ENSEIRB-MATMECA

Initiation algorithmique
&
Structures arborescentes

Denis Lapoire

septembre 2024

Initiation algorithmique

Structures arborescentes

Chapitre 1
Problèmes et algorithmes

Nous présenterons ici les 3 qualités attendues d'un algorithme :

- la correction qui comprend l'exécutabilité et la correction syntaxique.
- la lisibilité
- les faibles complexités en temps et en espace

1) LE JOB

Dans tout job, il est important de se poser :

- d'abord la question **pour qui,** le "client"**,** est fait le job et **quoi** l'attendu du "client".
- avant de se poser la question de **comment** faire le job.

Il en est de même pour les algorithmes qui ont pour "clients**"** des *problèmes algorithmiques*, appelés de façon concise *problèmes*.

1.1) les problèmes algorithmiques

Définition : Un problème algorithmique est une relation binaire associant à une *entrée* c.a.d une séquence d'objets de types fixés une *sortie*, elle-même séquence d'objets de types fixés. Le couple formé des 2 séquences des types est appelé le *type* du problème (ou sa *signature*).

hypothèse de simplification 1 :
dans ce cours, les séquences d'entrée et de sortie évoquées ci-avant sont de longueur fixées.

hypothèse de simplification 2 :
en théorie de la complexité, certains problèmes associent à certaines entrées X la sortie « ne termine pas » ce qui signifie que sur les entrées X un algorithme correct peut voir doit ne pas terminer (boucle infinie). Ce cours ne l'admettra pas : un algorithme correct terrmine !

Exemple : le problème

```
PUISSANCE_ENTIER
      ENTREE : deux entiers a et b avec (a,b)≠(0,0)
      SORTIE : l'entier a^b
```

a pour type (ou signature) `(entier , entier)→entier`.

Ce problème contient (notamment) les couples ((2,3),8) et ((3,2),9).

Certains problèmes peuvent associer à une même entrée, une ou plusieurs sorties :

Exemple : le problème

```
DIVISEUR_PROPRE
      ENTREE : un entier n
      SORTIE : un couple (b,p) de type booléen*entier tel que :
                 b est le booléen (n n'est pas premier)
                 si b est vrai, p est diviseur propre de a
```

a pour signature `entier → (booléen,entier)`.

Ce problème contient les couples (8,(1,4)), (8,(1,2)), (3,(0,17)).

1.2) Les algorithmes

Définition :
Un algorithme est un texte (très structuré!) qui pour toute entrée détermine aucune, une ou plusieurs *exécutions*, chaque exécution terminant associant à l'entrée une sortie.

Cette définition est très générale. Nous fournirons dans les chapitres suivants une syntaxe très précise du langage algorithmique que nous manipulerons ensemble, ainsi qu'une sémantique.

1.3) Première qualité d'un algorithme : la correction (relativement à un problème)

Cette première propriété d'un algorithme n'est pas une propriété intrinsèque : elle est relative à un problème.

Définition :
Un algorithme A *résoud* un problème P, si pour toute entrée X de P, toute exécution de A sur X fournit une sortie Y *attendue* par P c'est à dire telle que : (X,Y) $\in$ P.

Dans ce cas, nous dirons au choix :

- A est dit une *solution* (algorithmique) de P
- A *résoud* P ou P est *résolu* par A
- A est *correct* vis à vis de P.

Exemple :
`DIVISEUR_PROPRE` admet pour solution l'algorithme suivant :

```
fonction diviseur_propre(n:entier) : booléen,entier
début
      pour i de 2 à n-1 faire
            si estDiviseur(i,n) alors
                  retourner (vrai(),i)
      ;
      retourner (faux(), 876)
fin
```

1.4) l'exécutabilité et la correction syntaxique nécessaires à la correction

Conséquence de la définition précédente, si un algorithme est correct, il doit s'exécuter sur toute entrée attendue. De même, si un algorithme ne respecte pas la syntaxe, il ne peut pas s'exécuter donc est (toujours) incorrect.

Ces 2 conditions, exécutabilité et correction syntaxique, sont nécessaires à la correction mais malheureusement et de très loin ne sont pas des conditions suffisantes à sa correction.

Exemple : l'instruction `A ← 2 / 0` vérifie la syntaxe `VARIABLE ← entier / entier` mais ne peut pas s'exécuter car la division par `0` n'est pas définie.

2) Algorithmes et Programmes

Les algorithmes ont plus de 4000 ans

Ces procédés sont anciens et utilisaient les premiers calculateurs disponibles : les êtres humains . Nous pouvons dater à l'invention des chiffres (écriture cunéiforme Summer 2ème millénaire avant J.-C.) la conception du premier algorithme : l'incrémentation d'un entier qui permettait de calculer le nombre de moutons.

Les programmes ont moins de 100 ans

Si l'on définit le programme comme un texte pouvant être exécuté sur un ordinateur (*computer*), leur création est récente et date du premier ordinateur : la machine de Turing.

Opposition algorithmes/programmes

Si l'innovation numérique rend rapidement obsolète toute machine et donc tout langage de programmation et donc tout programme, il en est autrement des algorithmes dont l'objet premier est l'échange dans l'espace et dans le temps entre être humains concepteurs aujourd'hui et demain de programmes.

2.1) Deuxième qualité : la lisibilité

Contrairement aux programmes dont le 1er usage est leur exécution par des machines, l'usage premier d'un algorithme est d'être manipulé (écrit, lu et exécuté) par un être humain.

Pour cela il doit être *lisible*. Voici quelques techniques pour assurer la lisibilité :

- la concision : privilégier une écriture en au plus 15 lignes.
- le nommage adapté des variables et des fonctions.
- l'indentation qui met en valeur la structure du texte
- le découpage d'un algorithme à l'aide de sous algorithmes.
- l'absence de commentaires dans le texte de l'algorithme
- (parfois) l'utilisation d'un style récursif

2.2) troisième qualité : une faible complexité en temps et/ou en espace

Un algorithme utilise deux ressources principales : la première est le temps, la seconde est l'espace mémoire de la machine sur lequel s'exécute l'algorithme. Un algorithme sera considéré comme meilleur si il est est de plus faible complexité temps ou ou de plus faible complexité en espace.

3) Quelle méthode pour résoudre un problème algorithmique ?

A la question : existe t-il une méthode permettant de résoudre efficacement un problème algorithmique ; c'est-à-dire d'associer à un problème algorithmique un algorithme ayant les 3 bonnes propriétés ?
La réponse est : NON

En effet, ce problème nécessiterait de savoir résoudre le problème algorithmique:

```
ENTREE : un problème P
SORTIE : un algorithme A solution de P
```

qui contient lui-même le problème :

```
TERMINAISON
```

```
ENTREE : un algorithme A
SORTIE : OUI si A termine sur toute entrée
```

Or Kurt Gödel a prouvé dès 1931 que ce problème et tant d'autres sont *indécidable* c'est à dire qu'ils n'admettent aucune solution algorithmique (références ici et ici).

3.1) trouver une solution optimale en temps à un problème est parfois très compliqué

Depuis presque 100 ans, l'humanité cherche sans succès une solution algorithmique au problème 3SAT qui serait de complexité en temps polynomial et qui améliorerait celles connues à ce jour toutes exponentielles en temps.

3.2) trouver une solution optimale en temps à un problème nécessite toujours de la méthode

3.2.1) première règle : des exemples

Il est absolument nécessaire de manipuler à toute occasion des exemples. Qu'il s'agisse de :

- la compréhension d'un problème fourni
- l'écriture d'un nouveau problème
- l'écriture d'un algorithme
- l'étude de la correction (ou de la complexité) d'un algorithme
- la présentation d'un problème, d'un algorithme, d'une de ses qualités à un tiers : qu'il s'agisse de votre collègue ou de l'enseignant

3.2.2) deuxième règle : la découpe de haut en bas

Comme l'annonçait Descartes dans le discours de la Méthode, les algorithmes doivent être découpés en autant de sous-algorithmes qu'il y a d'idées. En clair, chaque algorithme n'excède jamais 15 lignes. Et utilise des fonctions auxiliaires résolvant chacune un problème algorithmique.

Cette découpe se réalise de haut en bas : en partant du problème initial à résoudre et en identifiant au fur et à mesure de nouveaux problèmes eux mêmes résolus par de nouveaux algorithmes.

4) Conclusion

Avec méthode, nous pouvons parfois exhiber de bonnes solutions algorithmiques à un problème donné. Sans toutefois pouvoir en distinguer une meilleure. Par exemple, un problème P peut admettre (assez souvent) :

- un algorithme A1 rapide mais utilisant beaucoup d'espace.
- un algorithme A2 plus lent mais utilisant moins d'espace.

De façon générale, ces deux algorithmes sont incomparables (aucun n'est meilleur). Mais peuvent être comparés si l'on connaît le contexte précis de leur utilisation :

- un robot sur Mars avec peu de mémoire mais disposant de temps long pour réaliser chaque tâche.
- ou un DATACENTER très puissant mais devant répondre dans le 10ème de seconde.

Chapitre 2
La syntaxe du langage algorithmique

La syntaxe du langage de description algorithmique (LDA) est simple : 3 pages suffisent ! Sa sémantique est plus longue à définir et est contenue dans le chapitre suivant. Cette syntaxe est librement inspiré du langage C, notamment en ce qui concerne les parenthésages.
Voici l'algorithme `test` remplace par `0` un éventuel entier égal à 100 d'une matrice d'entier :

```
fonction test(M : matrice entier)
debut
      (REP, ind_ligne, ind_colonne) ← appartientMatrice(100, M);
      si REP alors M[ind_ligne][ind_colonne] ← 0
fin
```

qui utilise la fonction auxiliaire `appartientMatrice` dont le type est :

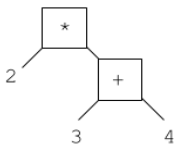
`(entier, matrice entier) → (booléen, entier, entier).`

1) définitions préliminaires

Rappel : désignons par ENTIER un mot composé uniquement de chiffres `0, 1, .. , 9`, désignons par OP l'un des opérateurs `+,*,-,/, ^` . La ligne suivante suffit à produire donc définir toute expression arithmétiques entière de format désigné par le symbole E :

E → (E OP E) OU E OP E OU ENTIER

Ces règles permettent aussi d'évaluer à 14 l'expression non ambigüe `2*(3+4)`. Cette évaluation est réalisée en « reconstruisant » l'arbre syntaxique ci-contre :



Un langage algorithmique se définit à l'aide de règles syntaxiques semblables, qui garantissent de construire l'arbre syntaxique associé à tout algorithme. Pour définir cette syntaxe, nous utilisons les symboles symboles → , OU , séquence,de<> et séquence;de<> ainsi que deux méta-règles :

- la règle A → B C OU D E signifie qu'un texte de format A est par définition un texte de format B suivi d'un texte de format C ou un texte de format D suivi d'un texte de format E.
- séquence,de<X> désigne soit l'expression vide, soit une séquence non vide d'expressions de format X séparées par des virgules "," (formellement X ou X,X ou X,X,X etc. Définition similaire en ce qui concerne séquence;de<X> en utilisant le séparateur ";" au lieu de ",".

2) Syntaxe

Un algorithme est un texte de mots séparé par des caractères blancs ou des retours à la ligne où les mots sont de trois types possibles :

- des *caractères spéciaux* : `, ; ← ( ) : [ ] .`
- des *mots clefs* : `fonction`, `début`, `fin`, `tantque`, `si`, `alors`, `sinon`, `retourner`, `entier`, `booléen`, `réel`, `tableau`, `matrice`.
- d'autres mots de format NOM qui sont des mots composés de caractères latins ou de chiffres autres que les mots clefs.

Le format TYPE désigne l'un des trois mots `entier, booléen, réel` ou l'un des 2 mots `tableau,` matrice suivi de l'un des mots `entier, booléen, réel` .

Voici en quelques lignes la syntaxe qui vous permet d'écrire tout algorithme :

r1 ALGORITHME →
`fonction`
NOM
(
séquence,de<NOM : TYPE>
)
séquence,de<TYPE>
`debut`
séquence;de<INSTRUCTION >
`fin`

r2 INSTRUCTION →
r2.1 `vide`
r2.2 OU EXPRESSION
r2.3 OU AFFECTATION
r2.4 OU `retourner` séquence,de<EXPRESSION>
r2.5 OU `debut` séquence;de<INSTRUCTION> `fin`
r2.6 OU `si` EXPR_BOOL `alors` INSTRUCTION `sinon` INSTRUCTION
r2.7 OU `tantque` EXPR_BOOL `faire` INSTRUCTION

r4 AFFECTATION -→
EXPRESSION_GAUCHE ← EXPRESSION

r3 EXPRESSION →
r3.1 NOM
r3.2 OU NOM [EXPRESSION]
r3.3 OU NOM [EXPRESSION][EXPRESSION]
r3.4 OU NOM (séquence,de<EXPRESSION >)
r3.5 OU (EXPRESSION OP EXPRESSION)
r3.6 OU EXPRESSION OP EXPRESSION
r3.7 OU ENTIER

r5 EXPRESSION_GAUCHE -→
r5.1 NOM
r5.2 OU NOM [EXPRESSION]
r5.3 OU NOM [EXPRESSION] [EXPRESSION]
r5.4 OU (séquence,de< NOM
OU NOM [EXPRESSION]
OU NOM [EXPRESSION][EXPRESSION]>)

3) augmentation de la syntaxe

Afin de gagner en lisibilité, on enrichit la syntaxe. Ainsi les boucles `pour, répéter` et `faire` sont moins expressives que `tantque` mais plus lisibles :

3.1)la structure de contrôle `si alors`

`si` EXPRESSION `alors`
INSTRUCTION
	a pour définition :	`si` EXPRESSION `alors`
		INSTRUCTION
		`sinon`
		`vide`

3.2) boucle `pour`

`pour` NOM `de` A `à` B `faire`
INSTRUCTION
	a pour définition :	NOM ← A ;
		`tantque` NOM ≤ B `faire`
		`debut`
		INSTRUCTION ;
		NOM ← NOM + 1
		`fin`

3.3) boucle `répéter`

`répéter` A `fois` INSTRUCTION	a pour définition :	`pour` i `de` 1 `à` A `faire` INSTRUCTION ;

3.4) boucle `faire`

`faire` INSTRUCTION `jusqu'à` EXPR_BOOL	a pour définition :	INSTRUCTION ; `tantque non (`EXPR_BOOL`) faire` INSTRUCTION

4) syntaxe ambigüe ?

Autoriser certaines écritures crée un risque d'erreur d'incompréhension ou d'ambigüité.

4.1) expressions ambiguës ?

Par exemple , la règle r3.6 EXPRESSION → EXPRESSION `opérateur` EXPRESSION permet d'économiser un grand nombre de parenthèses ; la suppression des ambiguïtés est réalisée à l'aide de règles de priorités. La plus célèbre est connue de tous de tous les enfants de 10 ans et affiche la priorité du produit * devant l'opérateur + . Ainsi l'expression `2*3+4` équivaut à `((2*3)+4)` et non pas à `(2*(3+4))` et est ainsi égale à 10 et non pas à 14.

Attention à ne préjuger de votre maîtrise de la cinquantaine de règles (61 règles en C, 38 règles Python) et veillez à sécuriser votre code en ajoutant suffisamment de parenthèses.
Exemple :
Question : que vaut l'expression suivante : 1<0<=1

Réponse : la priorité de < devant <= permet d'interpréter 1<0<=1 comme (1<0)<=1 de valeur 1. Et non d'interpréter 1<0<=1 comme 1<(0<=1) de valeur 0.

4.2) instructions ambiguës ?

Le langage défini plus haut possède une ambiguïté lié à l'utilisation d'un unique mot clef si pour parenthéser les 2 structures de contrôles **si alors sinon** et si **alors** .
En effet, quelle est l'exécution de l'instruction :

`si` A `alors si` B `alors` C `sinon` D

2 interprétations sont possibles ;

- la première est : `si` A `alors debut si` B `alors` C `sinon` D `fin`
- la seconde est : `si` A `alors debut si` B `alors` C `fin sinon` D

et sont très différentes :

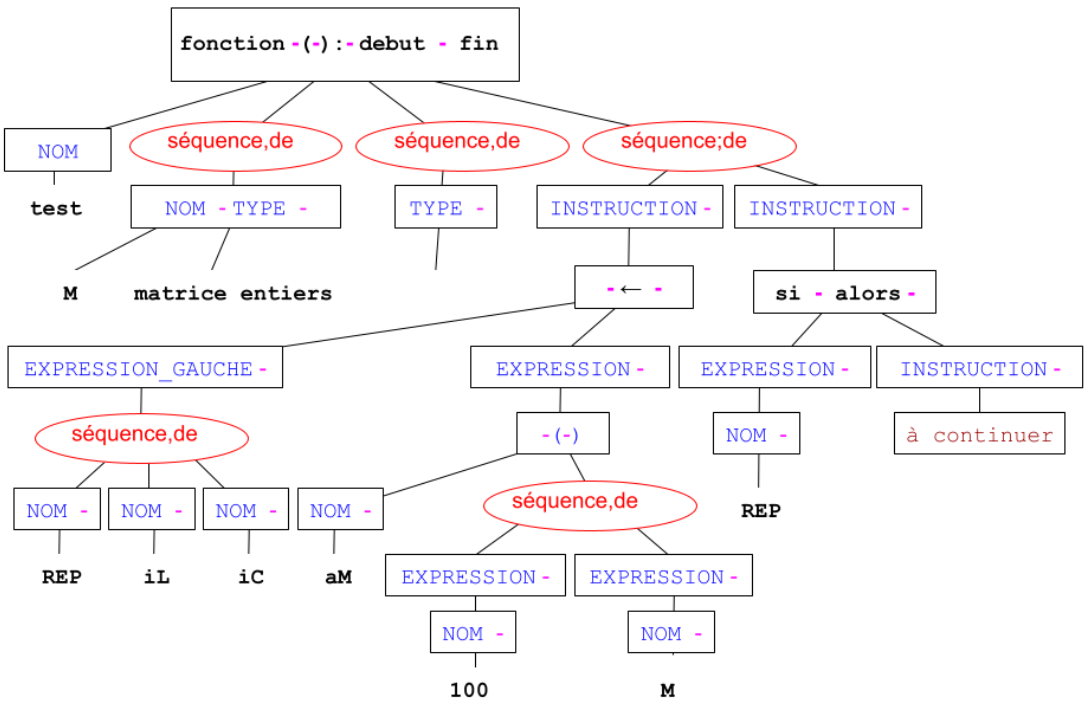
- `si faux() alors debut si faux() alors` C `sinon a←0 fin` équivaut à `VIDE`
- `si faux() alors debut si faux() alors` C `fin sinon a←0` équivaut à `a←0`

Il existe différentes façons ou d'éviter l'ambiguïté ou de la lever (voir https://fr.wikipedia.org/wiki/Dangling_else). Cependant, nous faisons le choix contraire d'une très grande frugalité en ajoutant aucune de ces règles ; nous recommandons simplement d'ajouter les parenthésages `debut fin` en nombre suffisant.

5) conclusion

Comme annoncé en introduction, les règles permettant de produire tout algorithme permettent de calculer à partir de ce texte son arbre syntaxique. Comme l'illustre l'exemple suivant :

```
fonction test(M : matrice entier)
debut
  (REP, iL, iC) ← aM(100, M);
  si REP alors M[iL][iC] ← 0
fin
```



Chapitre 3
La sémantique du langage algorithmique

Le cours initiation algorithmique utilise un nombre limité de types qui seront définis dans les annexes à ce chapitre :

1. des types primitifs de taille constante
   `entier`, `réel`, `booléen`, `int`
2. des types non primitifs de taille non constante
   `tableau`, `matrice` et `entier long`

1) les types

Le langage que nous utilisons est fortement typé. Chaque variable possède dès sa création un type dont elle ne change pas. De même, chaque fonction possède un type, appelé signature, dont elle ne change pas.

Définition : un *type abstrait* est un ensemble de fonctions et d'opérations. Chaque opération possède une *signature* c'est-à-dire un couple de séquences de type. Un usage courant est d'écrire ce couple en utilisant le symbole "→".

Exemple : l'opération binaire ≤ de signature `(entier,entier)` → `booléen` associe, par exemple, à l'expression `3 ≤ 2` la valeur booléenne `faux()`.

Remarque : Un même symbole désignant des opérations différentes. Conséquence de la définition ci-dessus, l'opération addition, de signature `(entier,entier)→entier`, est différente de l'addition sur les réels, de signature `(réel,réel)→réel` . En conséquence de la définition plus haut, nous pourrions utiliser deux symboles distincts par exemple $+_e$ et $+_r$ pour désigner ces 2 opérations. Pour des raisons de simplicité nous utilisons le même symbole +.

2) la mémoire, la Pile et le Tas

Définition : La *mémoire* est un tableau d'octets (8 bits) indicés par des entiers nul et positifs. Ce tableau est supposé infini. L'indice de chaque octet est appelé son *adresse mémoire*.

