorithme si ce n'est en le réécrivant totalement :dit autrement, cet algorithme est à prendre ou à jeter entièrement. Nous ne le conserverons donc pas.

Avant de résoudre un problème, il faut comprendre celui-ci.

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* (voir Chapitre 1).

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 P !

3.3.1 Augmenter les contraintes en entrée

Une façon d'en extraire un plus simple est de limiter les instances. Ainsi, par exemple, le problème `FacteurProprePremier` induit comme nouveau problème si l'on se réduit aux entiers nombres non premiers :

`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.`

Rappelons qu'un entier est *premier* si il admet exactement deux diviseurs 1 et lui-même et est *composite* 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 avions 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 avions 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, entraineraient 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:entier) : mot

    tantque a<>1 faire
        si a est pair alors
            a ← a/2 ;
        sinon
            a ← (3·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 ;

    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 ;
    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

Trois hypothèses apparaissent :

1. La première façon de représenter un entier est de lui allouer un nombre d'octets fixe. Ce choix réalisé dans le langage de programmation `C` (un octet pour le type `char`, 4 octets pour le type `int`) a une limite : il ne permet de représenter que des entiers en nombre fini. Un entier positif représenté sur 4 octets peut prendre au plus $2^{4 \cdot 8}$ valeurs possibles : les entiers sur 4 octets décrivent l'intervalle $[0, 2^{32} - 1]$. Sous cette hypothèse, la complexité d'un entier est constante.

2. La seconde façon de représenter un entier n est de lui allouer autant d'octets que nécessite sa représentation, à savoir $\lceil \log_2(n+1) \rceil$ (que nous noterons $\log(n)$). Cette représentation est nécessaire quand les entiers utilisés peuvent prendre des valeurs réellement très grandes : c'est le cas d'applications cryptologiques qui manipulent des entiers représentés sur des centaines de bits.
3. Pour des raisons pédagogiques et de simplicité, nous supposerons souvent un type entier pouvant prendre n'importe quelle grande valeur et dotée d'opérations comme l'addition se réalisant en temps constant. Ce choix est un choix pédagogique mais n'admet aucune implémentation machine concrète.

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 manipulation de bits qu'il y en a de présent.

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:entier):entier

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

```
fonction factRec(a:entier):entier

    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:entier):entier

    si a = 0 alors retourner 0

    tant que estPair(a) faire
        a ← 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 considèrerons 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 moyens d'exécutions de $\frac{a}{2}$ est sur des entiers de taille 3 égal à $\frac{4}{7}$.

Exercice 3 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 algorithmiques, 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 $f : \mathbb{N}^* \to \mathbb{N}^*$ 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 $f : \mathbb{N}^* \to \mathbb{N}^*$ 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 $f : \mathbb{N}^* \to \mathbb{N}^*$ une fonction. Nous noterons $\Theta(f)$ l'ensemble des fonctions $g : \mathbb{N}^* \to \mathbb{N}^*$ 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 5). Pour cette raison, la notation $f \in \Theta(g)$ est remplacée par la notation $f = \Theta(g)$.

Exercice 4 Démontrer pour toutes fonctions $f, g : \mathbb{N}^* \to \mathbb{N}^*$ et tout réel $\alpha > 0$:

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

Exercice 5 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. $f \in \Theta(g)$ entraine $g \in \Theta(f)$.
3. $f \in \Theta(g)$ et $g \in \Theta(h)$ entraine $f \in \Theta(h)$.

4.6.2 Des fonctions étalons

Certaines fonctions $\mathbb{N}^* \to \mathbb{N}^*$ 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, ..., 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 6 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 : entier): entier

    i ← n ;

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

    retourner i
```

Supposons que `testInconnu` 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 `testInconnu`. La complexité en temps de `toto` est, par exemple, :

1. $\Theta(1)$ si `testInconnu` teste la parité.
2. $\Theta(\log(n))$ si `testInconnu` teste la primalité : la proportion de nombres premiers est $\frac{1}{\log(n)}$.
3. $\Theta(n)$ si `testInconnu` 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 $f : \mathbb{N}^* \to \mathbb{N}^*$ 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 7 Démontrer pour toutes fonctions $f, g : \mathbb{N}^* \to \mathbb{N}^*$ 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 8 Démontrer pour toutes fonctions $f, g : \mathbb{N}^* \to \mathbb{N}^*$:

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 9 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 $g \in O(h)$ entraine $f \in O(h)$.

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 10 Considérer l'algorithme suivant :

```
fonction f1(i : entier ):entier

    i ← f2(i) ;
    j ← f3(i) ;

    retourner i + j ;
```

Démontrer que :

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

Exercice 11 Considérer l'algorithme suivant :

```
fonction f1(i : entier ):entier

    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 choisirez 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 12 Considérer l'algorithme suivant :

```
fonction f1(n:entier):entier

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

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 le problème de la comparaison de deux matrices carrées de même taille :

```
fonction égal(A,B: matrice): 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();
```

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 $\Theta(n)$.