Définition :
Lors de l'exécution d'un algorithme, la mémoire se répartit en 4 zones mémoires :

1. la zone *CODE*, qui contient l'algorithme lui-même. Cette zone est de taille fixée.
2. la zone *GLOBALE*, qui contient les variables statiques et globales. Cette zone est de taille fixée. Nous utiliserons jamais (ou exceptionnellement) des variables statiques ou globales. En bref, cette zone ne sera jamais utilisée (dans ce cours).
3. la zone *PILE d'APPEL*, qui contient une séquence d'espaces mémoires contiguës ; à chaque appel de fonction, **MACHINE** associe un unique espace mémoire, appelé

bloc, qui est détruit (ou libéré) à la fin de l'appel ; **MACHINE** gère ces espaces à l'aide d'une pile d'où le nom.

4. zone *LE_TAS*, qui contient les tableaux et les matrices (ainsi que les structures qui seront présentées en Structures arborescentes).

3) les variables

Définition : une *variable* est un quadruplet contenant :

1. un *type*
2. un *identifiant* (une chaîne de caractère)
3. une *adresse mémoire*
4. une *valeur*

Définition :
Un type T1 est *primitif* si tous les objets de type T1 peuvent être représentés à l'aide d'un même nombre d'octets ; la *valeur* d'une variable d'un tel type T1 est cette représentation.

Pour un type T2 *non primitif*, la *valeur* de toute variable de type T2 est égale à l'**adresse mémoire** de l'objet qu'elle désigne.

Exemple :
L'entier 17 de type primitif `int` est représenté par 4 octets : 00000000 00000000 00000000 00010001. La valeur de la variable issue de l'exécution de a←17 est l'entier 17 représenté par les 4 octets.

Exemple :
Ainsi, l'expression `allouerTableau(10,5)` est de type `tableau d'entiers` ; son évaluation par **MACHINE** consiste à :

1. allouer dans la mémoire `LE_TAS` 10 octets successifs mémorisant chacun l'entier 5.
2. retourner l'adresse de ce tableau.

Remarque :
En `langage C`, l'adresse du tableau est l'adresse de son premier élément. Ce point est important car le langage C permet de réaliser des opérations sur les adresses mémoires. En `LDA`, ces opérations ne sont pas autorisées ; exception faite éventuellement de la comparaison =, de signature `(adresse,adresse)→booléen.`

Conséquence :
Pour un type donné, primitif ou non, toute valeur d'une variable de ce type utilise un nombre d'octets constant.

4) l'affectation

Définition :

Soient `appel` un appel d'une fonction et `BLOC` l'élément de `PILE d'APPEL` associée à `appel` ; l'exécution par **MACHINE** de l'instruction `EXPR1 ← EXPR2` lors de l'appel `appel` consiste à :

1) **MACHINE** évalue l'expression `EXPR2` et retourne un type `type2` et une valeur `val2`.

2) si `EXPR1` est de sorte `VARIABLE` et si cette variable n'existe pas dans `BLOC`, alors **MACHINE** crée une nouvelle variable dont les caractéristiques sont :
   - son adresse mémoire est placée dans `BLOC`
   - son identifiant est `EXPR1`
   - son type est `type2`
   - sa valeur est `val2`

3) sinon si `EXPR1` est de sorte `VARIABLE` et si cette variable existe dans `BLOC`, alors **MACHINE** place à l'adresse mémoire de `EXPR1` la valeur `val2`.
4) sinon,
   `EXPR1` est de forme `NOM_TAB[EXPR3]`,
   alors **MACHINE** évalue l'expression `EXPR3` (notons sa valeur `indice`), et recopie la valeur `val2` à l'indice `indice` du tableau `NOM_TAB` placé en mémoire `LE_TAS`.

Conséquence 1 : pour créer une variable de nom NOM, il suffit d'utiliser l'affectation NOM ← EXPRESSION qui évalue le type et la valeur de l'expression EXPRESSION et qui l'affecte à la nouvelle variable NOM.

Exemple : ainsi, si T[3] a pour valeur 4, l'exécution de : `A ← T[3] + 2` a pour conséquence la création d'une variable de nom A, de type `entier` et de valeur 6.

Conséquence 2 :

Naturellement, l'affectation ← permet de modifier la valeur d'une variable existante. Voir exemple suivant :

Exemple :

L'algorithme suivant :

```
100 fonction test1()
101 debut
102   a ← 6 ;
103   b ← 4 ;
104   tantque a ≠ b faire
105         si a > b alors
106               a ← a - b
107         sinon
108               b ← b - a
109 fin
```

a pour exécution celle décrite ci-après. Le schéma décrit les instants de création des variables a et b ainsi que l'évolution de leurs valeurs.

état de la variable a	état de la variable b	ligne traitée	commentaire
-	-	100	
-	-	101	
-	-	102	création de la variable a du à : a ← 6
6	-	103	création de la variable b du à : b ← 4
6	4	104	évaluation de : a≠b et branchement en 105
6	4	105	évaluation de : a>b et branchement en 106
6	4	106	modification de valeur du à : a ← a-b
2	4	104	évaluation de : a≠b et branchement en 105
2	4	105	évaluation de : a>b et branchement en 108
2	4	108	modification de valeur du à : b ← b-a
2	2	104	évaluation de : a≠b et branchement en 109
2	2	109	

Conséquence 3 : L'algorithme suivant retourne à l'évidence le booléen `faux` :

```
0000 fonction main()
0001 debut
0002       a ← t2()
0003 fin

0004 fonction t2() booléen
0005 début
0006       a ← 4 ;
0007       b ← a ;
0008       b ← 5 ;
0009       retourner a = b
0010 fin
```

Voici l'état des différents espaces mémoires :

après `0000`

CODE		fonction main() … fin
PILE	bloc appel main	

au début exécution `a ← t2()`

CODE		fonction main() … fin
PILE	bloc appel main	a booléen:?
	bloc appel t2	

après appel de `t2()`

CODE		fonction main() … fin
PILE	bloc appel main	a booléen:?
	bloc appel t2	S booléen: ?

après exécution de `a ← 4`

CODE		fonction main() … Fin
PILE	bloc appel main	a booléen: ?
	bloc appel t2	S booléen: ? a entier : 4

après exécution de `b ← a`

CODE		fonction main() … Fin
PILE	bloc appel main	a booléen: ?
	bloc appel t2	S booléen: ? a entier : 4 b entier : 4

après exécution de `b ← 5`

CODE		fonction main() … fin
PILE	bloc appel main	a booléen:?
	bloc appel t2	S booléen: ? a entier : 4 b entier : 5

fin exécution `0009`

CODE		fonction main() … fin
PILE	bloc appel main	a booléen: ?
	bloc appel t2	S booléen: 0 a entier : 4 b entier : 5

après exécution de `0002`

CODE		fonction main() … fin
PILE	bloc appel main	a booléen: 0

L'algorithme suivant retourne le booléen `vrai()` :

```
000 fonction main()
001 début
002       a ← t3()
003 fin
004   fonction t3() booléen
005   début
006         a ← allouerTableau(3,10) ;
007         b ← a ;
008         b[1] ← 1000 ;
009         retourner a = b ET a[1]=1000
010   fin
```

La raison est simple : après exécution de b ← a les deux variables différentes a et b sont du type non primitif `tableau d'entiers` et partagent une même valeur qui est l'adresse d'un même tableau. Quand on modifie le contenu du tableau désigné par a , on modifie le contenu de celui désigné par b : ils sont identiques au sens physique du terme. Voici différents états de la mémoire lors de l'exécution de `t3()` :

après exécution de `allouerT.(3,10)`

CODE		000 010	fonction … fin
PILE	bloc main	011	a ? : ?
	bloc appel t3	012	S booléen : ?
		013	a tab._ent. : **101**
TAS		101 102 103	10 10 10

après exécution de `b ← a`

CODE		000 010	fonction … fin
PILE	bloc main	011	a ? : ?
	bloc appel t3	012	S booléen : ?
		013	a tab._ent. : **101**
		015	b tab._ent. : **101**
TAS		101 102 103	10 10 10

après exécution de `b[1] ← 1000`

CODE		000 010	Fonction … fin
PILE	bloc main	011	a ? : ?
	bloc appel t3	012	S booléen : ?
		013	a tab._ent. : 101
		015	b tab._ent. : 101
TAS		101 102 103	1000 10 10

après exécution de `a ← t3()`

CODE		000 010	Fonction … fin
PILE	bloc main	011	a booléen : 1
TAS		101 102 103	1000 10 10

Conséquence 4 :

L'algorithme suivant

```
1021  fonction test4(n:entier) booléen
1022  début
1023        faire n fois test5()
1024  fin

1026  fonction test5()
1027  début
1028        a ← 1 ;
1029  fin
```

utilise une mémoire de taille constante. En effet la mémoire allouée lors de chaque appel de `test5` est située dans un bloc de la pile d'appel ; ce bloc est libéré à chaque retour d'appel.

L'algorithme suivant

```
1021  fonction test6(n:entier) booléen
1022  début
1023        faire n fois test7()
1024  fin

1026  fonction test7()
1027  début
1028        a ← allouerTableau(1,1) ;
1029  fin
```

utilise une mémoire de taille non constante. En effet la mémoire allouée lors de chaque appel de `test7` est située d'une part dans le bloc de la pile d'appel qui est libéré à chaque retour d'appel mais aussi d'autre part dans LE_TAS qui accumule à la fin de l'exécution de test5(1000) un **millier** de tableaux. Chacun de ces tableaux, sauf éventuellement un, n'est désigné par aucune variable donc est inutilisable. Pour éviter cette situation de grand gâchis écologique, il faut utiliser la fonction `libérer` qui est la seule façon libérer l'espace mémoire d'un tableau.

5) appel de fonctions

Définition

Soit une fonction de prototype :

$\texttt{fonction NOM\_fonction(VAR}_1\texttt{ : type}_1\texttt{,..,VAR}_k\texttt{ : type}_k\texttt{)type}_{k+1},..,\texttt{ type}_m$

Pour évaluer une expression de la forme $\texttt{NOM\_fonction(EXPR}_1\texttt{,..,EXPR}_k\texttt{)}$ **MACHINE** réalise les tâches suivantes :

1. **MACHINE** évalue les k expressions $\texttt{EXPR}_1\texttt{,..,EXPR}_k$; notons $\texttt{val}_1\texttt{,..,val}_k$ les valeurs issues de ces k évaluations ; notons $\texttt{type}_1\texttt{,..,type}_k$ les k types associés ; notons $\texttt{VAR}_1\texttt{,..,VAR}_k$ les noms dans le CODE des k variables en entrée de `NOM_fonction`.

2. **MACHINE** *réalise un appel de fonction* sur la fonction `NOM_fonction` ce qui consiste à :
    a. créer un bloc `BLOC_APPEL` dans la pile d'appel contenant :
        i. m-k espaces mémoires munis des m-k types $type_{k+1}, .., type_m$
        ii. k espaces mémoires associés aux k variables $VAR_1, .., VAR_k$ et aux k types $type_1, .., type_k$
        iii. un espace mémoire pour chacune des variables apparaissant dans le code de `NOM_fonction`
    b. exécute le code de `NOM_fonction` qui consiste notamment lors de l'exécution de `retourner`$(e_{k+1}, .., e_m)$ de copier dans les m-k espaces prévus les m-k valeurs issues de l'évaluations des m-k expressions $e_{k+1}, .., e_m$.
3. **MACHINE** *réalise un retour d'appel de fonction* en copiant les m-k valeurs placées dans `BLOC_APPEL` dans les espaces mémoires devant recevoir ces valeurs ; ces espaces mémoires se trouvent soit dans le bloc de la pile d'appel précédant `BLOC_APPEL` soit dans `LE_TAS`. `BLOC_APPEL` est ensuite effacé.

Chapitre 4
Récursivité et arbre des appels

L'écriture récursive est pour beaucoup le style d'écriture le plus simple voire le plus naturel. Voici en 2 lignes le principe récursif de la recherche dichotomique :

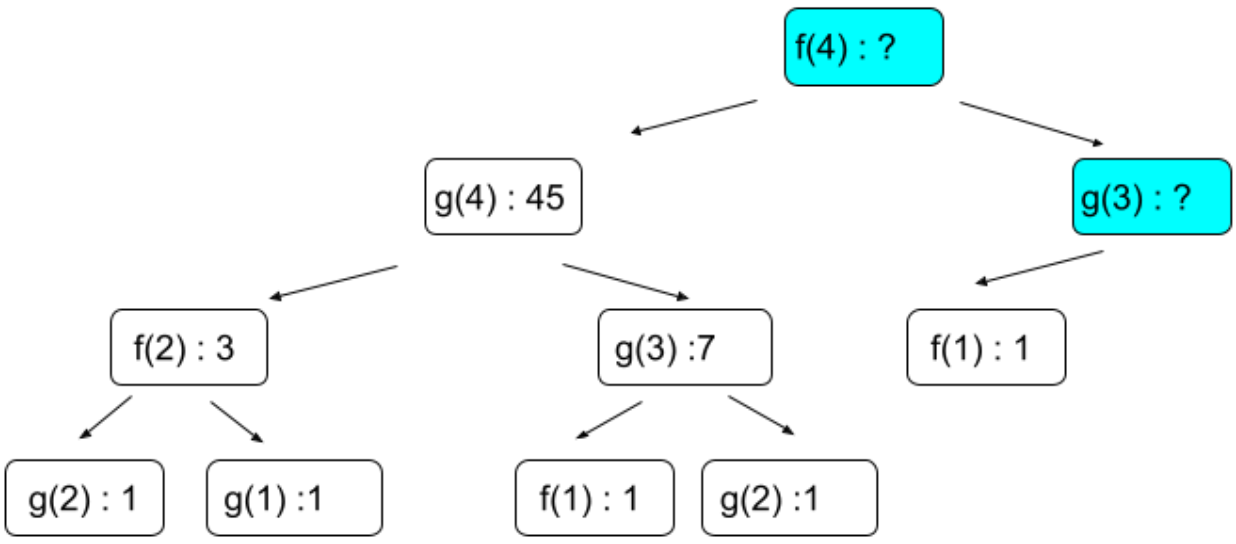
Chercher un entier x dans un tableau T trié consiste à le chercher au milieu y de T,
et sinon puis si x est inférieur à y à le chercher à la gauche de y ou sinon à sa droite.

La façon dont la machine exécute un algorithme récursif est réalisée à l'aide l'arbre des appels qui fournit l'historique des états mémoires de cette machine, appelés piles d'appels. Voici un exemple :

Exemple de fonction récursive et d'arbre des appels :

Voici un arbre des appels associé à la fonction f définie ci-dessous et l'entier 4 (voir détails plus loin) :

```
01    fonction f(a:entier) entier
02    début
03          si a ≤ 1 alors retourner 1 ;
04          A ← 2*g(a) ;
05          B ← g(a-1) ;
06          retourner A + B
07    fin
09    fonction g(a:entier) entier
10    début
11          si a ≤ 2 alors retourner 1 ;
12          A ← f(a-2) ;
13          B ← g(a-1) ;
14          C ← k(a) ;
15          retourner A*B+C
16    fin
18    fonction k(a:entier) entier
19    début
20          si a ≤ 0 alors retourner 1 ;
21          retourner a* k(a-1)
22    fin
```



fin_exemple

1) Arbre des appels

Définition

L'*arbre des appels* associé à l'exécution d'une fonction A sur une entrée X est l'arbre :

- dont les noeuds sont associés à chacun des appels des différentes fonctions appelées lors de l'exécution de A sur l'entrée XINIT.
- pour chacun des appels d'une fonction F sur une entrée X correspond un noeud ayant pour attributs le nom F, l'entrée X et la valeur retournée F(X)
- si l'exécution de F sur une entrée X entraine directement les appels des fonctions F1,..,Fm sur les entrées X1,..,Xm alors cela est représenté par m arcs de F vers les m Fi ; les l m noeuds associés aux m appels des fonctions Fi sont placés de la gauche vers la droite.

L'arbre des appels peut être *limité* à certaines fonctions : en ne conservant que les noeuds associés à ces fonctions.

L'*arbre des appels temporaire* est celui lié à un instant de l'exécution de la fonction et limité aux appels de fonctions exécutés. Il possède deux sortes de nœuds : ceux associés à des appels de fonctions totalement exécutés - qui ont donc été retournés - et les autres.

Exemple d'un arbre des appels temporaire :

Voici l'arbre des appels :

- associé à la fonction f et l'entrée 4
- limités aux fonctions f et g (les appels à la fonction k sont ici masqués)
- temporaire selon l'instant t1 où t1 est l'instant précis lors de l'exécution de g sur 3 (en bleu) après le retour de l'appel de f sur l'entrée 1 (en blanc) et donc avant l'appel de g sur l'entrée 2.

fin_exemple

2) La pile d'appel

Définition

Considérons un instant t lors de l'exécution d'un algorithme.La pile d'appel est l'ensemble (empilé) des espaces alloués en mémoire disposant chacun des informations nécessaires à l'exécution d'un appel de fonction en cours d'exécution.

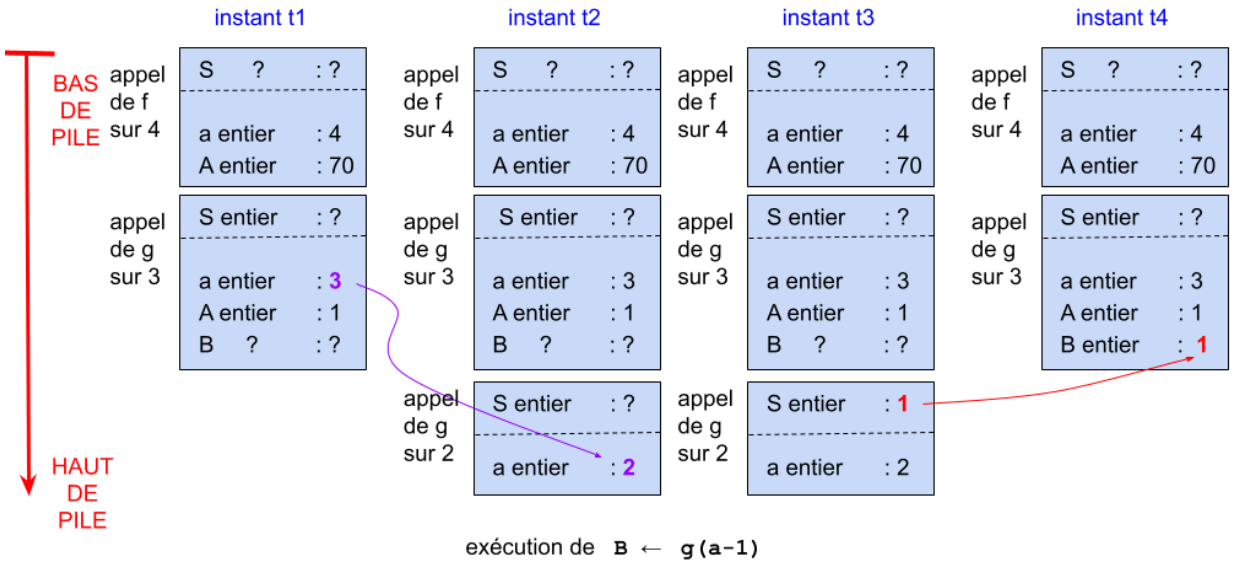
L'exemple ci-dessous fournit une description complète à trois instants successifs de l'état de la pile d'appel. En cohérence avec l'arbre des appels (dessiné du haut vers le bas), la pile sera dessinée du haut vers le bas : **le bas de la pile est donc en haut !**

Exemple de pile d'appel :

Cet exemple présente quatre états de la pile d'appel à quatre instants t1, t2, t3 et t4 permettant l'exécution de `(ligne 13)B ← g(a-1)` au cours de l'appel `g(3)` :

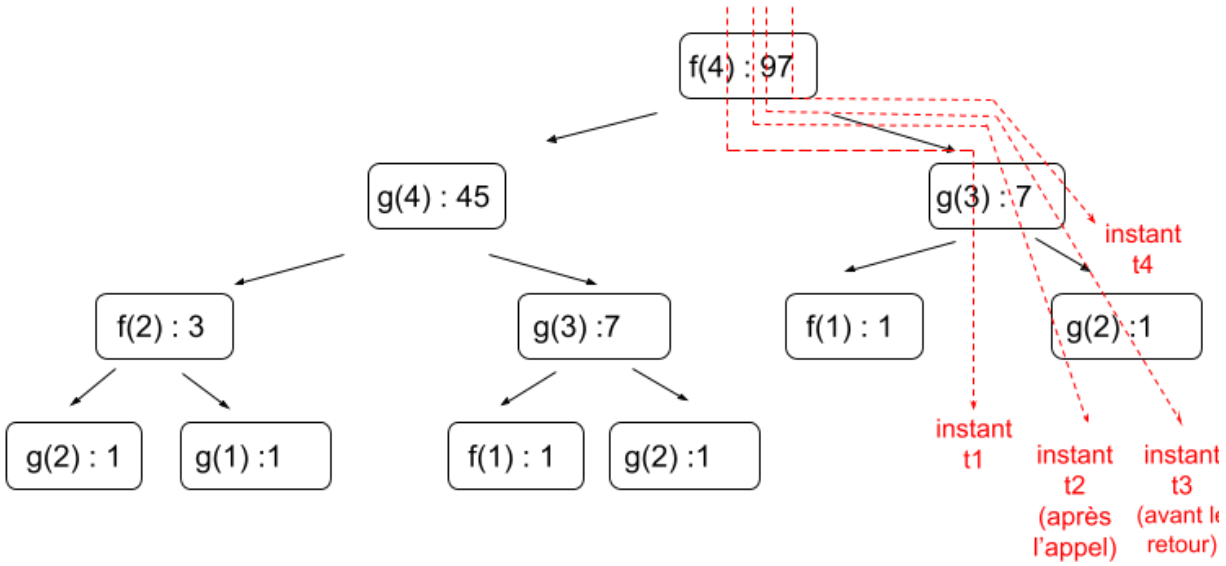
- t1 est l'instant juste après l'exécution de `(ligne 12)A ← f(a-2)` au cours de l'appel `g(3)`
- t2 est l'instant juste après l'exécution de `(ligne 10)debut` au cours de l'appel de `g(2)`
- t3 est l'instant juste avant la fin de l'exécution de `(ligne 11)si a≤2 alors retourner 1` au cours de l'appel de `g(2)`
- t3 est l'instant juste après l'exécution de `(ligne 13)B ← g(a-1)` au cours de l'appel `g(3)`

Note 1 : le choix fait ici est celui de langage compilé comme le langage C. Tous les éléments de pile associés à une fonction ont la même structure et prévoient un espace mémoire pour chacune des variables apparaissant dans le code (c'est la raison pour laquelle les variables sont déclarées). Ainsi, à l'instant t2 les variables A, B et C apparaissent dans l'élément de haut de pile même si elles ne jouent aucun rôle dans l'appel de g sur l'entrée 2.



3) L'arbre des appels décrit l'historique de la pile des appels

Voici l'arbre des appels de l'appel de la fonction f sur l'entrée 4.



Cet arbre fournit en fait l'historique des piles d'appels. Il suffit pour cela de tirer des lignes partant du noeud le plus haut (la racine) et traversant les noeuds de père en fils. Les trois lignes (pointillés rouges ci-dessus) sont associés aux trois évolutions de la pile d'appel t1, t2 et t3 présentées plus haut.

4) Conclusion

L'arbre des appels décrit complètement l'exécution de l'algorithme. C'est en analysant celui-ci que nous pourrons calculer les complexités en temps et en espace de l'algorithme. Rassurons nos lecteurs, il ne sera pas forcément nécessaire de le dessiner seulement d'étudier ses propriétés notamment deux principales :

- la hauteur maximale de l'arbre qui est souvent la complexité en espace.
- le nombre total de noeuds de l'arbre qui est souvent la complexité en temps.

Ces points seront abordés au prochain chapitre.

Chapitre 5
Complexités temps et espace

Ce chapitre introduit les notions de complexité temps et espace des algorithmes ainsi que des problèmes algorithmiques. Leurs définitions s'appuient sur la notion suivante :

La **complexité** (en espace) d'une entrée est le **nombre d'octets** pour la représenter.

Avant de définir les complexités temps d'un algorithme, voici 2 propriétés caractéristiques :

Un algorithme est de **complexité en temps linéaire**
si il s'exécute en un temps égal à celui nécessaire à lire les données en entrée.

Un algorithme est de **complexité en temps exponentielle**
si il s'exécute en un temps exponentiel à celui nécessaire à lire les données en entrée.

Afin de distinguer linéaire et exponentiel, convoquez vos souvenirs d'enfance quand votre simple instinct faisait la différence entre exercice et punition. Voici deux ordres : l'un simple exercice qu'un enfant réalise aussi simplement qu'il lit l'entier de droite à gauche. Le second est une punition, ici inhumaine. L'entier choisi ci-dessous utilise 8 octets c.a.d 20 digits :

un exercice d'enfant :
Calculer le double de l'entier 11 055 175 255 055 185 255 **;**

une punition (ici inhumaine) :
Recopier "je ne mérite pas ca" 11 055 175 255 055 185 255 **fois;**

L'impossibilité du calcul exponentiel concerne aussi la machine. Sous l'hypothèse que la machine fait 1 milliard d'opérations par seconde voici le temps de calcul pour 2 algorithmes l'un linéaire l'autre exponentiel pour 3 entrées de différentes complexités :

3 exemples d'entiers	# chiffres	Complexité #octets	temps linéaire	temps exponentiel
123	3	1	10^{-9} s	2.10^{-9} s
12345678901234567890123456789012345678901234567890 12345678901234567890123456789012345678901234567890	80	32	32.10^{-9} s	4 s
12345678901234567890123456789012345678901234567890 12345678901234567890123456789012345678901234567890 12345678901234567890123456789012345678901234567890 12345678901234567890123456789012345678901234567890	160	64	64.10^{-9} s	5 siècles

Détail de calcul : puisque 2^10 égale 10^3 (1024), 1,25 octets « égale » 3 chiffres.

1) les types

1.1 les types de complexité constante du langage C

La machine de calcul fournit un ensemble d'objets de types primitifs. Chaque type est en fait un ensemble d'opérations de complexités en espace et en temps constantes.

1.1.1 type `booléen`

Les opérations logiques sont unaires : NOT ou binaires : AND, OR, NOR, ⇔, ⇒ , ⇐

Complexité en temps : constante
Complexité en espace : constante ; plus exactement 1 bit

1.1.2 type `int` issu du langage C

Les opérations arithmétiques usuelles : +,*,/,^,%
Complexité en temps : constante
Complexité en espace : constante ; plus exactement 4 octets

Attention : toutes les opérations s'exécutent (parfois) incorrectement. Car 32 bits ne peuvent représenter au plus que 2^{32} entiers.

1.1.3 type `double` issu du langage C

Les opérations arithmétiques usuelles : +,*,/,^,
Complexité en temps : constante
Complexité en espace : constante ; plus exactement 8 octets

Attention : toutes les opérations s'exécutent (presque toujours) incorrectement. Même raison que précédemment.

1.2 les types de complexité constante de notre machine théorique

Prouver la correction d'algorithmes utilisant le type `int` nécessiterait de limiter les entiers manipulées ; ces limitations n'ont aucun intérêt dans ce cours d'initiation algorithmique. Pour cette raison, nous utiliserons un autre type primitif de nom `entier`:

- grand avantage : `entier` a une complexité en espace constant ; toutes les opérations s'exécutent correctement en temps constant.
- petit inconvénient : le type `entier` est irréaliste : il ne peut pas être implémenté dans un ordinateur réel.

Le caractère irréaliste de la complexité constante du type `entier` est acceptable quand cet entier concerne par exemple l'indice d'un tableau. Dans ce cas, la complexité des indices du tableau est négligeable devant la complexité des éléments du tableau. En résumé, pour représenter les indices on utilise le type `entier`.

1.2.1 type `entier`

Les opérations arithmétiques usuelles : +,*,/,^,%

Complexité en temps : constante
Complexité en espace : constante

1.2.2 type `réel`

Les opérations arithmétiques usuelles : +,*,/,^
Complexité en temps : constante
Complexité en espace : constante

1.3 types non primitifs

A l'opposé du cas précédent, si l'entier est la seule entrée d'un problème (par exemple factorisation d'un entier ou tout problème utilisé en cryptologie), il est nécessaire de ne pas considérer le type `entier` mais un type qui considère que tout entier n est de complexité log(n). Pour cela, nous utilisons le type suivant :

1.3.1 type `entier long`

Voici une première définition du type `entier long`.
Pour rassurer le lecteur, nous fournissons les fonctions incrémentation et addition qui sont de complexité linéaires. Mais nous ne fournissons pas ici les fonctions usuelles produit, puissance (ou exponentielle), division.

nom	Type
	`eL : entier long`
	`e : entier`
		complexité temps	complexité espace

ièmeBit	`e,eL→booléen`	constant	constant
entierLong	`e → eL`	-	-
décalage droit	`eL,e→eL`	constant	constant
incrémentation	`eL→eL`	linéaire	linéaire
addition	`eL,eL→eL`	linéaire	linéaire

Attention : linéaire signifie par exemple que additionner deux entiers a et b de complexités l'un $\log_2(a)$ bits et l'autre $\log_2(a)$ bits se réalise en $\log_2(a)+\log_2(a)$ opérations élémentaires.

Attention : les complexités temps et espace fournies ici sont indicatives. Par exemple, garantir des complexités constantes pour le décalage (1029>>2=257) suppose une implémentation astucieuse (tableau de int avec 2 indices).

1.3.2 type `tableau`

Opérations d'écriture (syntaxe : `T[1] ← 4`) et lecture (syntaxe : `a ← T[1]`)
Complexité en temps : constante
Complexité en espace : constante

`longueur(T:tableau) : entier`
Complexité en temps : constante
Complexité en espace : constante

`allouerTab(entier,objet) : tableau de type type(objet)`
exemple `allouerTab(9,2)` crée un tableau de longueur 9 à valeurs entières à 2.
Complexité en temps : longueur(T)
Complexité en espace : longueur(T)

`libérer(T:tableau)`
`libérer(T)` supprime le tableau T et libère son espace mémoire
Complexité en temps : constante
Complexité en espace : constante

Attention (rappel) : L'affectation `A←B` où `B` désigne un tableau est autorisé et a pour conséquence que la variable `A` désigne le même tableau que `B`. En conséquence, la modification de `A` entraîne celle de `B`, et réciproquement.

Le même mécanisme est utilisé lors des appels de fonctions qui réalisent sur les paramètres de type tableau des passages par référence. Qui sont de complexité temps et espace constante.

2) Complexités temps et espace

2.1 complexité d'une donnée

Définition :
La *complexité* d'une donnée est le nombre d'octets de sa représentation mémoire.

Exemple : la complexité d'un objet de type `entier, réel, booléen, real` ou `int` est constante. La complexité d'un tableau est la somme des complexités de ses éléments. La complexité d'un `entier long` n est $\log_2(n)$.

2.2 complexité en temps d'un algorithme

Définition :
La *complexité en temps* d'un algorithme **A** est la fonction notée $\#_{\mathbf{A}} : \mathbb{N}^* \rightarrow \mathbb{N}^*$ qui associe à tout entier n le nombre maximal d'instructions élémentaires exécutées par **A** sur des entrées de complexité au plus n. Cette complexité est couramment désignée comme celle dans le *pire des cas*.

D'autres complexités en temps existent. En voici 2 autres :
Définition :
La *complexité en temps dans le meilleur des cas* a pour définition celle obtenue à partir de la définition précédente en y remplaçant le terme "maximal" par "minimal".

La *complexité en temps en moyenne* a pour définition celle obtenue à partir de la définition précédente en y remplaçant le terme “maximal” par “moyen”.

Exemple : complexités en temps de `incrémenter : entier long → entier long`

Voir fichier

2.3 complexité en espace d’un algorithme

Définition :

La *complexité en espace* d’un algorithme **A** est la fonction $\mathbb{N}^* \to \mathbb{N}^*$ qui associe à tout entier n le nombre maximal d’octets alloués aux différents instants de toutes les exécutions de **A** sur toutes les entrées de complexité au plus n.

Remarque :

D’autres complexités en espace existent : celle dans le meilleur des cas ou celle moyenne.

3) Approximations et majorations

3.1 1ère simplification ; notation Θ

Systématiquement, nous fournirons une approximation des fonctions complexité.

Définition f=Θ(g)

Soient deux fonctions f et g de type $\mathbb{N}^* \to \mathbb{N}^*$. f est *égale à g à une constante multiplicative près, noté* $f=\Theta(g)$, *si* il existe deux réels A>0 et B>0 vérifiant : $A.g \leq f \leq B.g$

c’est à dire vérifiant $\forall n \in \mathbb{N}^*$, $A.g(n) \leq f(n)$ et $f(n) \leq B.g(n)$.

L’ensemble des fonctions égales à g à une constante multiplicative près est notée $\Theta(g)$.
fin_definition

Cette propriété est réflexive, symétrique et transitive comme l’indique la propriété suivante. Il s’agit donc d’une r*elation d’équivalence*. L’ensemble $\Theta(g)$ est ainsi une classe d’équivalence. Les preuves seont établies en TD.

Propriété[Equivalence] : Toutes fonctions f, g et h : $\mathbb{N}^* \to \mathbb{N}^*$ vérifient :

- **$f=\Theta(f)$**
- **$f=\Theta(g) \Leftrightarrow g=\Theta(f)$**
- **$f=\Theta(g)$ et $g=\Theta(h) \Rightarrow f=\Theta(h)$**

La propriété **$f=\Theta(g)$** concerne le comportement asymptotique de f. Comme l’indique la propriété suivante (établi en TD).

Propriété[Asymptote] : Si deux fonctions f, g $\mathbb{N}^*\to\mathbb{N}^*$ sont *asymptotiques*, c'est à dire égales sur un intervalle de la forme [N,+oo[, alors elles sont équivalentes (on a f=Θ(g)).

3.2 2ème simplification ; notation O

Quand nous n'arrivons pas à approximer une fonction complexité, nous nous contentons de la majorer !

Définition f=O(g)

Soient deux fonctions f et g de type $\mathbb{N}^*\to\mathbb{N}^*$.

f est *majorée* par *g à une constante multiplicative près, noté f=O(g), si* il existe un réel B>0 vérifiant : $f \leq B.g$

c'est à dire vérifiant $\forall\ n \in \mathbb{N}^*$, $f(n) \leq B.g(n)$. L'ensemble des fonctions majorées par g à une constante multiplicative près est notée O(g).

fin_definition

Comme cela a été indiqué en introduction, l'évaluation selon *O(g)* est réalisée quand on n'arrive pas à fournir l'évaluation selon Θ(g)**.** Il s'agit ainsi d'une approximation comme l'indiquent les propriétés suivantes (que nous considérons trivales) :

Propriété[Approximation]

Toute fonction f : $\mathbb{N}^*\to\mathbb{N}^*$ vérifie : **$\Theta(f) \subset O(f)$**

Contrairement aux ensembles **Θ(g),** les ensembles ne forment pas des classes d'équivalence mais des classes hiérarchiques. La relation *f=O(g)* n'est pas d'équivalence mais seulement transitive. Comme l'indique la propriété suivante :

Propriété[Transitivité]

Toutes fonctions f, g et h $\mathbb{N}^*\to\mathbb{N}^*$ vérifient :

- **f=O(f)**
- **f=O(g) et g=O(h) ⇒ f=O(h)**

3.3 une infinité non dénombrable de classes Θ et de classes O

Pour définir la fonction " $\forall\ n \in \mathbb{N}^*,\ n\mapsto n^2$ " est utilisée l'expression très courte " n^2 ". Ainsi, $O(n^2)$ est l'abréviation de la notation $O(\forall\ n \in \mathbb{N}^*,\ n\mapsto n^2)$ et désigne ainsi l'ensemble des fonctions majorée par la fonction quadratique.

Il existe une infinité de classes Θ et de classes O. Citons ici quelques unes. Pour chacune d'entre elles, nous fournirons une ou plusieurs équations récursives dont elles sont solutions.

Pour compléter chacune des équations récursives ci-dessous, il suffit d'ajouter l'égalité terminale : f(n)=1 si n=1.

3.3.1 quelques classes Θ

nom	**notation**	Définition récursive
constante	**Θ(1)**	f(n) = f(n-1)
logarithmique étoile	**Θ(log*(n))**	f(n) = 1 + f(log(n-1))
logarithmique	**Θ(log(n))**	f(n) = 1 + f(n/2)
linéaire	**Θ(n)**	f(n) = 1 + f(n-1) f(n)=1+2.f(n/2)
	Θ(n.log(n))	f(n) = n + 2.f(n/2)
quadratique	**Θ(n^2)**	f(n) = n +f(n-1) f(n)=1+4.f(n/2)
cubique	**Θ(n^3)**	f(n) = n^2 + f(n-1)
polynomial	**Θ(n^k)**	f(n) = n^{k-1} + f(n-1)
exponentielle en base 2	**Θ(2^n)**	f(n) = 2.f(n-1)
exponentielle en base k	**Θ(k^n)**	f(n) = k.f(n-1)
factorielle	**Θ(n!)**	f(n) = n.f(n-1)
double exponentielle	**Θ(2^{2^n})**	f(n) = $f(n-1)^2$
exponentielle étoile	**Θ(2^{*n})**	f(n) = $2^{f(n-1)}$
Infini	**Θ(+oo)**	f(n) = 1+f(n)

Notons dans cette hiérarchie des fonctions mutuellement réciproques : comme par exemple, log(n) et 2^n, log*(n) et 2^{*n} mais aussi, d'un certain point de vue, 1 et +oo .

III.3.2 quelques classes O

La liste ci-avant permet de définir un ensemble de classes O en remplaçant dans chaque équation le signe "=" par le signe "≤". Comme par exemple ici :

logarithmique	**O(log(n))**	f(n) ≤ 1 + f(n/2)

4) Comment calculer la fonction complexité d'un algorithme ?

Comme vu en introduction, il n'existe pas et n'existera jamais de méthode sûre (d'algorithme) qui permet de calculer ni la complexité en temps ni celle en espace d'un algorithme. Et ce même en utilisant les approximations Θ ou O. Cependant, pour la plupart des algorithmes que vous écrirez, il vous sera facile avec un peu de méthode de calculer ces complexités O. De même pour les complexités Θ, mais cela exigera plus de travail et de méthode.

4.1 Règle mise en séquence et branchement conditionnel

La seule règle générale concerne la mise en séquence et le branchement conditionnel. Si #(A) désigne par exemple le nombre d'instructions élémentaires exécutées par l'instruction A (ou l'algorithme A), il vient :

$$(1)\ \#(A\ ;\ B\ )\ \leq\ \#(A) + \#(B)$$

$$(2)\ \#(\texttt{si}\ \ A\ \texttt{alors}\ B\ \texttt{sinon}\ C\ )\ \leq\ \#(A) + \#(B) + \#(C)$$

Dans le cas de ma mise en séquence, il s'agit d'une inégalité et non d'une égalité, car A peut contenir une instruction `retourner` et ainsi empêcher B de s'exécuter. Même réflexion en ce qui concerne le branchement conditionnel. Il en découle qu'il n'existe aucune règle simple et générale pour majorer #(`tantque` A `faire` B).

4.2 Définition récursive de la complexité d'un algorithme récursif

1er exemple

Dans le cas d'algorithmes récursifs sans utilisation de boucles, les règles (1) et (2) permettent d'obtenir facilement des inéquations récursives pour la complexité en temps. Par exemple un algorithme (non optimal) qui décide si un entier a appartient à un tableau T trié :

```
fonction naïf(a : entier ; T : tableau d'entier) : booléen
debut
      si longueur(T)= 0 alors
            retourner faux();
      m ← 1+longueur(T)/2  ;
      si a = T[m] alors
            retourner vrai();
      si a < T[m] alors
            retourner naïf(a , sousTableau(T,1,m-1));
      retourner naïf(a , sousTableau(T,m+1,longueur(T)))
fin
```

La fonction auxiliaire `sousTableau(T,i,j)` avec $1 \leq i \leq j \leq$ `longueur(T)` retourne un tableau contenant les j-i+1 valeurs T[i],..,T[j]. Cette fonction est donc de complexité en temps et en espace égale à Θ(j-i).

définition récursive de la fonction complexité

La complexité en temps désigne celle dans le pire des cas. Le pire des cas est celui où l'entier a n'appartient pas à T. Nous obtenons alors comme première définition récursive de la fonction complexité en temps de `naif` notée $\#_{naif}(n)$ avec n la longueur de T :

- $\#_{naif}(n) = 1$ si n= 0
- $\#_{naif}(n) = n + \#_{naif}(n/2)$ sinon

obtention de la fonction complexité

La résolution de cette équation donne : $\#_{naif}(n) = \Theta(n)$.
La complexité en espace est fournie par la même équation et est donc linéaire aussi.

Clairement, cet algorithme n'est pas du tout optimal car il gaspille du temps et de l'espace à écrire log(n) sous-tableaux de tailles respectives n/2, n/4, … 1.

2ème exemple

Pour calculer la fonction complexité, il faut identifier des valeurs pertinentes autres que la simple complexité des données fournies en entrée.
Voici un algorithme récursif optimisé de la recherche dichotomique :

```
fonction ∈(a : entier ; T : tableau d'entier) : booléen
debut
      retourner ∈REC(a,T,1,longueur(T))
fin

fonction ∈REC(a : entier;T:tableau d'entier ; i,j:entiers) : booléen
debut
      si j<i alors
            retourner faux();
      m <- 1+(j+i)/2  ;
      si a = T[m] alors
            retourner vrai();
      si a < T[m] alors
            retourner ∈REC(a , T, i , m-1)
      ;
      retourner ∈REC(a , T , m+1 , j)
fin
```

Notons # la fonction complexité en temps de $\in$`REC` .Si nous choisissons n égal à la complexité de l'entrée, c.a.d la longueur de T, nous obtenons comme définition récursive de # l'inéquation $\#(n) \leq 1 + \#(n)$ dont la résolution donne $\#(n)=O(+\infty)$ qui est vérifiée par toute fonction de complexité ! En clair, nous ne savons rien de #(n).

Dans le cas présent, l'information pertinente est la longueur du tableau fournie symboliquement en entrée grâce au triplet (T,i,j) à savoir la quantité n:=j-i+1. Sous cette

hypothèse, nous obtenons dans le pire des cas (a $\notin$ T : voir exemple plus haut) comme définition récursive : $\#(n) = 1 + \#(n/2)$ dont la résolution est $\#(n)=\Theta(\log(n))$.

En ce qui concerne la complexité en espace, la définition récursive est identique ainsi donc que sa résolution. Ainsi, les deux complexités temps et espace sont logarithmiques selon une approximation Θ.

remarque :

Le dernier exemple est facilité car il est facile de trouver des données en entrée qui nécessitent le maximum de calculs (ici a $\notin$ T). Identifier ces données est parfois beaucoup plus difficile.