4.7.2 Un algorithme exponentiel

Considérons l'algorithme suivant.

```
fonction factIter(k:entier):entier

    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 13 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 14 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: entier)
```

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

Même question que l'exercice précédent.

4.8 Étude du problème de puissance

Considérons le problème de puissance d'un élément x selon une puissance k. Nous supposerons ici que toute multiplication est une instruction élémentaire (complexité en temps et en espace constante).

problème Puissance
Entrée : un élément x, un entier k
Sortie : x^k

Un premier algorithme est :

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

    res ← 1 ;

    faire k fois
      res ← res · x ;

    retourner res
```

Les complexités dans le pire des cas sont :

1. $\Theta(k)$ en temps.
2. $\Theta(1)$ en espace.

Un second algorithme est :

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

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

Les complexités dans le pire des cas sont :

1. $\Theta(k)$ en temps.
2. $\Theta(k)$ en espace dû à la pile d'appel.

Un troisième algorithme récursif utilisant la propriété $x^{2 \cdot k} = (x^2)^k$ est :

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

```
si (k=0) alors
    retourner 1 ;
sinon si estPair(k)
    retourner puissance2(x·x,k/2) ;
sinon
    retourner x·puissance2(x·x,(k-1)/2) ;
```

Clairement à chaque appel récursif, l'argument `k` est divisé par 2, un entier `k` entraine $\log(k)$ appels récursifs successifs. Les complexités dans le pire des cas sont :

1. $\Theta(\log(k))$ en temps.
2. $\Theta(\log(k))$ en espace dû à la pile d'appel.

Cet algorithme induit la version itérative suivante :

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

    l ← nombreBits(k) ;

    res ← 1 ;

    pour i de 1 à l faire
        res ← res · res ;

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

    retourner res
```

Dans cet algorithme,

- `nombreBits(k)` retourne le nombre de bits de la représentation binaire de l'entier `k` (c'est à dire $\lceil \log_n(n+1) \rceil$). Par exemple `nombreBits(6)` vaut 3 car la représentation binaire de 6 est 110.
- `ièmeBits(k,i)` retourne le ième bit à partir du bit de poids fort. Ainsi, `ièmeBits(6,1)`, `ièmeBits(6,2)` et `ièmeBits(6,3)` valent respectivement 1, 1 et 0.

Les complexités dans le pire des cas sont :

1. $\Theta(\log(k))$ en temps.
2. $\Theta(1)$ en espace.

Exemple 8 Si l'on souhaite calculer `puissance4(10,6)` les instructions successivement exécutées sont :

```
res ← 1 ;          % res = 1
res ← res · res ;  % res = 1
```

```
res ← x · res ;     % res = 10
res ← res · res ;   % res = 100
res ← x · res ;     % res = 1000
res ← res · res ;   % res = 1000000
```

Nous voyons sur cet exemple comment à partir de deux définitions différentes d'un même objet produire deux algorithmes (algorithmes `puissance1` et `puissance2`). Ces deux algorithmes ont pour avantage leur simplicité mais non leur complexité. Ensuite, nous voyons comment à traduire une propriété arithmétique en un algorithme récursif (`puissance3`) dont la correction est immédiate car liée étroitement à cette même propriété. L'algorithme produit est optimal en temps mais non en espace. En observant comment il organise les calculs, nous pouvons le "derécursiver" et obtenir un algorithme itératif optimal en temps et en espace (algorithme `puissance4`).

4.8.1 Importance des hypothèses

Nous avons démontré dans la section précédente, que le nombre d'exécutions de l'opération multiplication est égal à :

- pour `puissance1`, $\Theta(k)$.
- pour `puissance2`, $\Theta(k)$.
- pour `puissance3`, $\Theta(\log(k))$.
- pour `puissance4`, $\Theta(\log(k))$.

Hypothèse 1

Dans cette même section, l'étude de la complexité repose sur le fait que

1. les éléments sont représentés sur un nombre constant d'octets.
2. (et donc) que l'opération de multiplication a une complexité en temps constante.

Cette hypothèse est conforme au cas par exemple de réels représentés sur un nombre d'octets fixes (hypothèses réalistes). Les complexités en temps sont alors :

- pour `puissance1`, $\Theta(k)$.
- pour `puissance2`, $\Theta(k)$.
- pour `puissance3`, $\Theta(\log(k))$.
- pour `puissance4`, $\Theta(\log(k))$.

Remarque 1 La représentation d'un réel x dans le langage `C` (type **float** norme IEEE 754) est normalisé sous la forme d'un triplet (α, β, γ) représenté sur 4 octets vérifiant l'équation

$$x = \alpha \cdot \beta \cdot 2^{\gamma}$$

où :

1. α est le *signe* -1 ou 1 codé sur un bit.