4.3 Résolution dans $\mathbb{N}^*\rightarrow\mathbb{N}^*$ d'une équation récursive

Il existe quelques très rares équations que l'on peut résoudre facilement. Par exemple, la propriété suivante permet de résoudre rapidement certaines équations associées à des algorithmes itératifs. Par exemple :

- l'équation $f(n)=n+f(n-1)$ a pour solution $\Theta(n^2)$.
- l'équation $f(n)=n^2.\ln(n)+f(n-1)$ a pour solution $\Theta(n^3.\ln(n))$.

Propriété

Soit g: $\mathbb{N}^*\rightarrow\mathbb{N}^*$ une fonction croissante dans $O(K^n)$ pour quelque réel K,

l'équation : $f(n)=g(n)+f(n-1)$ admet pour solution dans $\mathbb{N}^*\rightarrow\mathbb{N}^*$ une fonction dans $\Theta(G)$ où G est une primitive de g.

preuve

La quantité f(n) est égale à g(0)+g(1)+..+g(n) est est minorée par G(n) et est majorée par G(n+1) et qui vérifie $G(n+1)\leq 1/K.G(n)$. Il vient $f(n) = \Theta(G)$.

fin

Cette seconde propriété est utile dans le cas d'algorithmes récursifs :

Propriété :

$$\mathbf{f(n)=n^k+a.f(n/b)}$$

où k, a et b sont trois réels vérifiant $k\geq 0$, $a\geq 1$ et $b>1$

admet pour solution dans $\mathbb{N}^*\rightarrow\mathbb{N}^*$:

- $f(n) = \Theta(n^k)$ si $b^k>a$
- $f(n) = \Theta(n^k.\ln(n))$ si $b^k=a$ ce qui équivaut à $k=\ln(a)/\ln(b)$
- $f(n) = \Theta(n^{\ln(a)/\ln(b)})$ sinon

Cette propriété s'étend naturellement à toute équation : $f(n)=\Theta(n^k)+a.f(n/b)$

preuve :

Soit f $\mathbb{N}^*\rightarrow\mathbb{N}^*$ une fonction définie par $\mathbf{f(n)=n^k+a.f(n/b)}$ où k, a et b sont trois réels vérifiant $k\geq 0$, $a\geq 1$ et $b>1$. Il en découle les égalités :

$f(n) = n^k+a.(n/b)^k+a^2.(n/b)^{2k}+..+a^L.(n/b)^{kL}$ avec $L:=ln_b(n)$ (c.a.d défini par $n=b^L$)

$f(n) = n^k(1+(a/b^k)+(a/b^k)^2+..+(a/b^k)^L)=n^k(1+\propto+..+\propto^L)$ où $\propto := a/b^k$.

f(n) est égal à L si $\propto=1$ ou à $(\propto^{L+1} -1)/(\propto -1)$ sinon, il vient :

- Si $a/b^k=1$, on a : $f(n) = n^k$. L c'est à dire $f(n)=\Theta(n^k.ln(n))$.
- Si $a/b^k<1$, on a : $f(n) = \Theta(n^k)$ car $\propto^{L+1} = \Theta(1)$ et $(\propto^{L+1} -1)/(\propto -1) = \Theta(1)$.
- Si $a/b^k>1$, on a $f(n) = \Theta(n^k(a/b^k)^L)=\Theta(a^L)=\Theta(n^{ln(a)/ln(b)})$ car $a^L=(a^{lnb(n)})=(a^{lna(n)})^{ln(a)/ln(b)}=n^{ln(a)/ln(b)}$

fin_preuve

Exemple 1 :

L'équation $f(n)=n+2.f(n/2)$ a pour solution $f(n)=\Theta(n.ln(n))$. Car conséquence directe de la propriété précédente et des égalités $(k,a,b)=(1,2,2)$ et $2^1=2$.

Exemple 2 :

Certaines équations peuvent être représentées graphiquement et se résoudre ainsi. Voir les graphiques du répertoire courant associés aux équations :

f(n)=n+2.f(n/2)
f(n)=1+2.(n/2)
f(n)=1+f(n/2)
f(n)=n+f(n-1)
f(n)=1+2.f(n-1)

Il est toujours plus facile de trouver la correction d'une solution que de trouver la solution

Cette règle encourage tout scientifique ayant trouvé une solution après d'éventuelles longues et laborieuses réflexions à tester sa solution en s'assurant qu'elle vérifie ou non l'équation.

Par exemple : n.log(n) est bien solution de f(n)=n+2f(n/2)

car $\forall$ N f(N)=N.log(N) entraîne $n+2f(n/2)=n+2.n/2.log(n/2)=n+n(log(n)-1)=nlog(n)=f(n)$.

Chapitre 6
Algorithmes Divide and conquer

Les termes anglais “Divide and conquer”, francais “Diviser pour Régner” ou latin “Divide ut Imperes” désignent une, si ce n’est la, méthode la plus célèbre pour trouver un algorithme récursif permettant de résoudre tout problème algorithmique.

Cette célébrité est due à la puissance de cette méthode qui peut fournir assez rapidement:

- une écriture d’un algorithme,
- sa complexités en temps
- sa complexité en espace
- ainsi que sa preuve de correction,

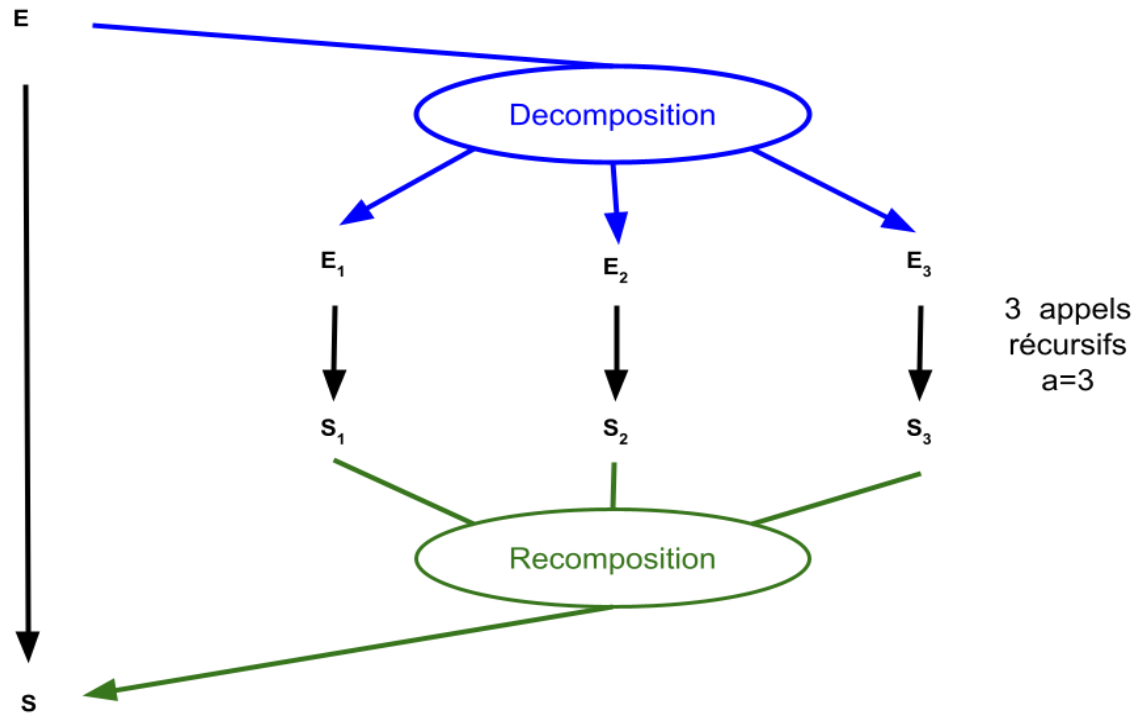
Toutes quatre définies récursivement naturellement !

1) Définition

A la terminologie “Diviser pour Régner”, nous préférons une moins poétique mais plus fondée “Décomposer pour recomposer". Le schéma général d’un algorithme Décomposer pour recomposer comprend trois étapes :

1. la décomposition : elle décompose l’entrée E en plusieurs entrées E_1, …, E_a. Notons **a** leur nombre.
2. la récursion : elle réalise a appels récursifs sur les entrées E_1,…, E_a.
3. la recomposition : elle calcule la sortie S à partir des a sorties S_1,…, S_a.

Voici un dessin de ce schéma associé à un algorithme requérant a=3 appels récursifs :



2) Calcul des complexités temps et espace

La définition récursive de l'algorithme fournit une définition récursive de la complexité en temps. Notons :

- n la complexité d'une entrée E
- #décomposition la complexité en temps de l'algorithme de décomposition
- $r_i(n)$ la complexité (maximale) de chaque entrée E_i.
- #recomposition la complexité en temps de l'algorithme de décomposition

La complexité en temps de l'algorithme, notée #algo, est ainsi définie. Pour tout entier n :

$$\begin{aligned} \#algo(n) = \; & \#décomposition(n) \\ + \; & \#algo(r_1(n)) \\ + \; & \ldots \\ + \; & \#algo(r_a(n)) \\ + \; & \#recomposition(n) \end{aligned}$$

Cela permet de caractériser certains algorithmes polynomiaux.

Propriété :

Si :

- il existe un réel $k \geq 0$ tel que : $\#décomposition(n)+\#recomposition(n)=\Theta(n^k)$
- il existe un réel $b>1$ tel que pour tout entier $i \in [1,a]$, on a : $r_i(n) = 1/b \; . \; n$

alors on a :

$$\#algo(n) = n^k + a \; . \; \#algo(n/b)$$

L'équation précédente fournit immédiatement la définition de #algo (voir chapitre précédent).

Faute de grives, mangeons des merles !

Naturellement, il arrive souvent que l'on ne dispose pas d'évaluations exactes mais de simples majorations. Dans ce cas, on utilise la propriété suivante:

Propriété :

Si :

- il existe un réel $k \geq 0$ tel que : $\#décomposition(n)+\#recomposition(n)=O(n^k)$
- il existe un réel $b>1$ tel que pour tout entier $i \in [1,a]$, on a : $r_i(n) \leq 1/b \; . \; n$

alors on a :

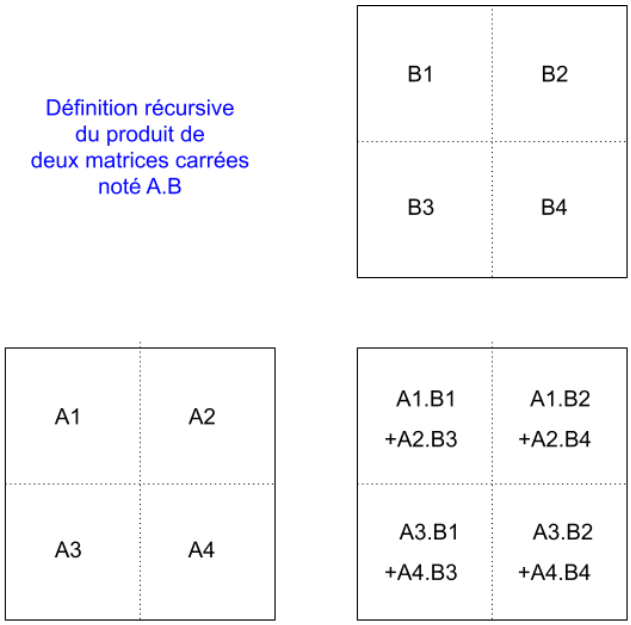
$$\#algo(n) \leq n^k + a \; . \; \#algo(n/b)$$

Exemple 1

Considérons la définition récursive du produit de deux matrices d'entiers A et B fournie par le schéma ci-dessous. Elle réalise a=8 appels récursifs sur 8 entrées Ai.Bj de complexité chacune le quart de celle en entrée (b=¼). Les opérations de composition et de recomposition sont linéaires : la décomposition peut être réalisée symboliquement en temps constant ; la recomposition nécessite la création de mémoire et l'exécution de simples additions.

Si n désigne la complexité de l'entrée, l'équation complexité en temps est: $\#(n) = n + 8.\#(n/4)$ et admet pour solution $\Theta(n^{3/2})$.

En bref, cet algorithme est donc de complexité en temps égal à celui classique itératif qui réalise n produits ligne-colonne de complexité chacun √n.



Exemple 2 : l'algorithme de Strassen est la première amélioration historique du produit matriciel : il s'agit d'un algorithme qui contient non pas 8 mais seulement 7 appels récursifs sur des quarts de matrices. Sa complexité en temps vérifie #algo(n) = n + 7.#algo(n/4) et est donc $\Theta(n^{\log(7)/\log(4)})$ inférieure à $\Theta(n^{1,5})$ (on a log(7)/log(4)≃1,403 < 1,5=3/2).

3) La décomposition impose la recomposition, ou inversement

Divide and Conquer comprend 2 principales façons de résoudre un problème algorithmique :

1. soit on impose une décomposition et on trouve la recomposition induite.
2. soit on impose une recomposition et on trouve la décomposition induite.

Un exemple commun de cette méthode est le problème du tri d'un tableau d'entiers.

```
TRI
Entrée :   un tableau d'entiers T
Sortie :   (par effet de bord) un tableau trié U contenant les
           éléments de T en même nombre.
```

remarque préalable : découpe symbolique des tableaux

Pour découper en temps constant les tableaux, on les découpe de façon symbolique en manipulant des indices. Ainsi, tout tableau est fourni en entrée à l'aide d'un tableau et deux indices i et j de la façon suivante :

```
fonction tri(T:tableau d'entier)
debut
      triRec(T,1,longueur(T))
fin
```

3.1 Où la décomposition est une simple découpe en 2 moitiés : le tri fusion

Tentons la plus simple des décompositions : la décomposition d'un tableau en son milieu réalisée en temps constant. Le format de cet algorithme est donc le suivant :

```
fonction triFusion(T:tableau d'entier)
debut
      triFusionREC(T,1,longueur(T))
fin

fonction triFusionREC(T:tableau d'entier ; i,j : entier)
debut
      si i < j alors
      debut
            m ← milieu(i,j);
            triFusionREC(T,i,m);
            triFusionREC(T,m+1,j);
            fusion(T,i,m,j)
      fin
fin
```

où `FUSION` est le problème suivant :

`E : un tableau T d'entier, i,m,j : entiers avec 1≤i≤m<j≤|T|`
`S : (par effet de bord sur T) un tableau U d'entiers tel que:`

- `U[1..i-1]=T[1..i-1]`
- `U[j+1..|U|]=T[j+1..|T|]`
- `U[i..j] et T[i..j] possèdent les mêmes valeurs`
- `U[i..j] est trié si T[i..m] et T[m+1..j] sont triés`

Puisque `FUSION` admet une solution linéaire en temps (voir exercice), la complexité en temps #(n) de `triFusion` vérifie $\#(n)=n+2.\#(n/2)$, est $\Theta(n.\ln(n))$ et est donc optimale.

Puisque `FUSION` est de complexité en espace linéaire, `triFusion` est de complexité en espace linéaire. Il existe de meilleurs algorithmes ayant une même complexité en temps $\Theta(n.\ln(n))$ mais de complexité en espace $\Theta(\ln(n))$. C'est l'objet de la prochaine section.

3.2 Où la recomposition est de ne rien faire : tri médian ou QuickSort

Les algorithmes suivants se caractérisent par la plus simple des recompositions, qui consiste à ne rien faire. Pour que cet algorithme soit correct, il est nécessaire et suffisant que toutes les valeurs entre i et m soient inférieures à toutes celles entre m+1 et j. Cela se fait au moyen d'un *pivot* autour duquel sont réparties à sa gauche les valeurs inférieures de T et à sa droite les autres valeurs. L'algorithme est :

```
fonction triPivot(T:tableau d'entier ; i,j : entier)
debut
      si i < j alors
      debut
            indicePivot ← decomposition_Pivot(T,i,j);
            triPivot(T,i,indicePivot-1);
            triPivot(T,indicePivot+1,j)
      fin
fin
```

La définition du pivot et de la modification du tableau T[i..j] autour du pivot T[m] est décrite formellement ici :

`DECOMPOSITION_PIVOT` est le problème suivant :

`E : un tableau T d'entier, i, j : entiers avec 1≤i<j≤|T|`
`S : un entier indPivot appartenant à [i,j]`
`et un tableau U d'entiers (retourné par effet de bord) tels que:`

- `U[1..i-1]=T[1..i-1]`
- `U[j+1..|U|]=T[j+1..|T|]`
- `U[i..j] et T[i..j] contiennent les mêmes valeurs`
- `tout élément de U[i..indPivot-1] est inférieur ou égal à U[indPivot]`
- `U[indPivot] est inférieur à tout élément de U[indPivot+1..j]`

Quelle meilleure décomposition pour le problème du `TRI` ?

Il existe plusieurs façons de choisir le pivot. Nous en distinguons 2 :

- le choix du pivot médian ; le nom du tri associé est `triMedian`
- le choix aléatoire du pivot ; le nom du tri associé est `quickSort`

Voici une comparaisons des qualités de ces 2 algorithmes :

	`triMedian`		`quickSort`
complexité en espace	Θ(ln(n))	=	Θ(ln(n))
complexité en temps dans le pire des cas	Θ(n.ln(n))	<	Θ(n^2)
complexité en temps en moyenne	Θ(n.ln(n))	=	Θ(n.ln(n))
simplicité du code	moins simple !	>	simplissime

un champion : `quickSort`

La grande simplicité du calcul de pivot de `quickSort` entraîne une constante multiplicative K de la complexité en temps en moyenne K.ln(n).n de `quickSort` très inférieure à celle de `triMedian`. Et bien que `quickSort` ait une probabilité non nulle sur toute entrée d'être plus lent (Θ(n^2)), `quickSort` est préféré à `triMedian`.

4) Trouver le bon problème à récursiver

Les chapitres précédents ont montré comment trouver une solution algorithmique récursive A à un problème **P**. Malheureusement, cette méthode ne suffit pas pour trouver des solutions efficaces (en temps ou en espace). Il est souvent nécessaire de chercher un problème plus gros **P+** qui "contient" **P** et dont toute solution algorithmique induit une solution de **P** .

4.1 Réduction logique de problème

Une première tentative est d'augmenter **P** en conservant son type, en "augmentant" les entrées de **P** mais en "diminuant" les sorties :

Définition réduction logique

Un problème **P** est *contenu* dans un problème **Q**, noté ***P≤Q*****,** si ils ont même type et si :

- toute entrée de **P** est une entrée de **Q**
- pour toute entrée E de **P**, toute sortie associée à E selon **Q** est associée à E selon **P**.

Les deux propriétés suivantes assez immédiates seront établies en TD :

Propriété : La relation ≤définie ci-avant est une relation d'ordre, c'est à dire transitive, réflexive et antisymétrique.

Propriété :

Si deux problèmes **P** et **Q** vérifient **P≤Q**,

alors toute solution algorithmique de **Q** est une solution algorithmique de **P**.

4.2 Une autre réduction de problème : la **réduction linéaire**

La réduction **P≤Q** vue précédemment est rarement suffisante. C'est le cas par exemple du problème du `tri` où l'entrée T est transformée en un triplet (T,i,j) qui est l'entrée du problème `triREc`. Il est souvent aussi nécessaire d'enrichir les sorties. Voici la définition :

Définition : réduction linéaire

Un problème **P** se *réduit linéairement* en un problème **Q**, noté ***P≤$_L$Q***, si il existe deux algorithmes `A` et `B` de complexité en temps linéaire tels que :

1. `A` transforme toute entrée E de **P** en une entrée A(E) de **Q**
2. `B` transforme tout couple de la forme (E,S) en une sortie B(E,S) de **P,** sous réserve que E est une entrée de **P** et que S est une sortie de **Q** associée à A(E).

Il s'ensuit que toute solution de **Q** induit une solution de **P** de même complexité en temps:

Propriété :

Si P se réduit linéairement en un problème **Q**,

alors toute solution algorithmique `solutionQ` de **Q** de complexité en temps au moins linéaire induit une solution algorithmique de **P** de même complexité en temps.

preuve

Sous ces hypothèses et notations, l'algorithme induit est :

```
fonction solutionP(E : TYPE_ENTREE_P)TYPE_SORTIE_P
début
      retourner B(E,solutionQ(A(E)))
fin
```

Sa complexité en temps est le max des complexités en temps de `solutionQ`, `A` et `B` c'est à dire la complexité en temps de `solutionQ`.

fin

Autres réductions

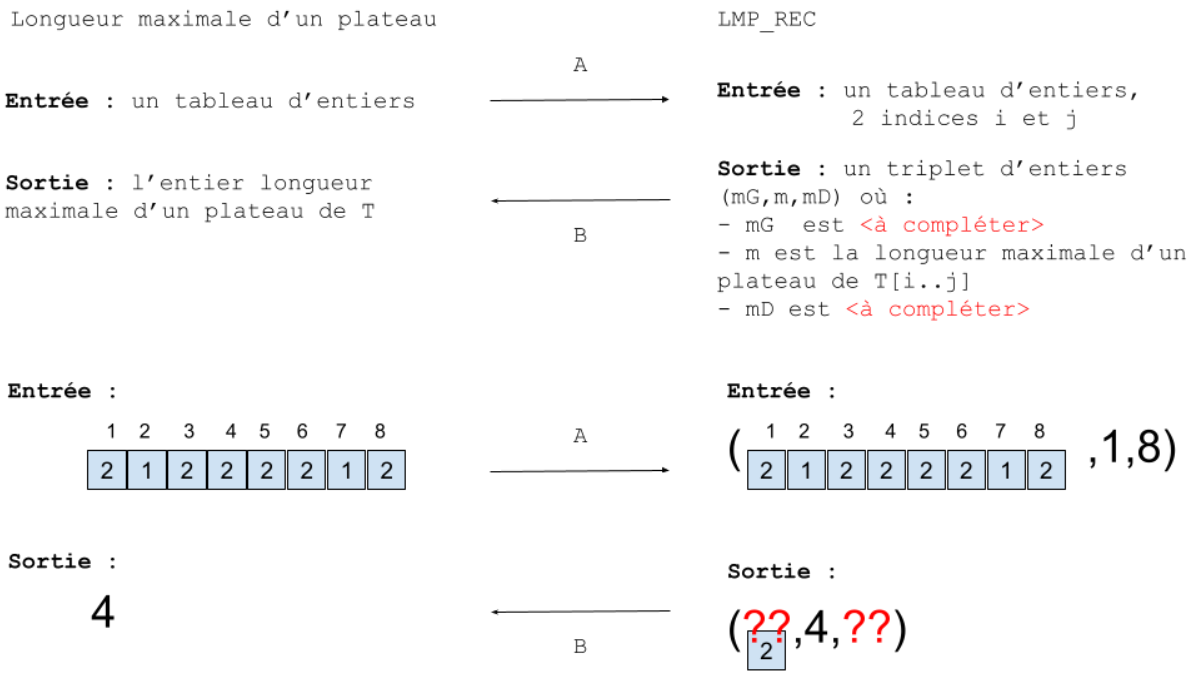
Naturellement, si l'on s'intéresse à résoudre un problème simplement en temps polynomial, on pourra s'intéresser à des réductions polynomiales (en imposant que les transformations A et B soient de complexité en temps polynomiale, et non pas strictement linéaires). La relation est notée $\leq_P$.

Observez les implications $P \leq Q \Rightarrow P \leq_L Q$ *et* $P \leq_L Q \Rightarrow P \leq_P Q$

Exemple :

Voici un exemple de réduction linéaire (en fait "constante") permettant résoudre de façon optimale en temps le calcul de longueur maximale d'un *plateau* d'un tableau T.

La transformation linéaire `A` est celle classique qui permet de découper un tableau en plusieurs sous-tableaux (ici 2) à l'aide de simples manipulations d'indices toutes en temps constant Θ(1). La transformation extrait simplement la bonne information.

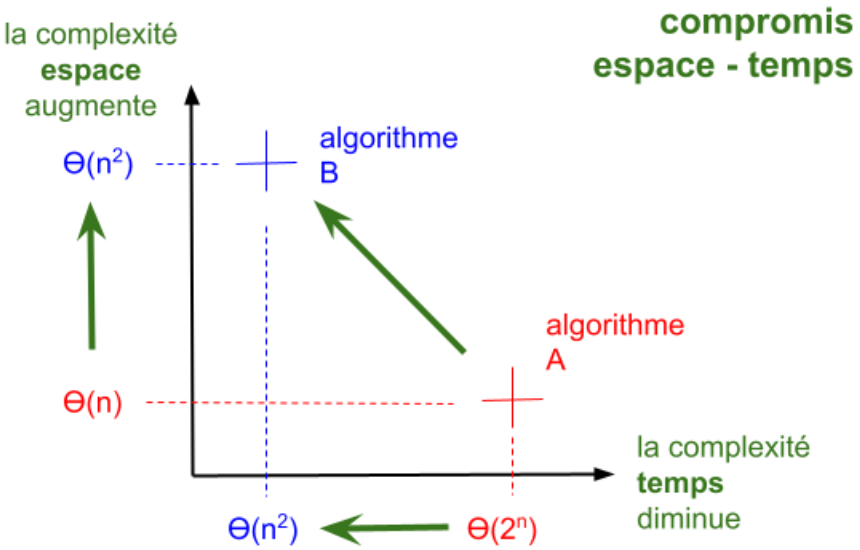


Chapitre 7
Programmation dynamique

1) compromis espace-temps

Nous présenterons ici une méthode qui permet de transformer un algorithme A en un algorithme B beaucoup plus rapide. Cela s'effectue selon un "*compromis espace-temps*" ce qui signifie que l'on améliore la complexité en temps en augmentant la complexité en espace.

Le diagramme ci-dessous l'illustre par un exemple :

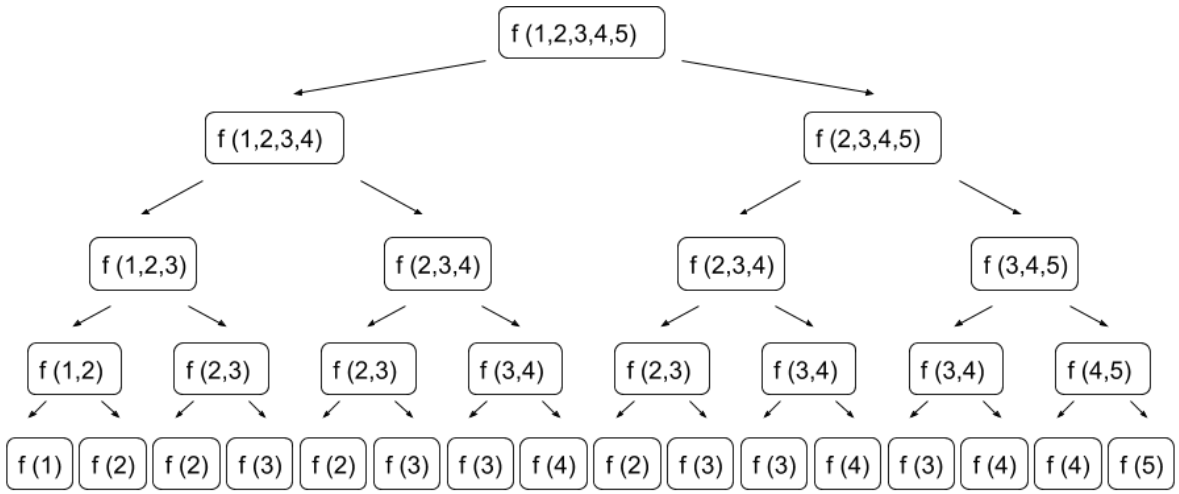


Un algorithme très lent, à améliorer

Considérons l'algorithme suivant qui prend en entrée une séquence d'entiers et qui retourne un entier.

```
fonction f(E:séquence d'entiers) entier
début
      si longueur(E)=1 alors
         retourner uniqueElement(E)
      sinon
         retourner calc(f(supprimeFin(E)),f(supprimeDébut(E)))
fin
```

Nous supposons ici que toutes les fonctions auxiliaires sont de complexités temps et espace constantes. La complexité en temps #f de f vérifie l'équation #f(n)=1+2.#f(n-1) et est donc exponentielle. Pour comprendre la raison de ce coût exponentiel, il suffit de regarder l'arbre des appels. Est dessiné ci-dessous l'arbre des appels de la fonction f sur l'entrée qui est la séquence (1,2,3,4,5). Nous observons que f est appelé 2 fois sur l'entrée (2,3,4), 3 fois sur l'entrée (3,4), 6 fois sur l'entrée (3).



La raison du coût en temps exponentiel est liée au simple fait que certains calculs sont réalisés plusieurs fois.

2) Un premier algorithme dynamique générique

Pour supprimer la redondance des calculs, il suffit d'utiliser une mémoire supplémentaire MEM qui mémorise l'ensemble des entrées déjà traitées et leurs sorties associées. Cela peut se faire à l'aide de trois fonctions auxiliaires :

- `appartient(E,MEM)` qui indique si `MEM` contient une valeur associée à l'entrée `E`
- `valeur_associée_à(E,MEM)` qui retourne la valeur associée à l'entrée `E`
- `ajout(E,S,MEM)` qui ajoute à `MEM` le couple `(E,S)`
- `ensembleVide()` qui initialise `MEM` à vide

La réécriture de f se fait "mécaniquement" de la façon suivante :

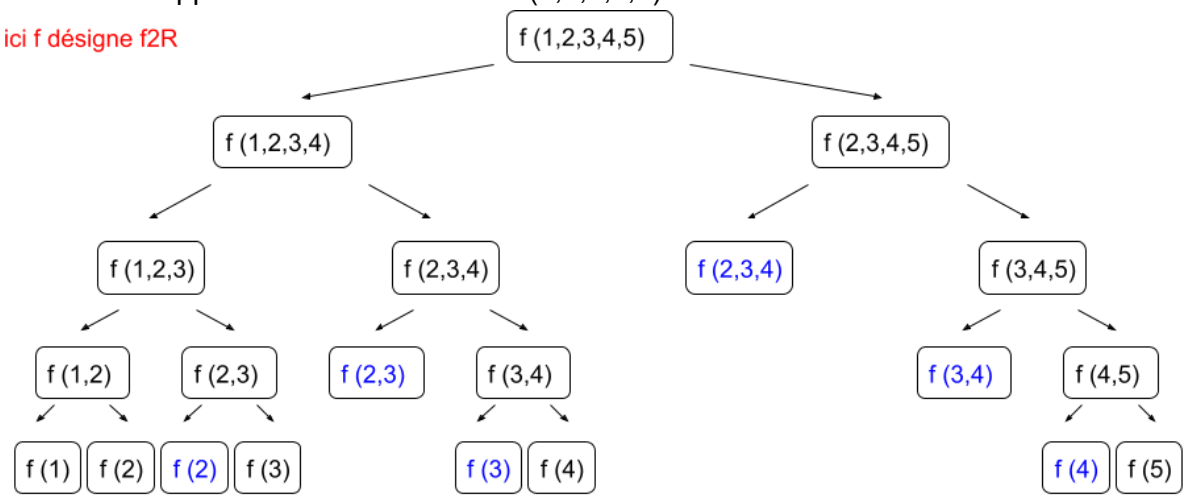
```
fonction f2(E:séquence d'entiers) entier
début
      retourner f2R(s,ensembleVide())
fin

fonction f2R(E:séquence d'entiers , MEM : type_a_définir) entier
début
      si appartient(E,MEM) alors
            retourner valeur_associée_à(E,MEM)
      ;
      si longueur(E)=1 alors
         S ← uniqueElement(E)
      sinon
         S ← calc(f2R(supprimeFin(E),MEM),f2R(supprimeDebut(E),MEM))
      ;
      ajout(E,S,MEM)
      ;
      retourner S
fin
```

L'efficacité de ce nouvel algorithme dépend des propriétés de cette mémoire, c'est-à-dire de sa taille en octets ainsi que de la complexité en temps (et en espace) des fonctions auxiliaires `appartient()`, `valeur_associée_à()`, `ajout()` et `ensembleVide()`.

Le nombre d'appels de f2R est $\Theta(n^2)$

L'arbre des appels de f2aux sur l'entrée (1,2,3,4,5) est dessiné ci-dessous :



Il y 2 sortes d'appels :

- les noirs : des appels sur des entrées non déjà calculées. Leur nombre ne peut excéder le nombre d'entrées possibles soit $\Theta(n^2)$
- les bleus : des appels sur des entrées déjà calculées. Leurs parents sont nécessairement noirs. Puisque chaque parent a deux enfants, le nombre d'appels bleus est $\Theta(n^2)$.

L'arbre des appels possède donc au plus $\Theta(n^2)$ noeuds.

Un premier algorithme dynamique complètement implémenté

Pour évaluer la complexité en temps et en espace, il faut fournir une implémentation des fonctions

- `appartient(E,MEM)`
- `valeur_associée_à(E,MEM)`
- `ajout(E,S,MEM)`
- `ensembleVide()`

Une solution ici est très simple est réalisée à l'aide d'une simple matrice de booléen `APPART` et d'une matrice d'entiers `VALEUR`.

L'algorithme devient alors :

```
fonction f3(E:tableau d'entiers) : entier
début
      n ← longueur(E) ;
      APPART ← construire_matrice_carrée(n,faux());
      VALEUR ← construire_matrice_carrée(n,118) ;
      retourner f3R(1,n,E,APPART,VALEUR)
fin
```

```
fonction f3aux    (
                  i,j : entier ,
                  E : tableau d'entier,
                  APPART : matrice carrée de booléens ,
                  VALEUR : matrice carrée d'entiers
                  )
                  entier
début
      si APPART[i][j] alors
            retourner VALEUR[i][j]
      ;
      si i=j alors
            S ← E[i]
      sinon
            S ← calc(
                  f3R(i,j-1,E,APPART,VALEUR),
                  f3R(i+1,j,E,APPART,VALEUR)
                  )
      ;
      APPART[i][j] ← vrai() ;
      VALEUR[i][j] ← S ;
      retourner S
fin
```

Complexité temps et espace de f3R est $\Theta(n^2)$

L'arbre des appels est similaire à celui de `f2R` donc possède $\Theta(n^2)$ noeuds.
Les opérations lecture et écriture dans les tableaux `APPART` et `VALEUR` étant constantes, il vient :

- la complexité en temps de `f3R` est $\Theta(n^2)$.
- la complexité en espace de `f3R` est $\Theta(n^2)$.

Chapitre 8
Algorithmes Gloutons

Le principe d'un algorithme glouton consiste à calculer une solution en calculant successivement des éléments de cette solution. Cette approche algorithmique fournit des solutions efficaces pour un certain nombre de problèmes d'optimisation.
L'écriture de ces algorithmes est essentiellement itérative.

1) 1er exemple : tri glouton

L'algorithme le plus intuitif et le plus populaire pour trier par taille croissante un ensemble de personnes consiste à placer en 1ère position la plus petite personne et ainsi de suite. Cette approche est gloutonne car à chaque itération une partie de la solution est calculée :

1.1) Tri sélection

Une première solution est le tri par sélection qui utilise la fonction auxiliaire `indiceDuMinimum(i,T)` calculant l'indice du plus petit élément de `T[i..|T|]` :

```
fonction triParSelection(T : tableau d'entiers)
debut
      n ← longueur(T);
      pour i de 1 à n-1 faire
            echanger(i,indiceDuMinimum(i,T),T)
fin
```

Cette première implémentation de l'approche gloutonne fournit une complexité en espace constante **Θ(1)** et une complexité en temps **Θ(n^2)** donc non minimale.

1.2) Tri par tas

Une seconde implémentation gloutonne est plus sophistiquée : elle utilise une technique d'ordonnancement partiel des éléments de T appelé « *Tas* ». Cette technique permet d'extraire le minimum d'un ensemble (ou de façon duale le maximum) :

- selon une complexité en temps **Θ(log(n))**
- selon une complexité en espace constante **Θ(1)**

L'algorithme de tri est de complexité optimale en temps **Θ(log(n))** et en espace **Θ(1)** (voir https://fr.wikipedia.org/wiki/Tri_par_tas). En voici une définition :

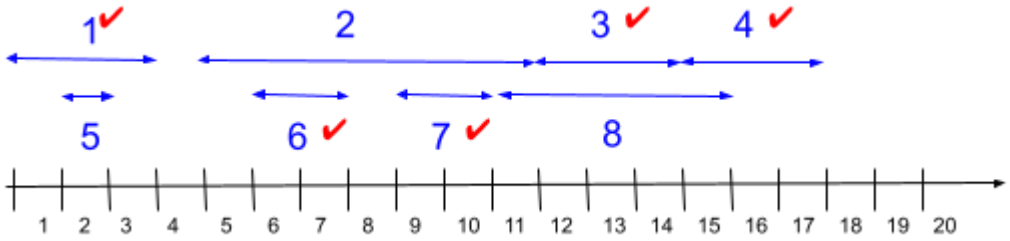
```
fonction triParTas(T : tableau d'entiers)
debut
      n ← longueur(T);
      pour i de 2 à n faire
            ajouterTas(T,i)
      ;
      pour i de n à 2 faire
            extraireMax(T,i)
fin
```

2) 2ème exemple : location d'une salle

Il s'agit ici de réaliser plusieurs évènements dans une salle. Tout événement possède une date de début et une date de fin. Il s'agit de calculer un nombre maximum d'évènements 2 à 2 compatibles, c'est à dire tels que la date de fin de l'un est strictement inférieure à la date de début de l'autre. Le problème sera étudié en TD.

Exemple :

Dans le graphique ci-dessus, est dessiné un ensemble de 8 évènements. Une solution est l'ensemble {1,3,4,6,7} (marqués ✓ sur le graphique).



Limites de l'approche gloutonne

Les algorithmes gloutons permettent un résoudre en temps polynomial plusieurs problèmes d'optimisation. Cependant, certains résistent et n'admettent pas à ce jour de telles solutions algorithmiques (de complexité en temps polynomial). Le plus célèbre d'entre est le problème du sac à dos :

SacADos
Entrée : un entier long W, un tableau d'entiers T
Sortie : un sous-tableau de T de somme au plus égale à W et maximale.

Chapitre 9
Structures mathématiques et types abstraits

Ecrire de bons algorithmes nécessite de savoir décomposer un algorithme en une fonction principale et plusieurs fonctions auxiliaires. Une partie de cette décomposition correspond à différents niveaux :

- Un haut niveau dit abstrait qui considèrent des objets logiques
- Un bas niveau dit machine qui manipule des représentations machines de ces objets.

Différentes notions de représentation sont présentées qui évoluent dans les trois strates :

- **Strate mathématique** : **données**
  Cette strate est associée au problème mathématique **P** à résoudre . Elle concerne les **données** qu'elle regroupe en structures mathématiques.
- **Strate informatique logique** : **types abstraits**
  Cette strate est associée à un algorithme **A** solution de **P**. Elle fournit les descriptions logiques des opérations manipulant les données ; ces opérations sont regroupées en Types abstraits.
- **Strate informatique machine** : **implémentations**
  Cette strate concerne à la fois la façon de représenter les données en machine mais aussi l'écriture des algorithmes implémentant les types abstraits.

1) L'exemple d'un fournisseur d'écrous

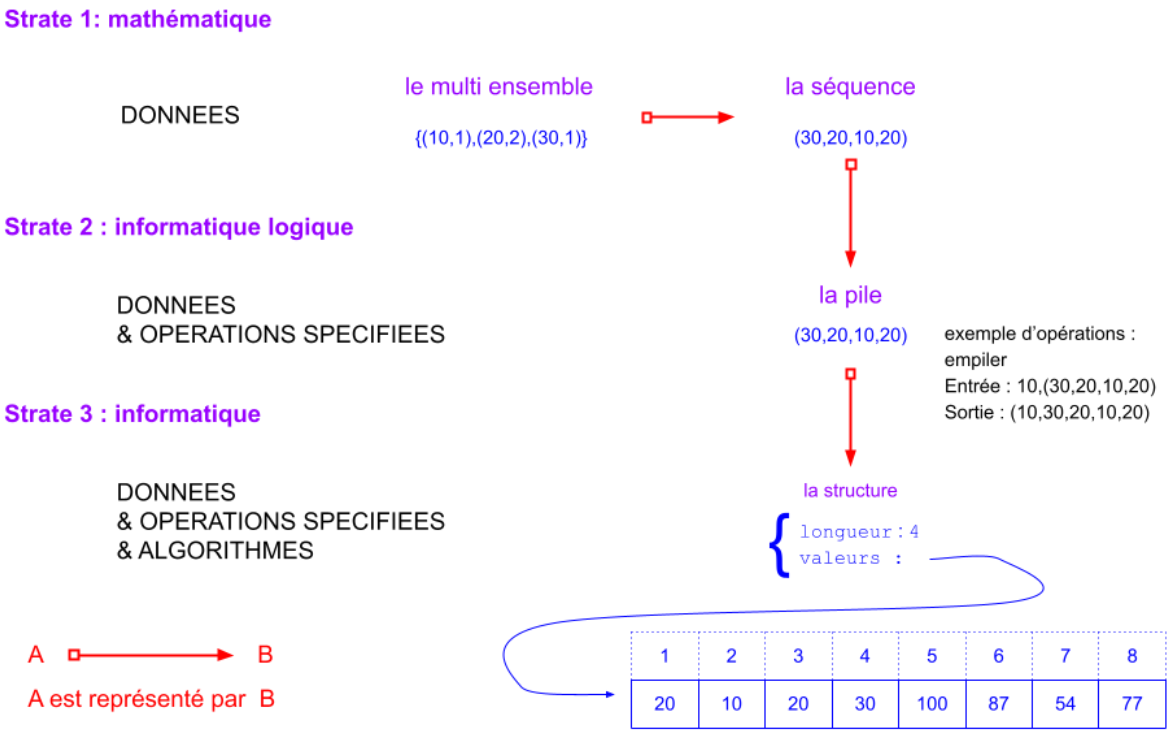
Considérons l'exemple d'une usine dirigé par sa directrice, **Anne**, produisant uniquement des écrous (un unique modèle d'un unique produit) qu'elle livre à une unique client, **Bernard,** qui assemble des automobiles.
Bernard n'exige pas une priorisation des livraisons selon l'ancienneté des livraisons mais exige simplement qu'à chaque commande corresponde un paquet mentionnant le nombre d'écrous. La représentation de ces informations peut suivre le processus suivant (voir schéma ci-dessous) :

- **strate 1** : après discussion avec **Bernard**, **Anne** décide que les commandes à livrer sont représentées par un multi-ensemble. Par exemple, {(10,1),(20,2),(30,1)} signifie qu'il faut livrer 4 paquets d'écrous de contenances 10, 20, 30 et 20.
- **strate 2** : **Anne** rencontre **Charles**, son collaborateur chef d'atelier ; ils identifient un système simple mais efficace pour entreposer chaque nouveau paquet d'écrou : tout nouveau paquet sera ajouté en haut d'une pile ; tout enlèvement d'un paquet se fera en haut de cette pile. Par exemple : l'état des commandes est (30,20,10,20) après ajout de la nouvelle commande 10 devient (10,30,20,10,20).
- **strate 3** : **Anne** rencontre **Didier,** son collaborateur informaticien ; il est décidé de représenter la pile à l'aide d'une structure contenant 2 champs `longueur` et `tableau`. Par exemple, la séquence (30,20,10,20) est représentée par la **structure** :
  - `longueur` : l'entier `4`
  - `valeurs` : le tableau `(20,10,20,30,100,87,54,77)`

Dans l'exemple précédent, Anne a organisé ses consultations selon une priorité décroissante :

- **Bernard** le client, qui est « roi » ; ce qui signifie qu'il ne doit pas être importuné par des problèmes internes de l'usine de **Anne**.
- **Charles** qui pilote la production sa préoccupation est satisfaire le client **Bernard** et sait pouvoir compter sur le soutien du service support informatique piloté par **Didier**.
- et enfin **Didier** qui pilote le service informatique et qui doit satisfaire tous les acteurs !