2. β est la *mantisse*, une fraction réelle de l'intervalle $[1, 2[$ codée sur 23 bits.
3. γ est l'*exposant*, un entier codé sur 8 bits appartenant à $[-126, 127]$

Cependant si vous modifiez les hypothèses, si vous supposez que la complexité de la multiplication n'est pas constante mais est par exemple proportionnelle à la taille de l'élément multiplié y ($\Theta(\texttt{taille}(y))$) vos conclusions différeront.

Hypothèse 2

Ici, nous supposons que :

1. la complexité en temps de l'addition dépend de la complexité des éléments.
2. le produit de deux éléments de même taille conserve cette même taille

Cette hypothèse est notamment vérifiée dans le cas du produit de deux matrices carrées (à éléments de taille constante). Les complexités en temps sont alors :

- pour `puissance1`, $\Theta(\texttt{taille}(x) \cdot k)$.
- pour `puissance2`, $\Theta(\texttt{taille}(x) \cdot k)$.
- pour `puissance3`, $\Theta(\texttt{taille}(x) \cdot \log(k))$.
- pour `puissance4`, $\Theta(\texttt{taille}(x) \cdot \log(k))$.

Sous cette nouvelle hypothèse les deux derniers algorithmes sont encore bien meilleurs que les deux premiers.

Hypothèse 3

Ici, nous supposons que :

1. la complexité en temps de l'addition dépend de la complexité des éléments.
2. le produit de deux éléments a pour taille la somme des deux tailles.

Cette hypothèse est notamment vérifiée dans le cas du produit de deux entiers à taille variable. Les complexités en temps sont alors :

- pour `puissance1`, $\Theta(\texttt{taille}(x^k)) = \Theta(k \cdot \texttt{taille}(x))$.
- pour `puissance2`, $\Theta(\texttt{taille}(x^k)) = \Theta(k \cdot \texttt{taille}(x))$.
- pour `puissance3`, $\Theta(\texttt{taille}(x^k)) = \Theta(k \cdot \texttt{taille}(x))$.
- pour `puissance4`, $\Theta(\texttt{taille}(x^k)) = \Theta(k \cdot \texttt{taille}(x))$.

Sous cette dernière hypothèse, les algorithmes ont même complexité en temps !

Exercice 15 Évaluer la complexité en espace des différents algorithmes dans chacune des deux nouvelles hypothèses.

Exercice 16 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 17 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 $x_1, \ldots, x_a$.
2. a appels récursifs. Ceux-ci consistent à appliquer récursivement la fonction A sur chacune des nouvelles entrées $x, \ldots, x_a$ et à retourner les a solutions $s_1, \ldots, s_a$.
3. la recomposition. Celle-ci consiste à recomposer à partir des solutions partielles $s_1, \ldots, s_a$ 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. Nous sommes confrontés au problème suivant :

```
Problème Produit
Entrée : a, b : entier
Sortie : a·b
```

Ici, nous considérons comme opérations élémentaires, 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 `n>>1`) ou d'insertion d'un nouveau bit de poids faible (multiplication par 2 noté `n<<1`).

5.1.1 Un premier algorithme

Un premier algorithme est le suivant :

```
fonction produit(a,b: entier): entier

    res ← 0 ;

    tantque b ≠ 0 faire
        si estPair(b)
            res ← addition(res,a) ;

        a ← a << 1 ;
        b ← b >> 1 ;

    retourner res ;
```

Cet algorithme utilisant la fonction `addition` de complexité linéaire en la taille des entrées `a` et `b`, la complexité de `produit` est quadratique car étant exactement égal à $\Theta(\texttt{taille}(a) \cdot \texttt{taille}(b)) = \Theta(n^2)$ avec $n := \texttt{taille}(a) + \texttt{taille}(b)$ (on rappelle que la taille d'un entier a est le nombre de bits de se représentation binaire $\approx \log(a)$).

Cstret algorithme est quadratique. Peut-on faire mieux ?

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.

5.1.2 Un second algorithme

Notons n la taille maximale de a et b. 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 bine 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 entrée. Notons s_1, s_2 et s_3 les produits associés.