L'exemple de cette usine de production et de ses services illustre le fait que pour satisfaire plusieurs acteurs différents et leurs besoins légitimes, il est nécessaire de hiérarchiser ces acteurs ; dans l'exemple plus haut : d'abord le client, puis la production et enfin le service informatique.



1.1) Évolution des besoins

Naturellement, le client étant roi peut changer d'avis. Si il souhaitait prioriser la réception des écrous qui ont été commandés à la date la plus ancienne. Il faudrait solliciter le service informatique ; le simple remplacement du type pile par le type file suffirait à satisfaire le client. Si le client souhaitait disposer d'une option « commande très prioritaire / suspension de toute autre livraison » : il faudrait repenser non seulement le système informatique mais même le fonctionnement de l'atelier de production. En résumé, il est important d'avoir une vision claire, simple et donc structurée des spécifications. Cela permet d'économiser la production de codes.

2) les structures mathématiques : ensemble, multiensemble & séquence

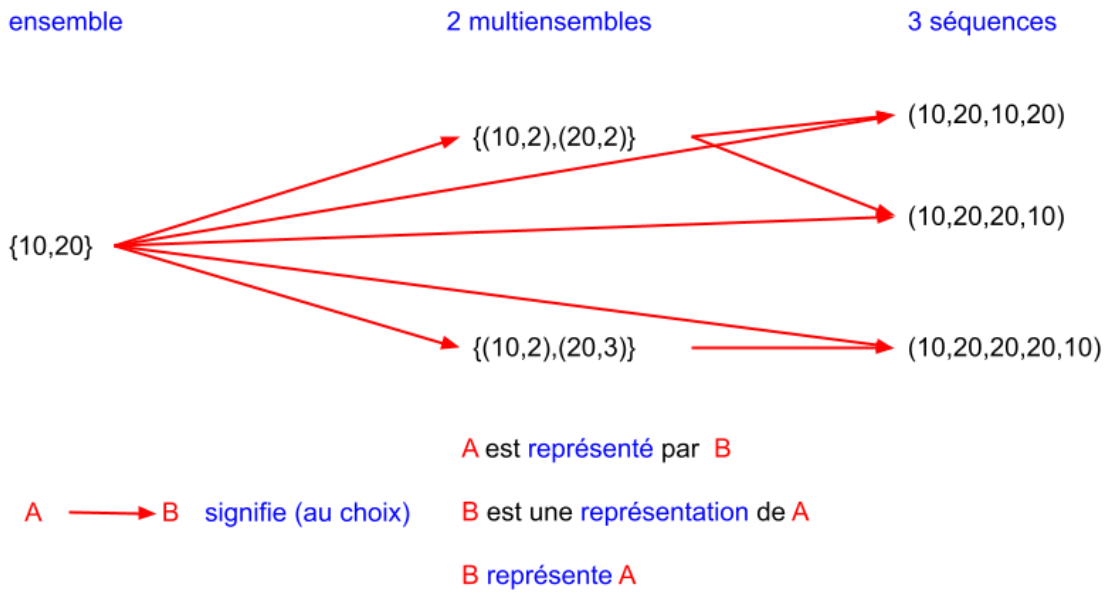
Les structures mathématiques “ensemble”, “multiensemble” et “séquence” forment une hiérarchie :

- un multiensemble M est un ensemble E augmenté d’une information d’un nouveau type : la *fréquence* de l’élément.
- une séquence S est un multiensemble M augmenté d’une information d’un nouveau type : la *position* de l’élément.

Nous dirons que :

- le multiensemble M est une *représentation* de l’ensemble E
- la séquence S est une *représentation* du multiensemble M
- la séquence S est une *représentation* de l’ensemble E

Exemple :
Ainsi, l’ensemble {10,20} est représenté par le multiensemble {(20,3),(10,3)} , lui-même représenté par la séquence (20,10,20,20,10) ainsi que par la séquence (20,20,10,20,10). Le dessin ci-après multiplie les exemples de représentation :



2.2) La structure mathématique ensemble et son type abstrait ENSEMBLE

A quelque détail près, l’objet manipulé sans cesse en informatique est un ensemble.

Rappel (qui servira de définition) :
Un ensemble Ω se caractérise par les éléments qu'il possède. Cette appartenance est notée $e \in \Omega$. Les symboles “{“, “,” et “}” sont utilisés pour désigner in extenso un ensemble.

Propriété :
Les 3 objets suivants sont des ensembles de cardinalité 2 et sont tous égaux :
{1,2,1,2,1,1} , {1,2} , {2,1}

Ici nous définissons le type abstrait (ensemble de primitives algorithmiques) `ENSEMBLE` :

Définition
`ENSEMBLE` est l'ensemble des primitives :

```
ensembleVide      :                          → ensemble
estVide           :     ensemble             → booléen
appartient        :     élément * ensemble   → booléen
ajouter           :     élément * ensemble   → ensemble
supprimer         :     élément * ensemble   → ensemble
choisir           :     ensemble             → élément
libérer           :     ensemble             →
```

Dans la définition ci-dessus, nous avons fourni la signature (type des entrées et des sorties) sans fournir la définition. Certaines sont évidentes. Par exemple `appartient` de signature `élément*ensemble→booléen` résout le problème :

```
APPARTIENT
      Entrée : un élément e, un ensemble E
      Sortie : le booléen (e ∈ E)
```

Ce n'est pas le cas de `ajouter` de signature `élément*ensemble→ensemble`. Deux définitions différentes sont possibles selon que l'on autorise ou non d'ajouter un élément déjà présent dans l'ensemble. L'alternative est donc formellement :

```
AJOUTER1
      Entrée : élément e, un ensemble E
      Sortie : l'ensemble {e} ∪ E

AJOUTER2
      Entrée : élément e, un ensemble E tels que e ∉ E
      Sortie : l'ensemble {e} ∪ E
```

Le choix entre `AJOUTER1` et `AJOUTER2` résulte d'une négociation entre les deux informaticiens qui cherchent chacun la simplicité :

- l'informaticien qui **utilise** `ajouter` peut préférer `AJOUTER1` car plus puissante.
- l'informaticien qui **implémente** `ajouter` préfère `AJOUTER2` car plus restreinte.

S'agissant d'ensemble et non de multiensemble, pour des raisons de simplicité logique nous choisirons la seconde.

2.3) La structure mathématique multiensemble et son type abstrait

Définition

Soit Ω un ensemble. Un *multiensemble* défini sur Ω est une fonction η : $\Omega \to \mathbb{N}$.

L'entier η(e) d'un élément e est appelé la *fréquence* de e dans ME.

Exemple

Le multiensemble {(10,2),(30,4)} contient les deux éléments 10 et 30 selon des fréquences respectives 2 et 4.

Définition

MULTIENSEMBLE est l'ensemble des primitives :

```
multiEnsembleVide  :                                  → multiensemble
estVide            : multiensemble                    → booléen
fréquence          : élément * multiensemble          → entier
nouvelleFréquence  : élément * entier * multiensemble → multiensemble
retirer            : élément * multiensemble          → multiensemble
choisir            : multiensemble                    → élément
libérer            : multiensemble                    →
```

2.4) La structure mathématique séquence et 5 de ses types abstraits

Notation : Le terme [1,n] où n est un entier désigne l'intervalle des entiers compris entre 1 et n (inclus). [1,0] désigne l'ensemble vide.

Il existe deux façons parfaitement équivalentes de définir une séquence : l'une est itérative, l'autre récursive. Des deux définitions, la première semble la plus naturelle. Toutefois, de la seconde définition (récursive) sera issue l'implémentation la plus efficace (liste chainée).

Définition itérative

Soit Ω un ensemble. Une *séquence* sur Ω est une fonction S : [1,n] → Ω où n est un entier appelée la *longueur* de S. L'élément S(i) est appelé *l'élément de S en position i.*

Définition récursive

Soit Ω un ensemble. Une *séquence* sur Ω est soit la séquence vide notée () soit un couple <a,T> où a est un élément de Ω et T est une séquence sur Ω. L'élément a est appelée la *tête* de <a,T> ; la séquence T la *queue* de <a,T>.

Exemple :

La séquence (20,10,30,40, 50) peut se définir comme :

- (définition itérative) comme la fonction S : [1,5] → ℕ définie par :

  S(1)=20, S(2)=10, S(3)=30, S(4)=40, S(5)=50

- (définition récursive) comme le couple <20,S2>
  avec S2=<10,S3>, S3=<30,S4>, S4=<40,S5>, S5=<50,S6>, S6=().

5 types abstraits très connus permettent de manipuler les séquences :

1. le *tableau*
2. la *structure*
3. la *pile*
4. la *file*
5. la *liste*

Les deux premiers types `tableau` et `structure` sont fournis par la `Machine` . Ce sont les seuls fournis : tous les autres types seront définis à partir de `tableau` et `structure`.

2.4.1) premier type abstrait “séquence” : le tableau

Le *tableau* est le type abstrait le plus naturel pour manipuler une séquence d’éléments de même type. Revers de la médaille : il fournit une implémentation aux capacités limitées. En effet, la seule modification possible est le remplacement d’un élément par un autre : on ne peut ni supprimer, ni ajouter un élément.
Le type abstrait `Tableau` est l’ensemble des primitives :

```
allouerTableau : entier * élément           → tableau
ième           : entier * tableau           → élément
remplacerIème  : entier * tableau * élément → tableau
longueur       : tableau                    → entier
libérer        : tableau                    →
```

A la syntaxe `remplacerIème(i,T,a)` on utilise et on préfère une syntaxe équivalente mais plus compacte `T[i]←a`. De même, `A←T[i]` est préféré à `A←ième(i,T)`.

Rappelons ici que la pauvreté expressive de tableau (le seule modification est le remplacement) permet une implémentation très efficace : toutes les primitives (autre que l’allocation) sont de complexité en temps constante.

2.4.2) deuxième type abstrait “séquence” : la structure

La structure est l’unique type abstrait proposé ici permettant de manipuler des éléments de types différents. Pour faciliter l’écriture algorithmique, les indices ne sont pas des entiers mais des identifiants (des mots choisis librement). De façon identique au tableau, on ne modifie une structure que par remplacement.

La signature de l’allocation de structure est :

```
allouerStructure  : mot1 * élément1 * … * motn * élément    → structure
```

De façon assez analogue aux tableaux, on lit et on modifie un champ d’une structure selon une syntaxe très compacte comme le présente l’exemple suivant.

Exemple

Un monôme de la forme $x \mapsto \alpha.x^k$ (fonction à valeurs réelles utilisant une puissance entière) se représente à l’aide d’une simple structure contenant un champ `coeff` de type `réel` et un champ `degré` de type `entier`. Ainsi, le monôme $3.x^{100}$ est représenté par la structure :

```
coeff :     3.0
degré :     100
```

La fonction dérivée, associant par exemple à $3.x^{100}$ le monôme $300.x^{99}$ s’écrit ainsi :

```
fonction derivéeMonome(M : monome)
debut
      M.coeff ← M.coeff * M.degré ;
      si M.degré <> 0 alors
```

```
                M.degré ← M.degré - 1
        fin
fin_exemple
```

2.4.3) troisième type abstrait “séquence” : la pile

La pile est le type abstrait séquence le plus frugal : seule une de ses deux extrémités est concernée par les ajouts et les suppressions. Cette extrémité est appelée la *tête*.

Définition

PILE est l’ensemble des primitives :

```
pileVide        :                       → pile
estVide         : pile                  → booléen
empiler         : élément * pile        → pile
dépiler         : pile                  → pile
tête            : pile                  → élément
libérerPile     : pile                  →
```

convention d’écriture :

la tête de la pile est le premier élément de la pile. Il en découle l’exemple suivant :

```
tête
Entrée:(10,20,30)
Sortie:  10
```

2.4.4) quatrième type abstrait “séquence” : la file

La file est à peine plus riche qu’une pile : ajout et suppression se font aux deux extrémités opposées de la séquence. L’ajout concerne l’extrémité terminale ; la suppression l’extrémité initiale.

Définition

FILE est l’ensemble des primitives :

```
fileVide        :                       → file
estVide         : file                  → booléen
enfiler         : élément * file        → file
défiler         : file                  → file
tête            : file                  → élément
libérerFile     : file                  →
```

2.4.5) cinquième type abstrait “séquence” : la liste

Le type abstrait le plus puissant est le type liste qui permet de modifier par ajout ou suppression la séquence à n’importe laquelle de ses positions.

Définition

LISTE est l’ensemble des primitives :

```
listeVide       :                       → liste
longueur        : liste                 → entier
ièmeElement     : entier * liste        → element
```

```
insérer         : element*entier*liste    → liste
supprimer       : entier*liste            → liste
libérerListe    : liste                   →
```

Par son pouvoir expressif, le type abstrait liste “contient” les 3 autres types abstraits que sont tableau, pile et file. Cette puissance expressive aura un coût : comme nous le verrons dans le chapitre suivant, les primitives de liste sont plus complexes de 2 points de vue :

- l’écriture algorithmique est plus complexe.
- la complexité en temps est plus élevée.

Conclusion

Nous avons vu ici qu’un même ensemble peut être représenté par des structures mathématiques différentes. Nous avons vu qu’une même structure mathématique peut être manipulée à l’aide de types abstraits. De même, nous verrons qu’un même type abstrait peut être implémenté de différentes façons.

Cette explosion de possibles peut donner le tourni ! En fait, elle fournit autant d’opportunités pour représenter et structurer les données manipulées par un algorithme de façon à obtenir une complexité en temps ou en espace minimale.

chapitre 10
Implémentation de types abstraits

Le chapitre précédent fournit plusieurs types abstraits permettant de manipuler des ensembles, des multi ensembles ou des séquences. Nous étudions ici comment définir les algorithmes efficaces qui utilisent les deux seuls types abstraits fournis par le modèle de calcul (la “machine”) de calcul que sont la `structure` et le `tableau`.

1) Implémentations fournies des types structure et tableau

Les 2 types abstraits `structure` et `tableau` sont fournis par le modèle de calcul. Rappelons ici que les primitives lecture et modification par substitution sont de complexité constante. La raison est que les éléments des séquences apparaissent en mémoire dans l’ordre où ils apparaissent dans la séquence.

2) Premières implémentations de ensemble et multiensemble

2.1 fonction caractéristique

Une représentation très naturelle d’un ensemble d’entiers E utilise tableau de booléens. Cela peut se faire à l’aide d’une structure contenant deux champs l’un de nom `cardinalité`, de type entier, et l’autre de nom `f_caractéristique`, de type tableau de booléens.

Exemple :
L’ensemble **{9,2,3,8}** est représenté par la structure :

`cardinalité` : 4

`f_caractéristique`:

1	2	3	4	5	6	7	8	9	10
0	1	1	0	0	0	0	1	1	0

Cette implémentation possède :

- une énorme qualité : la complexité en temps de `appartient` est constante.
- un énorme défaut : la complexité en espace est exponentielle .

En effet, la représentation du singleton {1 000 000 000 000 000} qui contient un entier codé sur 8 octets nécessite un tableau de longueur 1 000 000 000 000 000.

Pour cette raison, cette implémentation est à proscrire pour des ensembles d’entiers quelconques mais est fortement recommandée pour des ensembles de petits entiers; par exemple un ensemble d’indices de tableau.

2.2 fonction de hachage

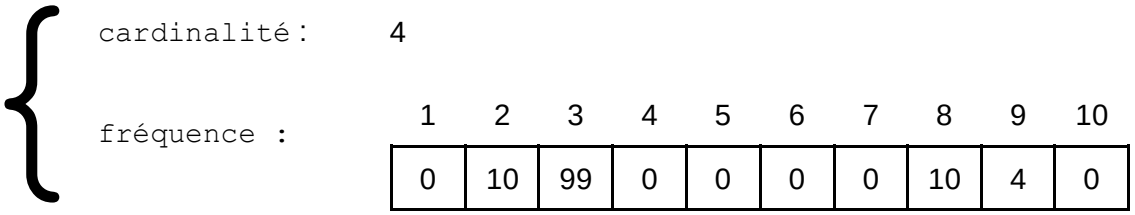
Une méthode permettant de contourner le défaut “exponentiel” précédent est d’utiliser une *fonction de hachage*, à savoir une fonction h qui associe à tout entier N un entier dans un intervalle fixé I=[1,n].

L’idée générale est de représenter l’ensemble E par la fonction de hachage h et la fonction caractéristique de l’ensemble haché h(E). A ceci près, qu’il faut gérer le risque de *collisions*, c’est à dire le fait que deux entiers N et M peuvent être associés à un même haché h(N)=h(M). Un exercice est dédié aux fonctions de hachage.

2.3 multiensemble

Les techniques vues plus haut s’étendent naturellement aux multi ensembles comme l’illustre l’exemple suivant :

Exemple :
Le multi-ensemble {(9,4),(2,10)(3,99),(8,10)} est représenté par la structure :



2.4 autres implémentations

Du simple fait qu’un ensemble ou un multiensemble peut être représenté par une séquence ou un arbre, toutes les implémentations des séquences et des arbres que nous verrons ensuite fournissent autant d’implémentations pour les ensembles et les multi ensembles.

3) Première implémentation de la séquence

Une façon naturelle de représenter une séquence S est de placer les éléments à des adresses mémoire qui respectent l’ordre des positions de ces éléments dans S. Dit plus simplement, de placer l’élément en position i dans la séquence à l’indice i du tableau.

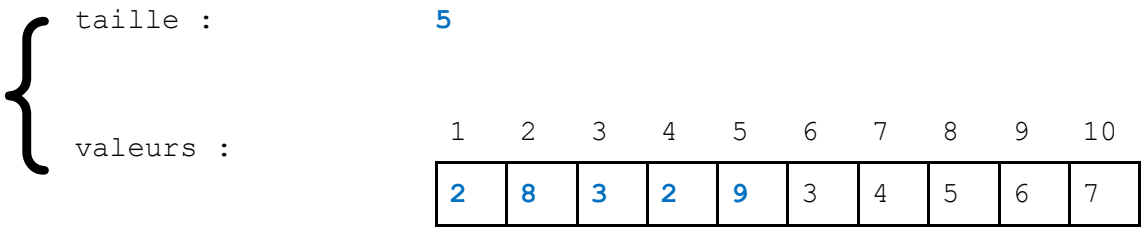
Ce principe est très efficace pour les types abstraits `pile` et la `file`.

3.1 Implémentation de la pile

Une façon simple de représenter une pile est d'utiliser une structure contenant un tableau de taille supérieure à celle de la pile, comme le montre l'exemple suivant.

Rappel : Rappelons ici que la tête de la pile est le premier élément de la séquence.