3. recomposer le résultat final s à l'aide de ces 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 produit2(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=n/2

    (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·2^p
    res ← addition(res, s2<<n) ;
    res ← soustraction(res, s2<<p) ;
    res ← addition(res, s3) ;
    res ← soustraction(res, s3<<p) ;

    retourner res
```

5.1.3 Évaluation de la complexité

Notons $f : \mathbb{N} \to \mathbb{N}^*$ 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

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

Nous verrons comment résoudre un tel système d'équations. Nous pouvons ici le résoudre à la main. La solution est $f(n) = n+3(\frac{n}{2}+3(\frac{n}{2^2}+\ldots)) = n((\frac{3}{2})^0+(\frac{3}{2})^1+\ldots) \approx n.(\frac{3}{2})^{\ln_2(n)})$ 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 :

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

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 :

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

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

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

Exercice 18 Démontrer que les fonctions f et g vérifiant les deux systèmes d'équation précédent vérifient $f = \Theta(g)$ et donc $g = \Theta(f)$.

5.3 Résolution de certaines fonctions $\mathbb{N} \to \mathbb{N}$ définies récursivement

Il n'existe pas de méthode générale pour résoudre une équation. Les équations que nous proposons de résoudre sont de la forme :

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

Si votre définition récursive n'est pas de cette forme, il faut tenter de s'y ramener par tous les moyens. Souvent, les moyens présentés dans la section précédente suffisent. Parfois, nous avons une inéquation de la forme : $f(n) \leq a \cdot f(\frac{n}{b}) + n^k$. Auquel cas, la conclusion que l'on peut en tirer est $f(n) = O(g(n))$ avec g solution de l'équation précédente.

La solution de l'équation est égale à :

1. $g(n) = \Theta(n^{\ln_b(a)} \cdot ln(n))$ si $a = b^k$ c.a.d si $k = \ln_b(a)$.
2. $g(n) = \Theta(n^{\ln_b(a)})$ si $a > b^k$.
3. $g(n) = \Theta(n^k)$ si $a < b^k$.

preuve

Soipt g la fonction définie par : $g(n) = a \cdot g(\frac{n}{b}) + n^k$. Pour tout entier n on a :
- $g(n) = n^k + a \cdot (g(\frac{n}{b}))$.
- $g(n) = n^k + a \cdot ((\frac{n}{b})^k + a \cdot g(\frac{n}{b^2}))$.
- $g(n) = n^k(1 + \frac{a}{b^k} + (\frac{a}{b^k})^2 + \ldots + (\frac{a}{b^k})^{\ln_b(n)})$.

L'observation que toute somme de la forme $1 + r + \ldots + r^l$ définie à partir d'un terme r constant est égale :
- à $l + 1$ donc à $\Theta(l)$ si $r = 1$.
- à $(r^{l+1} - 1)/(r - 1)$ donc à $\Theta(r^l)$ si $r > 1$.
- à $(r^{l+1} - 1)/(r - 1)$ donc à $\Theta(1)$ si $r < 1$.

permet de conclure à :

1. si $a = b^k$, $g(n) = \Theta(n^{\ln_b(a)} \cdot ln(n))$ puisque $k = \ln_b(a)$.
2. si $a > b^k$, $g(n) = \Theta(n^k \cdot (\frac{a}{b^k})^{\ln_b(n)})$. Observant que $(\frac{a}{b^k})^{\ln_b(n)}$ est égal à $((\frac{a}{b^k})^{\ln_{\frac{a}{b^k}}(n)})^{ln(\frac{a}{b^k})/ln(b)}$ 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 19 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.

5.4 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 :

```
     | A B |        | E F |        | I J |
 X = | C D |    Y= | G H |    Z= | K L |
```

5.4.1 Première méthode

La première méthode découle des équations évidentes :

```
I = AE + BG
J = AF + BH
K = CE + DG
L = CF + DH
```

Exercice 20 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 a ce même problème.

5.4.2 Seconde méthode dite de Strassen

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

```
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)

I = P5 + P4 - P2 + P6
J = P1 + P2
K = P3 + P4
L = P5 + P1 - P3 - P7
```

Exercice 21

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 décrite plus haut.
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 a ce même problème.

5.4.3 Conclusion

Une solution algorithmique améliore l'algorithme de Strassen ($\Theta(n^{1,403})$ et fournit un algorithme de complexité $\Theta(n^{1.138})$ (Coppersmith & Winogend). 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

Programmation Dynamique

Considérons l'algorithme suivant :

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

    si longueuer(s) = 1 alors
        retourner s_1
    sinon
        retourner toto(extract1(s)) + toto(extract2(s)) ;
```

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

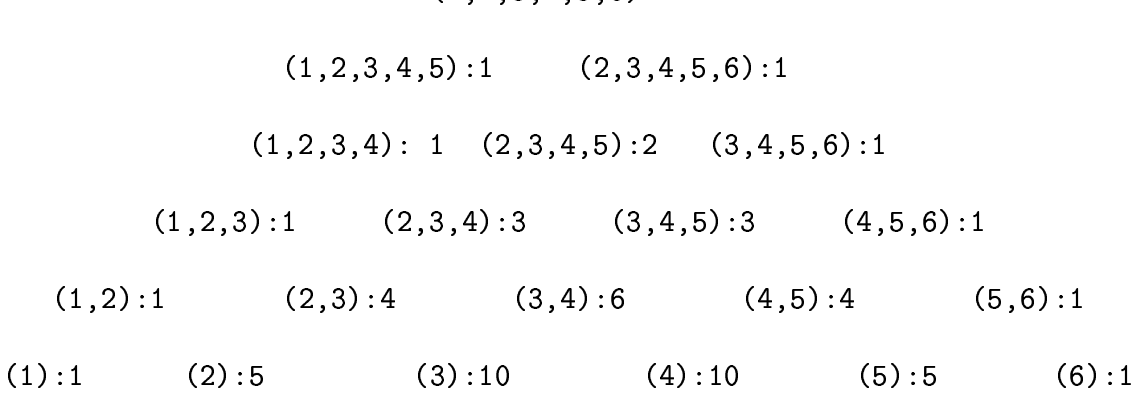
Notons n la longueur de la séquence s. Supposons pour simplifier que la complexité de `extract1` et `extract2` est $\Theta(1)$. Dans le cas où `extract1(s)` et `extract2(s)` sont systématiquement de longueur $\leq \frac{n}{b}$ avec $b > 1$, nous nous trouvons face à un algorithme "Divide and Conquer"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



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} : 2^5 appels sur les séquences de longueurs 1 ont été exécutés alors que 6 en tout auraient suffit.

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 de 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,l) ;

    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 dûe :

- 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 et 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 $(s_i, \ldots, s_j)$ avec $s = (s_1, \ldots, s_n)$) 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 découper l'entrée initiale en plusieurs sous-entrées et après avoir calculer 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] ← ⌊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 souhaitiez maximiser le nombre d'événements ; chaque événement e ayant une date de début d_e et une date de fin d_f.

Exemple 9 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 tentavive

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 tentavive

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 $\{(1,100),(2,2),(3,3),\ldots,(10,10)\}$ le singleton $\{(1,100)\}$ qui n'est pas optimal la solution optimale étant $\{(2,2),(3,3),\ldots,(10,10)\}$.

7.2.3 Troisième tentavive 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 $\{(1,100),(2,2),(3,3),\ldots,(10,10)\}$ la solution optimale étant $\{(2,2),(3,3),\ldots,(10,10)\}$.

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

Exercice 22 Considérons l'algorithme `réserv3`. 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 mous 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 triGloutonRecursif(E: ensemble):séquence

    si estVide(E) alors
        retourner sequenceVide()
    sinon
        (e,E) ← extraireMinimum(E) ;
        retourner ajouterEnDebut(e,triGloutonRecursif(E))
```

Cet algorithme se traduit immédiatementen 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.

8.2 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`).

Extraire l'élément minimum de `E` consistant à parcourir l'intervalle [`indDeb`,`n`], calculer l'indice `indMin` du plus petit élément, échanger les éléments aux indices `indDeb` et `indMin` puis incrémenter `indDeb`. L'algorithme est ainsi :

```
fonction triGlouton1(T:tableau):tableau

    n ← longueur(T) ;
    indDeb ← 1 ;

    pour indDeb de 1 à n-1 faire

        indMin ← indDeb ;

        pour i de indDeb+1 à n faire

            si T[i] < T[indMin] alors
                indMin ← i ;

        T ← échanger(T,indDeb,indMin) ;

    retourner E
```

L'algorithme a pour complexité en espace $\Theta(1)$ et pour complexité en temps $\Theta(n^2)$.

8.3 Une deuxième implémentation à l'aide d'un tas

Ici, nous implémenterons nons pas l'algorithme qui extrait le minimum des éléments restants mais son algorithme dual qui considère les éléments maximums :

```
fonction triGloutonRecursif2(E: ensemble):séquence

    si estVide(E) alors
        retourner sequenceVide()
    sinon
        (e,E) ← extraireMaximum(E) ;
        retourner ajouterEnFin(e,triGloutonRecursif(E))


fonction triGloutonItératif2(E: ensemble):séquence

   Solution ← séquenceVide();