Exemple : La pile **(9,2,3,8,2)** (d'élément de tête l'entier 9) est représentée par la structure :



Les deux idées (non naturelles) d'une part de placer la fin de la pile au début du tableau et d'autre part d'utiliser un tableau de taille éventuellement supérieure à celle de la pile permettent d'obtenir des complexités en temps constante pour chacune des primitives du type abstrait `pile`.

```
fonction preparer(p : pile)
debut
    si taille(p) = longueur(p.valeurs) alors
    debut
        n ← longueur(p.valeurs) ;
        nouveauTableau ← allouerTableau(augmenter(n),p.valeurs[1]);

        pour i de 1 à n faire
            nouveauTableau[i] ← p.valeurs[i]
        ;
        libérer(p.valeurs) ;
        p.valeurs  ← nouveauTableau ;
    fin
fin

fonction augmenter(n:entier) : entier
debut
      retourner max(1,2*n)
fin
```

Complexité de `empiler` :

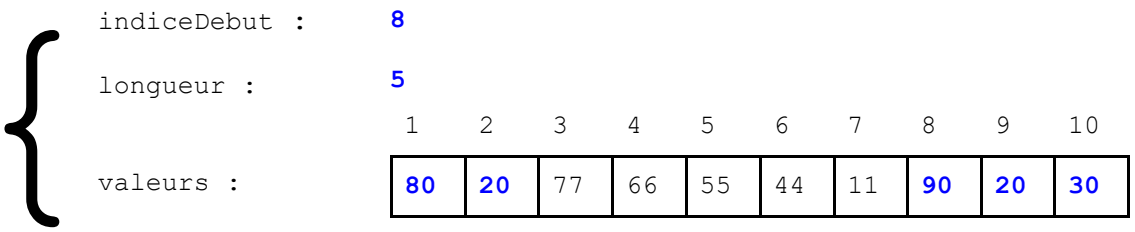
La primitive `empiler` est de complexité en temps (et en espace) non constante, elle est stricto sensu linéaire. Cependant selon d'un point de vue dit *amorti* , nous pouvons considérer que les complexités temps et espace de `empiler` sont constantes si nous doublons la taille du tableau chaque fois qu'il est nécessaire de l'augmenter.
Ce point fera l'objet d'un exercice.

```
fonction empiler(e : élément, p : pile)
debut
      preparer(p) ;
      p.taille ← p.taille + 1 ;
      p.valeurs[p.taille] ← e
fin
```

3.2 Implémentation de la file

La file étant à peine deux fois plus compliquée qu'une pile (la file possède deux positions actives (ses deux extrémités) ; la pile une seule : l'extrémité initiale), il est facile de représenter une file à l'aide d'un tableau et d'obtenir des primitives de complexité temps constante (avec une réserve concernant `enfiler` identique que celle concernant `empiler`).

Exemple : La file (90,20,30,80,20) est représentée par la structure :



La définition algorithmique des primitives fera l'objet d'un exercice.

4) Implémentation de la liste par chaînage de nœuds

Le saut qualitatif réalisée par la liste est de pouvoir insérer (ou supprimer) un élément à n'importe quelle position de la séquence. Cela la distingue du tableau où aucune insertion n'est possible et de la pile ou de la file où l'insertion est à la marge (à une extrémité). Pour obtenir des primitives efficaces, il faut rendre avec le principe du tableau où les ordres d'apparition des éléments dans la séquence et en mémoire se confondent.

4.1 une première idée : le nœud

```
nœud est le type abstrait :
      allouerNoeud      :    élement                  → noeud
      admetSuivant      :    noeud                    → booléen
      noeudSuivant      :    noeud                    → noeud
      contenu           :    noeud                    → élément
      changerContenu    :    nœud * élément           → noeud
      changerSuivant    :    nœud * noeud             → noeud
      libérerNoeud      :    noeud                    →
```

L'idée est de placer les éléments à des adresses quelconques et d'indiquer pour chaque élément l'adresse de l'élément suivant. Pour simplifier l'écriture algorithmique, nous introduisons un nouvel objet : le nœud qui contient l'élément et l'adresse de son successeur. Il prend la forme du type abstrait défini ici.

Définition (incomplète) :

En créant un type abstrait intermédiaire entre la liste et l'élément, nous pouvons obtenir des algorithmes plus simples. Première innovation est la nouvelle primitive à ajouter dans le type `liste` de signature : `iemeNoeud : entier * liste → noeud`
L'insertion d'un nouvel élément en position i dans une liste l s'écrit simplement :

```
fonction insérer(e:élément , i: entier , L : liste)
début
      noeud_nouveau ← allouerNoeud(e) ;
      noeud_prec ← iemeNoeud(i-1,L) ;
      noeud_succ ← noeudSuivant(noeud_prec) ;
      changerSuivant(noeud_prec,noeud_nouveau);
      changerSuivant(noeud_nouveau,noeud_succ)
fin
```

Certains pourront objecter, à juste raison !, que ce code est incorrect car dans le cas où i est égal à 1, un traitement particulier doit être effectué. Pour rendre ce code correct, nous allons enrichir le type nœud de la notion de sentinelle.

4.2 une deuxième idée : les nœuds sentinelles

Définition :
Un *nœud sentinelle* est un nœud qui précède (*sentinelle avant*) le nœud contenant le 1er élément de la séquence représentée ou succède (*sentinelle arrière*) au nœud contenant le dernier élément de cette séquence. Nous supposons dans ce cours que l'élément contenu dans une sentinelle n'a aucune importance.

La création et l'utilisation des sentinelles nécessitent d'enrichir le type abstrait nœud en ajoutant de nouvelles primitives.

Définition (complète) :

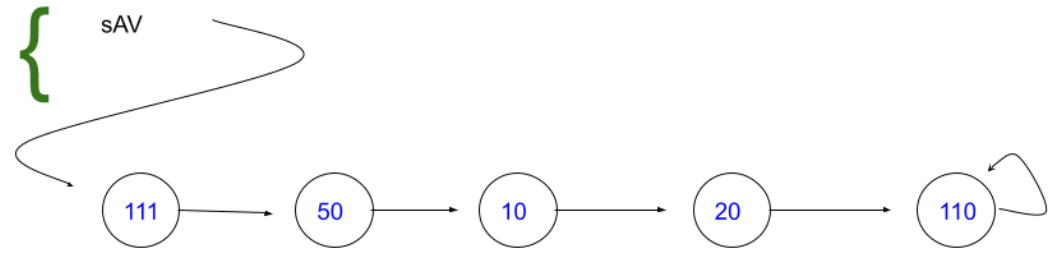
```
noeud est le type abstrait :
     allouerNoeud          : élément            → noeud
     admetSuivant          : noeud              → booléen
     noeudSuivant          : noeud              → noeud
     contenu               : noeud              → élément
     changerContenu        : nœud * élément     → noeud
     changerSuivant        : nœud * noeud       → noeud
     libérerNoeud          : noeud              →
     allouerSentinelle     :                    → noeud
     estSentinelle         : noeud              → booléen
```

Sous cette hypothèse :

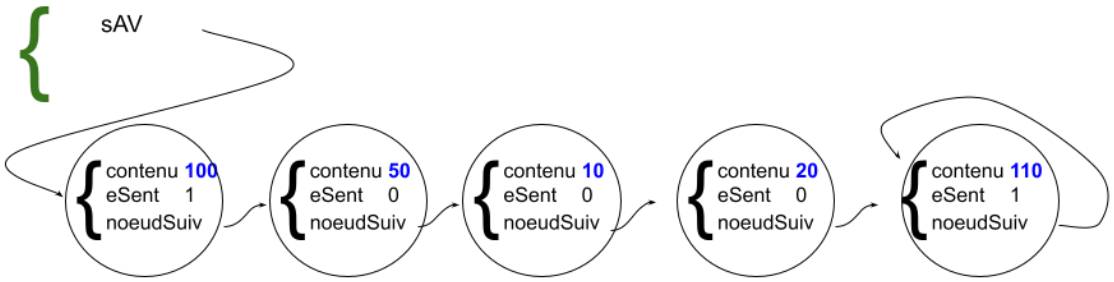
- une liste est représentée par une structure contenant comme champ principal le nœud sentinelle avant.
- Un nœud est représenté par une structure contenant au moins 3 champs : l'un de contenu l'élément, le second un booléen indiquant si c'est une sentinelle, le troisième contenant l'adresse du nœud suivant.

Exemple :

Le dessin ci-dessous fournit la représentation de la liste (50,10,20). Elle est représentée par une structure contenant l'adresse de la sentinelle avant. 5 nœuds sont chaînés dont 2 sentinelles. Ces 5 nœuds sont représentés par 5 structures dessinées en bas du dessin.



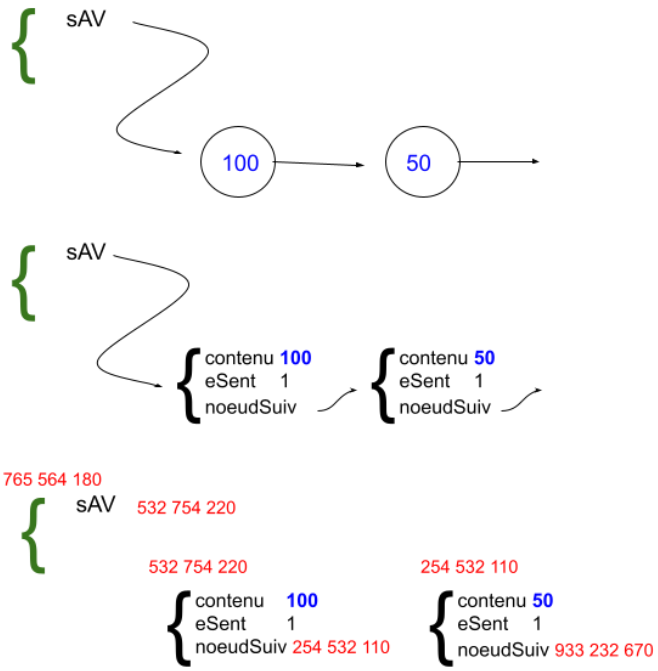
Dessin avec des noeuds



Dessin avec des noeuds et leurs structures

Signification de la flèche

Sur le dessin ci-dessous, figurent différents nœuds, la description de leurs structures ainsi que des valeurs supposées de leurs adresses mémoires (en rouge). Ainsi une flèche allant du contenu d'un champ A vers une structure B signifie que le contenu du champ A est l'adresse de la structure B.



...	254 532 110	254 532 114	254 532 115	...	532 754 220	532 754 224	532 754 225	...	...	765 564 180	...
...	50	1	933 232 670	...	100	1	254 532 110	...	...	532 754 220	...

4.3 une troisième idée :chaînage arrière

Si chaque nœud ne possède pour information que l'adresse du nœud suivant, calculer le nœud précédent d'un nœud n est de complexité en temps linéaire : il est nécessaire à partir de la sentinelle avant de chercher le nœud dont le nœud suivant est n.

Certains algorithmes requièrent de progresser tantôt de gauche à droite tantôt de droite à gauche. Pour cela, une solution permettant de réaliser ces déplacements en temps constant consiste simplement à ajouter dans tout nœud un nouveau champ contenant l'adresse du nœud précédent.

4.4 une quatrième idée : nœud curseur

Certains algorithmes nécessitent de mémoriser la position et le dernier nœud manipulé dans une liste. Cela se fait simplement en ajoutant deux nouveaux champs dans la structure liste :

- un champ `positionCurseur` de contenu la position p du nœud dans liste.
- un champ `noeudCurseur` de contenu l'adresse du nœud de position p.

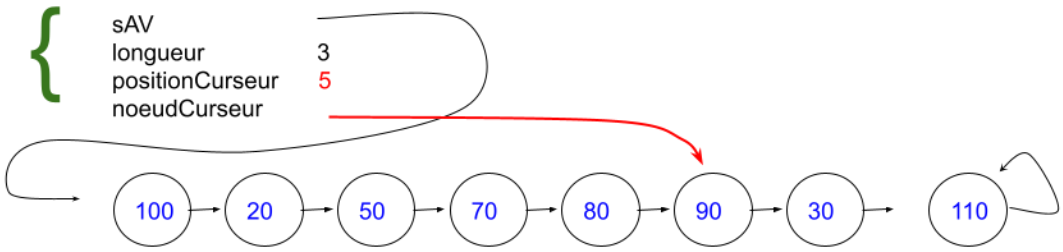
Ceci permet par exemple pour calculer le ième nœud d'une liste de ne pas repartir de la sentinelle avant mais si cela est plus rapide de partir du nœud curseur.

Ainsi l'algorithme parcours suivant :

```
fonction parcours(L:liste)
début
      n ← longueur(L) ;

      pour i de 1 à n faire
            a ← contenu(iemeNoeud(i,L))
end
```

est de complexité en temps $\Theta(n^2)$ si la liste est implémenté sans curseur mais est de complexité en temps linéaire $\Theta(n)$ si elle utilise un curseur.



Liste avec champs curseur et longueur

4.5 complexités en temps et en espace

Les primitives du type liste issues d'une implémentation par chaînage (qui seront écrites en TP) sont de complexité en espace constante mais de complexité en temps linéaire. L'obtention d'algorithmes efficaces nécessite d'écrire un algorithme parcourant un petit nombre de fois de gauche à droite les listes manipulées.

Chapitre 11
L'arbre binaire :
définitions, implémentations et parcours

Dans les chapitres précédents, nous avons étudié deux implémentations de la séquence : la première est le tableau très efficace pour accéder (en temps constant) à un élément selon sa position mais très inefficace pour insérer (en temps linéaire) un élément à une position ; la seconde est la liste chaînée qui possède les deux propriétés opposées (respectivement en temps linéaire et en temps constant).

Nous présentons ici un objet mathématique, l'*arbre binaire* que nous définirons de deux façons différentes mais équivalentes :

- 3 définitions récursives
- 1 définition itérative

Nous en déduirons 2 implémentations différentes :

- une implémentation à l'aide de noeuds, qualifiée "récursive"
- une implémentation à l'aide de tableaux, qualifiée "itérative"

Nous présenterons 4 façons classiques de parcourir les arbres :

- une première triviale car récursive "pure" (R-R-R)
- une seconde triviale car itérative "pure" (I-I-I)
- deux autres non triviales mais classiques qui sont les écritures itératives des parcours profondeur ou largeur d'un arbre chaîné (que nous codons R-I-R et I-I-R).

1) Définitions

1.1) Définitions récursives

De façon similaire à la séquence, il existe 4 définitions équivalentes de l'arbre binaire : une itérative, 3 récursives. Contrairement à la séquence, celles récursives sont plus naturelles que celle itérative.

Définition récursive (préfixe) d'un arbre binaire
Soit Ω un ensemble. Un *arbre binaire T sur Ω*, appelé ici simplement un *arbre*, est :

- soit l'*arbre vide*, noté ∅
- soit un triplet <a,U,V> composé d'un élément a de Ω et deux arbres binaires U et V ; a est appelé l'élément *racine* de <a,U,V> ; U est appelé le *sous-arbre gauche* de T ; V le *sous-arbre droit* de T.

Définition hauteur d'un arbre
La *hauteur* d'un arbre est -1 si l'arbre est vide ou sinon la somme de 1 et du maximum des hauteurs de ses sous-arbres gauche et droit.

Définitions récursive infixe et postfixe d'un arbre binaire.
Deux définitions alternatives sont possibles:

- une dite *infixe* en considérant le triplet <U,a,V>
- une dite *postfixe* en considérant le triplet <U,V,a>

Par extension, un arbre A est un *sous-arbre* d'un arbre T si il est égal à T ou si il est un sous-arbre du sous-arbre gauche de T ou du sous-arbre droit de T.

L'ensemble des *sommets* d'un arbre T est l'ensemble des racines des sous-arbres non vides de T. Un sommet n est *enfant gauche* (resp. *enfant droit*) d'un sommet p si il est racine du sous-arbre gauche (resp. droit) d'un arbre dont la racine est p ; le sommet p est dit le *parent* de n. Une *feuille* de T est un sommet qui possède ni enfant gauche ni enfant droit. Les sommets non feuilles sont dits *internes*.

1.2) Définition itérative

Définition préalable

Un *2-intervalle* est un ensemble d'entiers I tel que :

- soit : I est l'ensemble vide
- soit : I contient l'entier 1 et I contient tout entier de la forme x/2 avec x ∈ I \ 1.

Définition itérative d'un arbre binaire

Soit Ω un ensemble. Un *arbre binaire T* sur Ω est une fonction I → Ω où I est un 2-intervalle.

Les *sommets* de T sont les éléments de I. L'*enfant gauche* (resp. *droit*) d'un sommet p de T est l'entier 2*p (resp. 2*p+1) si il appartient à I.

Exemple :

Le dessin ci-dessous fournit le dessin d'un arbre T possédant 5 sommets de contenus {10,20,50,90,110}. La hauteur de l'arbre est 3. Le sommet 1, qui est la racine, a pour enfant gauche le sommet 2 et pour enfant droit le sommet 3. Le sommet 3 a pour enfant gauche le sommet 6 et ne possède pas d'enfant droit. Les deux sommets 2 et 13 sont les feuilles.
Le dessin après fournit le dessin et la définition récursive du sous-arbre droit de T.

Définition récursive préfixe

T= <50,<20,∅,∅>,<110,<90,∅,<10,∅,∅>>,∅>>

Définition récursive infixe

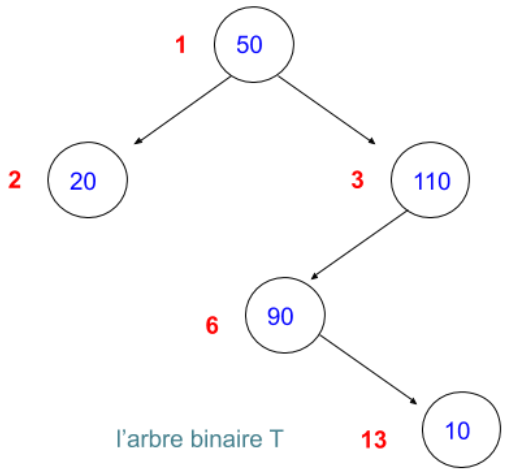
T= <<∅,20,∅>,50,<<∅,90,<∅,10,∅>>,110,∅>>

Définition récursive postfixe

T= <<∅,∅,20>,<<∅,<∅,∅,10>,90>,∅,110>,50>

Définition itérative

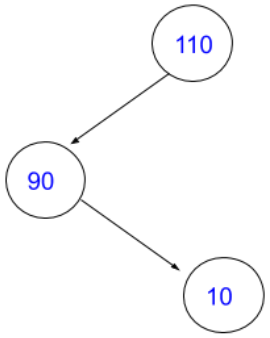
T={(1,50),(2,20),(3,110),(6,90),(13,10)}



Définition récursive (préfixe) de U

U= <110,<90,∅,<10,∅,∅>>,∅>

Le sous-arbre droit U de l'arbre binaire T



2) Les 4 séquences induites d'un arbre binaire

Contrairement à la séquence qui induit un unique ensemble, l'arbre binaire induit 4 séquences différentes : une pour chacune des 4 définitions plus haut.

2.1) Les 3 séquences profondeur

Les 3 premières définitions sont issues directement des définitions récursives d'un arbre binaire. Ainsi, pour les définir on choisit une définition récursive opérant sur un arbre défini récursivement :

Définition séquence préfixe

La *séquence préfixe* d'un arbre T, notée séquencePréfixe(T), est égale à :

- () si l'arbre T est vide
- la concaténation (a)•séquencePréfixe(U)•séquencePréfixe(V) avec T=<a,U,V>

Définition séquence infixe

La *séquence infixe* est définie de façon similaire en considérant l'expression :

séquenceInfixe(T) := séquenceInfixe(U)•(a)•séquenceInfixe(V)

Définition séquence postfixe

La *séquence infixe* est définie de façon similaire en considérant l'expression :

séquencePostfixe(T) := séquencePostfixe(U)•séquencePostfixe(V)•(a)

2.2) La séquence largeur

La 4ième définition est issue directement de la définition itérative d'un arbre binaire. Aussi, pour la définir on choisit une définition itérative opérant sur un arbre itérativement.

Définition séquence largeur

La *séquence largeur* d'un arbre T, notée séquenceLargeur(T), est égale à :

$(T(s_1),T(s_2),\ldots,T(s_n))$ où $T : I \to \Omega$ est l'arbre binaire défini itérativement et où $s_1<s_2<\ldots<s_n$ sont les n entiers du 2-intervalle I.

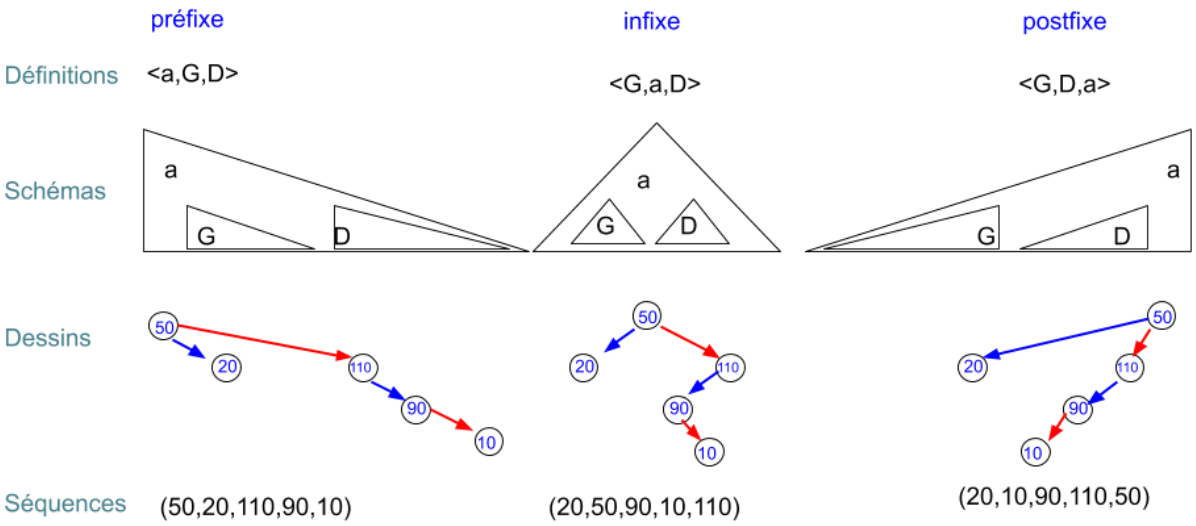
Exemple :

Voici sur l'exemple de l'arbre binaire T, les 4 séquences associées.