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

   retourner Solution
```

La complexité de l'algorithme est liée à la complexité de l'extraction du minimum. Si nous choisissons de représenter l'ensemble `E` par une séquence sous forme de tableau l'extraction est en $\Theta(n)$, la complexité globale est en $\Theta(n^2)$.

Une solution très rapide existe qui utilise un tas.

8.3.1 Définition de haut niveau d'un tas

Un tas peut être défini comme un ensemble dans lequel :

1. on peut extraire le plus grand élément.
2. on peut ajouter tout nouvel élément.

Ainsi, le tas est un ensemble dégradé dans lequel on ne peut extraire d'autre élément que le plus grand ! Nous verrons que cette restriction permet d'obtenir pour chacune de ces opérations une complexité en espace constante et en temps égale à $\Theta(\log(n))$ où n est la cardinalité de l'ensemble considéré.

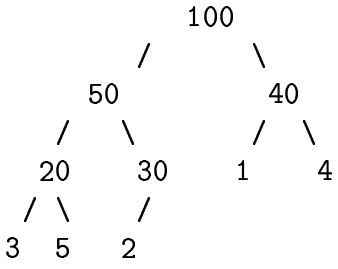
8.3.2 Implémentation d'un tas

Pour implémenter un tas, on utilise un *arbre binaire* c'est à dire un objet qui contrairement à l'objet mathématique unidimensionnelle qu'est la séquence (dessinée sur une ligne) est un objet bidimensionnel (dessinée sur une page).

Un arbre est un ensemble d'éléments qui admettent certains un *fils-gauche* et/ou un *fils-droit*. L'élément qui n'a pas de père est appelé la *racine*. Un tas est un arbre dans lequel :

1. l'élément fils-gauche (resp. fils-droit) y d'un élément x vérifie $y \leq x$.
2. tous les niveaux de l'arbre sont remplis excepté éventuellement le niveau le plus bas qui cependant ne doit pas laissé de "trou" dans sa partie gauche (voir exemple ci-dessous).

Exemple 10 Ainsi, l'ensemble $\{100, 40, 50, 20, 30, 1, 4, 3, 5, 2\}$ peut être représenté par l'arbre suivant :

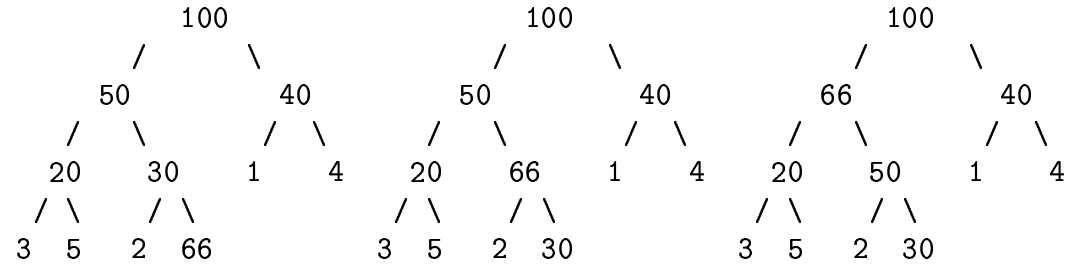


Cet arbre a pour racine `100`. Le fils gauche de `100` est `50`, son fils droit est `40` qui a pour fisl gauche `1` et fils droit `4`.

8.3.3 Ajouter un élément dans un tas

Pour ajouter un nouvel élément y dans un tas représenté par un arbre, il suffit de le placer au plus bas niveau et le "faire remonter" de façon à respecter la hiérarchie père fils. Cet algorithme est illustré par l'example ci-dessous et sera formellement écrit dans la section suivante.

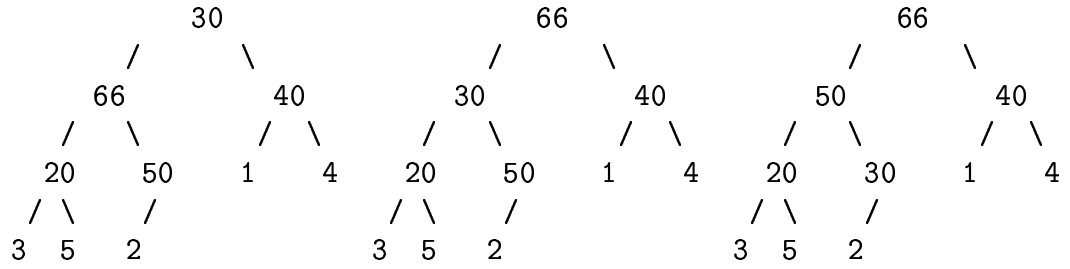
Exemple 11 Ajouter l'élément `66` dans le tas de l'exemple précédent consiste à le placer au niveau inférieur à droite de l'élément le plus à droite et à le faire remonter itérativement si il supérieur à son père :



8.3.4 Extraire le maximum dans un tas

Pour extraire le maximum d'un tas représenté par un arbre, il suffit de déplacer à la racine l'élément le plus à droite sur le dernier niveau, puis de faire "descendre" cet élément dans l'arbre de façon à respecter la hiérarchie père fils. Cet algorithme est illustré par l'exemple ci-dessous et sera formellement écrit dans la section suivante.

Exemple 12 Supprimer le maximum dans le tas de l'exemple précédent consiste à déplacer à la racine 30 et à le faire descendre dans l'arbre en échangeant l'élément courant éventuellement avec le maximum de son fils gauche et de son fils droit.



8.3.5 Implémentation d'un tas-arbre à l'aide d'un tableau

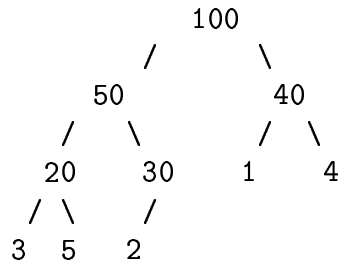
Une représentation simple et optimale d'un tas est l'utilisation d'un tableau dans lequel :

1. la racine est à l'indice 1
2. le fils gauche d'un élément à l'indice i a pour indice $2 \cdot i$.
3. le fils droit d'un élément à l'indice i a pour indice $2 \cdot i + 1$.

Exemple 13 Ainsi, le tableau suivant :

1	2	3	4	5	6	7	8	9	10
100	50	40	20	30	1	4	3	5	2

fournit une représentation du tas :



On observe que le fils-gauche de l'élément 50 (indice 2) est l'élément 20 (indice 4). On observe que le fils-droit de l'élément 50 (indice 2) est l'élément 30 (indice 5).

8.3.6 Écriture de l'algorithme

L'ensemble E sur lequel le tri est effectué est représenté à l'aide d'un tableau. Le tas utilisé dont la cardinalité varie au cours de l'algorithme est en fait constitué par la partie gauche du tableau borné par un indice `cardTas`. L'algorithme est :

```
fonction triParTas(t:tableau):tableau

   n ← longueur(t) ;

   pour i← 2 à n
       t ← ajouterTas(t,i) ;

   pour i← n à 2
       t ← échanger(t,1,i) ;
       t ← descenteTas(t,i-1) ;

   retourner t
```

où les deux fonctions auxiliaires résolvent les deux problèmes définis par les Figures 8.1 et 8.2.

```
problème AjouterTas
Entrée : un tableau t, un indice i tel que t[1..i-1] soit un tas
Sortie : un tableau u tel que
              u[1..i] est un tas
              u[1..i] contient les mêmes éléments que t[1..i]
              u et t  contiennent les mêmes éléments
```

FIG. 8.1 – Le problème `AjouterTas`

```
problème DescenteTas
Entrée : un tableau t, un indice i tel que t[1..i] soit un tas
         excepté en ce qui concerne la racine dont chacun des fils
         peut lui être supérieur
Sortie : un tableau u tel que
              u[1..i] est un tas
              u[1..i] contient les mêmes éléments que t[1..i]
              u[i+1..n] = t[i+1..n] avec n=longueur(t)
```

FIG. 8.2 – Le problème `DescenteTas`

8.3.7 Évaluation de la complexité

Comme cela sera établi dans les deux sections suivantes, les deux fonctions auxiliaires `ajouterTas` et `descenteTas` ont une complexité en espace en $\Theta(1)$ et une complexité en temps dans le pire des cas en $\Theta(\log(i))$. On en déduit que `triParTas` a une :

1. complexité en espace de $\Theta(1)$. L'espace mémoire utilisé est, excepté quelques variables pour gérer des indices, celui du tableau fourni en entrée. Il s'agit donc tri "sur place".
2. complexité en temps dans le pire des cas égale à $\Theta(n\log(n))$. Du fait de l'égalité $\Theta(n\log(n)) = \sum_{i\in[1,n]} i\log(i)$.

Pour ces deux raisons, cet algorithme de tri est optimal.

Écriture de `ajouterTas`

Une solution à `AjouterTas` est l'algorithme suivant illustré par l'exemple 11 :

```
fonction ajouterTas(t:tableau,cardTas:indice):tableau

    i ← cardTas ;

    tantque i>1 ET t[i]>t[⌊i/2⌋] faire
        t ← échanger(t,i,⌊i/2⌋) ;
        i ← ⌊i/2⌋ ;

    retourner t
```

La complexité en espace de cet algorithme est $\Theta(1)$ et en temps (dans le pire des cas) en $\Theta(\log(n))$ avec $n :=$ `cardTas` : en effet à chaque passage de boucle i est au moins divisé par deux. Il faut donc au plus $\log(n)$ passages de boucles pour que `i > 1` soit évalué à faux.

Exercice 23 En reprenant l'exemple du tas de l'Exemple 11, fournir les états différents du tableau représentant ce tas au cours de l'algorithme `ajouterTas`.

Écriture de `descenteTas`

Une solution à `DescenteTas` est l'algorithme défini par la Figure 8.3 et illustré par l'Exemple 12 :

La complexité en espace de cet algorithme est $\Theta(1)$ et en temps (dans le pire des cas) en $\Theta(\log(n))$ avec $n :=$ `cardTas` : en effet à chaque passage de boucle i est au moins multiplié par deux. Il faut donc au plus $\log(n)$ passages de boucles pour que `2.i > cardTas` soit vérifié.

Exercice 24 En reprenant l'exemple du tas de l'Exemple 12, fournir les états différents du tableau représentant ce tas au cours de l'algorithme `descenteTas`.

```
fonction descenteTas(t:tableau,cardTas:indice):tableau

    i ← 1 ;

    tantque vrai() faire

        si 2.i > cardTas alors
           retourner t

        sinon si (2.i+1>cardTas) ET t[2.i]≤t[i] alors
           retourner t

        sinon si (2.i+1>cardTas) alors
           t ← échanger(t,i,2.i) ;
           i ← 2.i ;

        sinon si t[2.i] ≥ t[2.i+1] ET t[2.i]>t[i] alors
           t ← échanger(t,i,2.i) ;
           i ← 2.i ;

        sinon si t[2.i+1] ≥ t[2.i] ET t[2.i+1]>t[i] alors
           t ← échanger(t,i,2.i+1) ;
           i ← 2.i+1 ;

        sinon
           retourner t
```

FIG. 8.3 – un algorithme de descente

8.4 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 :

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

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

    retourner 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≤i≤j≤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]
```

8.4.1 Première solution algorithmique de `TriRec`

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

```
fonction triRec1(T : tableau ; i,k : indices) : tableau

    si i = k
        retourner T ;

    j ← ⌈(i+k)/2⌉ ;

    T ← triRec1(T,i,j-1) ;
    T ← triRec1(T,j,k) ;

    T ← fusion(T,i,j,k) ;

    retourner T
```

La difficulté principale est ici de résoudre le problème :

```
problème Fusion
Entrée : un tableau T, trois indices i ≤ j ≤ 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] est trié
             U[k+1..n] = T[k+1..n]
```

L'algorithme `fusion` (Figure 8.4) 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 25 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 :

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

La complexité en temps de `triRec1` est ainsi $\Theta(n \cdot \log(n))$.

La faiblesse de cet algorithme est l'opération `copie` qui utilise une mémoire auxiliaire de taille $\Theta(n)$.

```
fonction fusion(T : tableau ; i0,j0,k0 : indices) : tableau

    gauche ← copie(T,i0,j0-1) ;
    droit  ← copie(T,j0,k0) ;

    i ← 1 ;
    j ← 1 ;

    pour indice de i0 à k0 faire

        si (i≤longueur(gauche)) ET (j≤longueur(droit))
                                ET (gauche[i]<droit[j]) alors
            T[indice] ← gauche[i] ;
            i ← i + 1 ;

        sinon si (i≤longueur(gauche)) ET (j≤longueur(droit)) alors
            T[indice] ← droit[j] ;
            j ← j + 1 ;

        sinon si (i≤longueur(gauche)) alors
            T[indice] ← gauche[i] ;
            i ← i + 1 ;

        sinon
            T[indice] ← droit[j] ;
            j ← j + 1 ;

    retourner T
```

FIG. 8.4 – L'algorithme `fusion`

8.4.2 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 ≤ i ≤ j ≤ longueur(T)
Sortie : un indice indPivot appartenant à [i,j]

problème Distribuer
Entrée : un tableau T, trois indices i,j,indPivot tels que :
                1 ≤ i ≤ indPivot ≤ j ≤ 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[indPivot]
              ∀a∈[i,k-1] u[a]≤u[k]
              ∀a∈[k+1,j] u[k]≤u[a]
```

La Figure 8.5 fournit la définition de l'algorithme récursif.

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

    si i≥ j alors
        retourner T
    sinon

        indPivot ← choixPivot(T,i,j) ;

        (T,k) ← distribuer(T,i,j,indPivot) ;

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

        retourner T
```

FIG. 8.5 – Algorithme de tri récursif

Une solution algorithmique pour `Distribuer`

Une solution algorithmique au problème `Distribuer` est fourni par la Figure 8.6.

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

    si i0 = j0 alors
        retourner (T,indPivot) ;

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

    i ← i0 - 1 ;

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

    T ← échange(T,i+1,j0) ;

    retourner(T,i+1) ;
```

FIG. 8.6 – L'algorithme `distribuer`

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 $\Theta(j_0 - i_0)$.

8.4.3 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 :

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

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

8.4.4 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 26 Pour tout entier n, notons T_n le tableau trié $(1, 2, \ldots, n)$.

1. Démontrer que pour tout n et tous indices i et j, `distribuer(Tn,i,j,i)` retourne le couple `(Tn,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 27 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 28

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.

8.4.5 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.

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

8.4.6 Le choix d'un pivot médian

Soit E un ensemble à n éléments, nous appelons *médian* le ième 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 30). 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∈ [1, card(E)]
Sortie : le 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 29 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 :
$$f(n) \leq n + f(\frac{n}{5}) + f(\frac{7n}{10})$$
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 avions considérer des blocs de longueur :
   (a) 3.
   (b) 7.
5. Écrire l'algorithme.

Exercice 30 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 $\alpha = \frac{1}{10}$, $\alpha = \frac{1}{10^{10}}$, $\alpha = \frac{1}{10^{10^{10}}}$?