Définition récursive préfixe	Séquence préfixe
T= <50,<20,⌀,⌀>,<110,<90,⌀,<10,⌀,⌀>>,⌀>>	(50,20,110,90,10,)
Définition récursive infixe	**Séquence infixe**
T= <<⌀,20,⌀>,50,<<⌀,90,<⌀,10,⌀>>,110,⌀>>	(20,50,90,10,110)
Définition récursive postfixe	**Séquence postfixe**
T= <<⌀,⌀,20>,<<⌀,<⌀,⌀,10>,90>,⌀,110>,50>	(20,10,90,110,50)
Définition itérative	**Séquence largeur**
T={(1,50),(2,20),(3,110),(6,90),(13,10)}	T=(50,20,110,90,10)

2.3) Dessins préfixe, infixe et postfixe

Une façon de “visualiser” les séquences préfixes, infixes ou postfixe est de dessiner l’arbre binaire selon un dessin préfixe, infixe ou suffixe. Puis de projeter sur une ligne horizontale les contenus des sommets selon un axe vertical. Ceci est illustré ci-dessous :
Attention : dans les dessins préfixe (resp. postfixe), la flèche enfant gauche (resp. droite) va vers la droite (resp. gauche). Cela peut créer une confusion. Pour cela, nous utilisons une couleur bleue pour la gauche et rouge pour la droite.



3) implémentations récursive et itérative d’un arbre binaire

3.1) Implémentation chaînée dite “récursive”

Corollaire immédiat du caractère naturel de la définition récursive d’un arbre binaire, l’implémentation la plus simple et naturelle d’un arbre binaire est “récursive” c'est-à-dire en utilisant des nœuds et en les chaînant. Les techniques utilisées sont exactement celles utilisées pour implémenter les séquences à l’aide de listes chaînées.

La première idée est l’utilisation du type abstrait noeud.

La deuxième idée est l'utilisation de sentinelles. La sentinelle "avant" sera appelée la *sentinelle racine* ; les sentinelles "arrière" seront appelées *sentinelles feuilles*.

Définition du type abstrait noeud

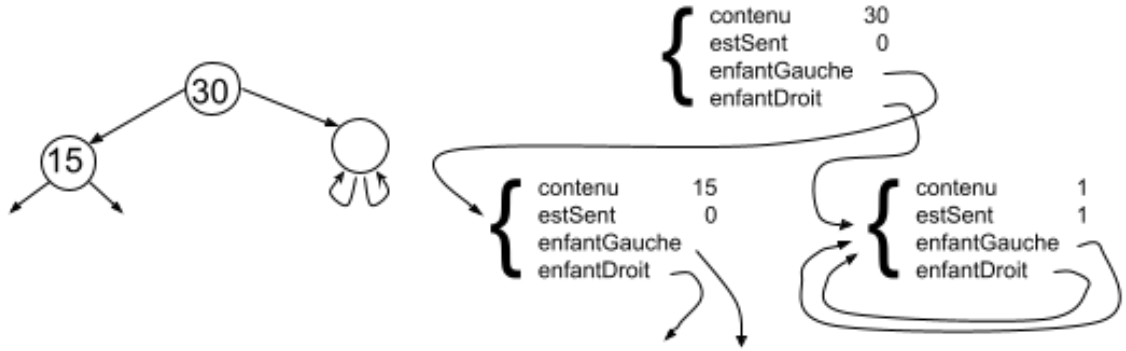
`noeud` est le type abstrait :

```
allouerNoeud          :   élément          → noeud
allouerSentinelle     :                    → noeud
libérerNoeud          :   noeud            →
estSentinelle         :   noeud            → booléen
enfantGauche          :   noeud            → noeud
enfantDroit           :   noeud            → noeud
contenu               :   noeud            → élément
changerContenu        :   noeud*élément    → noeud
changerEnfantGauche   :   noeud*noeud      → noeud
changerEnfantDroit    :   noeud*noeud      → noeud
```

Exemple une première représentation du nœud est une structure à 4 champs ;

```
contenu               :   élément
estSent               :   booléen
enfantGauche          :   noeud
enfantDroit           :   noeud
```

Voici le dessin de la représentation de 3 nœuds :



3 noeuds dont 1 sentinelle

les 3 représentations des 3 noeuds à l'aide de 3 structures

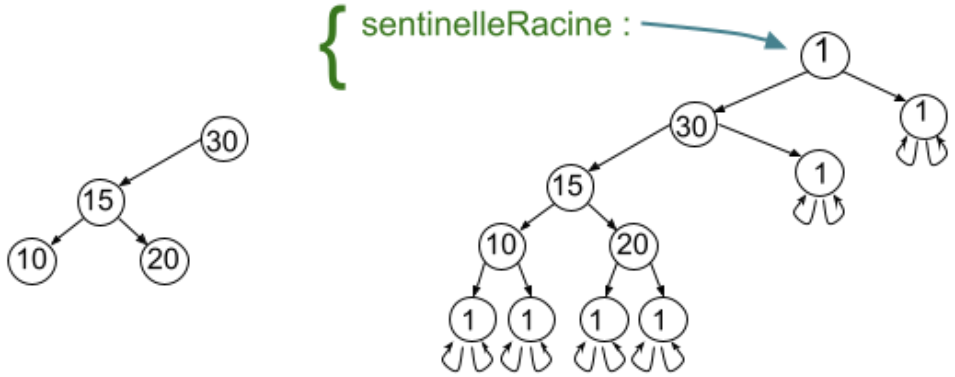
Pour les mêmes raisons que pour les listes chaînées, il peut être parfois très utile d'ajouter un nouveau chaînage pour chaque nœud : le nœud parent. Cela est réalisé au moyen d'un nouveau champ de nom `noeudParent` dans la structure représentant le nœud.
Dans ce cas, les primitives à ajouter sont :

```
noeudParent           :   noeud            → noeud
changerNoeudParent    :   noeud*noeud      → noeud
```

Exemple : ci-dessous est dessiné un arbre et sa représentation à l'aide de nœuds chainés. Les notions de sommet et de nœuds peuvent se confondre ; très souvent on emploie indifféremment ces mots. Les choix effectués concernant le chaînage des sentinelles sont :

- la sentinelle racine a pour enfant gauche le noeud racine et pour enfant droit une sentinelle
- les enfants d'un noeud sentinelle "feuille" sont lui-même

Les sentinelles ont pour particularité ici d'avoir pour contenu l'entier 1.



Un arbre binaire ayant 4 sommetS

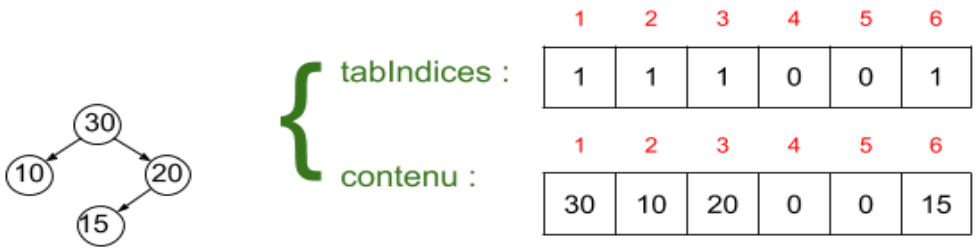
Sa représentation "machine" utilisant 11 noeuds dont 7 sentinelles et utilisant donc 12=1+11 structures

Cette représentation est linéaire et est donc optimale en espace. En effet, le nombre de nœuds n'excède pas 3 fois le nombre de sommets.

3.2) Implémentation itérative

Une définition issue directement de la définition itérative consiste d'une part à représenter le 2-intervalle I et la fonction $I \rightarrow \Omega$ à l'aide de deux tableaux comme l'indique l'exemple suivant.

Exemple :



Un arbre binaire ayant 4 sommets ; le 2-intervalle défini est {1,2,3,6}

Sa représentation "machine" utilisant 11 noeuds dont 7 sentinelles et utilisant donc 12=1+11 structures

Attention : cette représentation a un gros avantage : sa simplicité. Elle a un énorme défaut : sa complexité espace est **exponentielle**. Par exemple, l'arbre ayant **n** sommets définissant une unique branche d'enfants gauches induit un 2-intervalle de cardinalité n égal à $\{2^0,2^1,2^2,..,2^{n-1}\}$: sa représentation machine utilise donc deux tableaux de taille $\mathbf{2^{n-1}}$. Son utilisation n'est recommandée que pour des arbres binaires équilibrés.

4) 4 parcours récursif ou itératif d'arbres implémentés en récursif ou en itératif.

Un algorithme de parcours est un algorithme qui permet d'accéder à chaque élément d'un ensemble, c'est à dire ici à chacun des nœuds de l'arbre parcouru. Nous utilisons ici une fonction auxiliaire `traiter` qui réalise un traitement basique d'un noeud, qui pourrait par exemple être une simple lecture.

4.1) parcours Récursif-récursif-récursif (R-R-R) : très facile !

Définition parcours en profondeur

Tout objet défini récursivement a vocation à être parcouru récursivement. Voici la variante préfixe d'un parcours récursif, dit *parcours en profondeur*, (R) écrit en récursif (R) sur un arbre implémenté récursivement (R) :

```
fonction parcoursPréfixeArbre(T:arbre)
début
     si non(estVide(T)) alors
          parcoursPréfixeNoeud(racine(T))
fin

fonction parcoursPréfixeNoeud( n : noeud)
début
     si non(estSentinelle(n)) alors
     début
          traiter(n);
          parcoursPréfixeNoeud(enfantGauche(n)) ;
          parcoursPréfixeNoeud(enfantDroit(n))
     fin
fin
```

Les variantes infixe et postfixe s'obtiennent respectivement en déplaçant l'instruction `traiter(n)` entre ou après les instructions `parcoursPréfixeNoeud(enfantGauche(n))` et `parcoursPréfixeNoeud(enfantDroit(n))`.

4.2) parcours Itératif-Itératif-Itératif (I-I-I) : très facile !

Tout objet défini itérativement a vocation à être parcouru itérativement. Voici le parcours itératif (I), dit parcours en largeur, écrit en itératif (I) d'un arbre implémenté itérativement (I) :

```
fonction parcoursLargeur(T:arbre)
début
     n ← longueur(T.tabIndices) ;

     pour i de 1 à n faire
          si T.tabIndices[i] alors
              traiter(T.contenu[i])
fin
```

4.3) Itératif-itératif-récursif (I-I-R)
c-a-d parcours en largeur (I) écrit en itératif (I) d'un arbre chaîné (R) : un classique !

Il peut être utile de parcourir itérativement un arbre implémenté récursivement (c'est-à-dire à l'aide de nœuds chaînés).

Définition parcours en largeur

Ce parcours a pour nom *le parcours en largeur.* Son écriture est uniquement itérative et utilise une file. La voici :

```
Fonction parcoursLargeur(T : arbre binaire)
début
     FILE ← fileVide();
     si non(estVide(T))) alors
          enfiler(racine(T),FILE);

     tantque non(estVide(FILE)) alors
     début
          n ← Tete(FILE) ;
          si non(estSentinelle(n)) alors
          debut
               traiter(n);
               enfiler(enfantGauche(n),FILE) ;
               enfiler(enfantDroit(n),FILE)
          fin ;
     defiler(FILE)
     end
fin
```

4.4) Itératif-récursif-récursif (R-I-R)
c-a-d parcours en profondeur (R) écrit en itératif (I) d'un arbre chainé (R) : un classique!

Il sera traité en exercice.

4.5)

Les 4 autres algorithmes (**R-I-I, I-R-R, I-R-I, R-R-I**) présentent moins d'intérêt.

Chapitre 12
Quelques exemples d'arbres binaires

Nous présentons ici quelques exemples célèbres des arbres binaires :

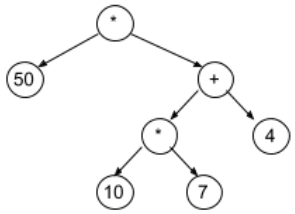
1. l'arbre binaire syntaxique
2. l'arbre binaire de recherche (A.B.R.)
3. le tas

1 L'arbre binaire syntaxique

Les arbres syntaxiques permettent de représenter de façon structurée des textes dotés d'une syntaxe avec parenthésage. L'exemple le plus familier est celui des expressions arithmétiques utilisant les opérateurs binaires : +,-,*,/,^.
L'arbre syntaxique est familier car depuis l'âge de 10 ans, les enfants le manipulent quand ils évaluent une expression arithmétique. En fait, ils ne le manipulent pas directement mais manipulent son codage infixe : son expression infixe complètement parenthésée.

Exemple 1 Arbre syntaxique et expression infixe

Arbre syntaxique



Expression infixe
complètement parenthésée

(50*((10*7)+4))

Le dessin ci-dessous représente un arbre syntaxique et son codage selon une expression infixe complètement parenthésée.
fin

Définition arbre syntaxique
Un *arbre syntaxique* est un arbre binaire où :

- chaque sommet interne a pour contenu un opérateur.
- chaque sommet feuille a pour contenu un entier.

fin

Voici l'ensemble des définitions mathématiques. Certaines utilisent l'opération concaténation entre séquences notée • .

Définition Expression arithmétique infixe
L'expression *infixe* (complètement parenthésée) d'un arbre (mathématique) est définie par :
expInfixe(T) est égal à :

- contenu(n) si T possède un seul sommet noté n

- la concaténation (“(“) • expInfixe(G) • (a) • expInfixe(D) • (“)”) si T égale <G,a,D>

fin

Définition Expression arithmétique postfixe

L’expression *postfixe* d’un arbre mathématique est définie par :

expPostfixe(T) est égal à :

- contenu(n) si T possède un seul sommet noté n
- la concaténation expPostfixe(G) • expPostfixe(D) • (a) si T égale <G,a,D>

fin

Exemple 2 Arbres k-aires

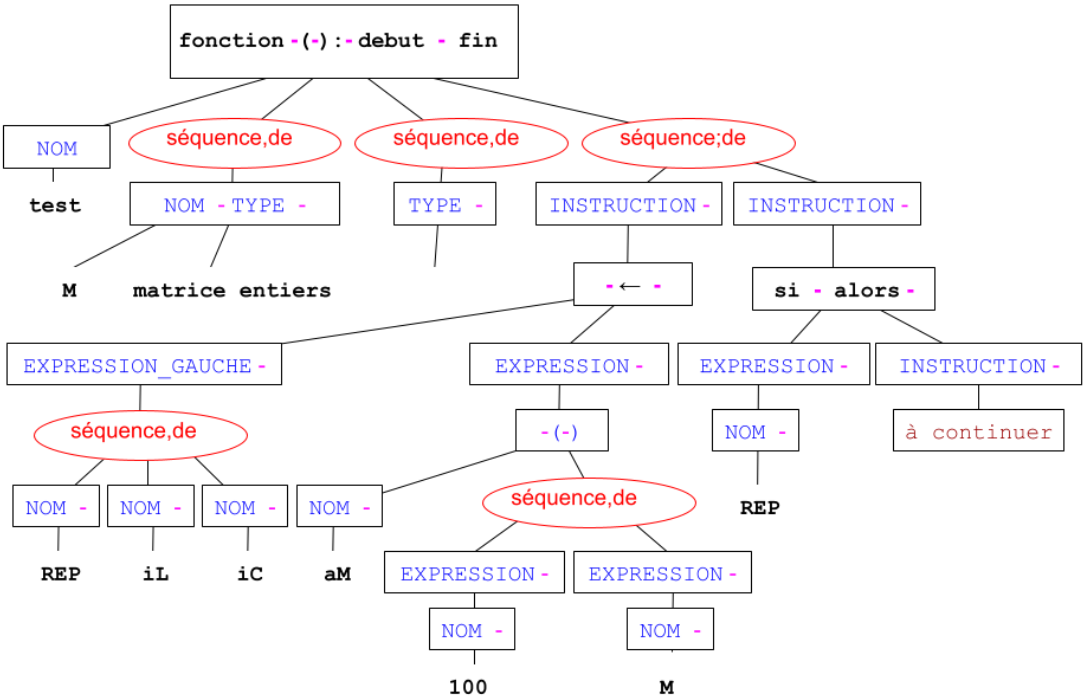
D’autres syntaxes donnent lieu à des représentations par arbres. Très souvent, ces syntaxes utilisent des opérateurs admettant un nombre d’opérandes autre que 2. Pour cela, nous utilisons des arbres k-aires : un sommet peut avoir un nombre quelconque d’enfants (autre que 2).

Un exemple d’arbres k-aire sont les arbres syntaxiques associés à nos algorithmes. Voici l’exemple présenté en conclusion du chapitre 2 :

un algorithme :

```
fonction test(M : matrice entier)
debut
  (REP, iL, iC) ← aM(100, M);
  si REP alors M[iL][iC] ← 0
fin
```

son arbre syntaxique :



2 L'arbre binaire de recherche (A.B.R)

Un second arbre binaire important est celui de recherche. Il permet, quand il est équilibré, de manipuler tout ensemble à l'aide de primitives de complexité en temps logarithmiques.

2.1 Définition de A.B.R et de `appartient`

Définition A.B.R

Un arbre binaire défini sur $\mathbb{N}$ est dit "*de recherche*" si sa séquence infixe est croissante.

Définition récursive de l'appartenance

La définition (mathématique) de l'appartenance d'un élément à un A.B.R T est très simple et est récursive.

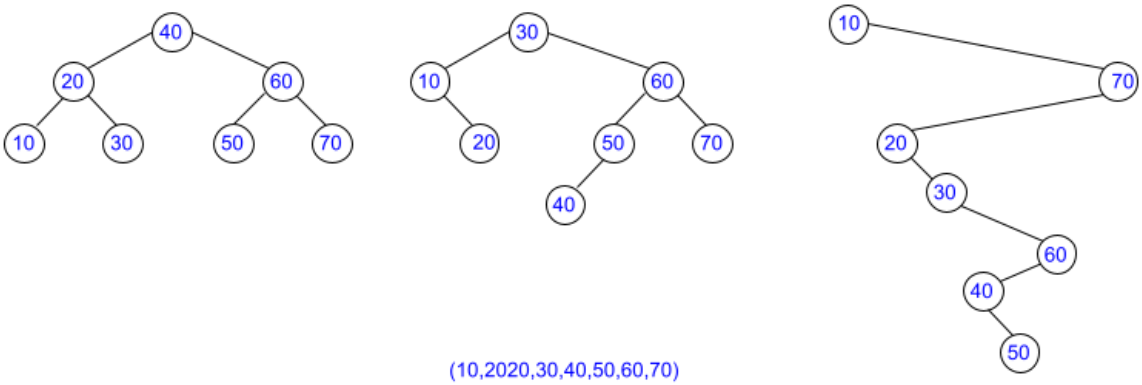
$\in$(x,T) est égal à :

$$\begin{cases} \text{faux}() & \text{si } T = \varnothing \\ \text{vrai}() & \text{si } x=a \text{ et } T = <a,\text{GAUCHE},\text{DROIT}> \\ \in(x,\text{GAUCHE}) & \text{si } x<a \text{ et } T = <a,\text{GAUCHE},\text{DROIT}> \\ \in(x,\text{DROIT}) & \text{si } x>a \text{ et } T = <a,\text{GAUCHE},\text{DROIT}> \end{cases}$$

Comme nous le verrons en TP, cette définition fournit un algorithme de complexité en temps la hauteur de l'arbre. Elle fournit en outre la structure algorithmique des algorithmes ajout et suppression : par exemple, ajouter un élément dans un ABR est réalisé en plaçant l'élément à la position où n'a pas été trouvé cet élément. Ces deux algorithmes sont aussi de complexité en temps la hauteur de l'arbre.

Exemple

Voici 3 ABR représentant le même ensemble {10,50,55,60,65,70,80} et donc ayant même séquence infixe triée (10,20,30,40,50,60,70) :



(10,2020,30,40,50,60,70)

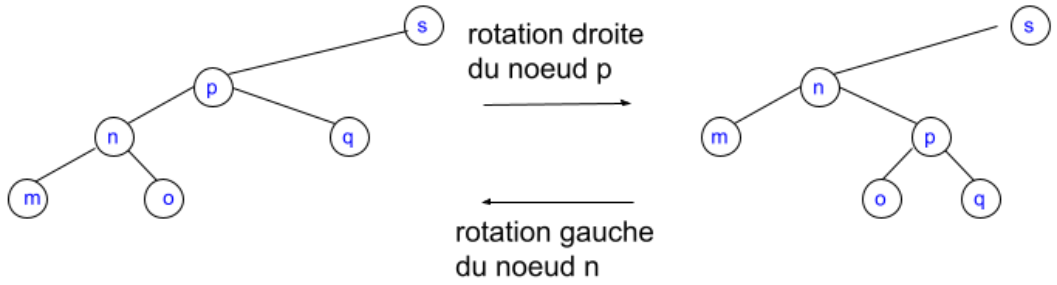
2.2 la rotation : opération emblématique sur les ABR

Nous présentons ici une opération qui transforme un ABR en un nouvel ABR.

Définitions
La *rotation droite* de l'arbre binaire <<A,n,B>,p,C> est l'arbre binaire <A,n,<B,p,C>>.
La *rotation gauche* de l'arbre binaire <A,n,<B,p,C>> est l'arbre binaire <<A,n,B>,p,C>.

Propriété
La rotation transforme tout ABR en un nouvel ABR.
fin
preuve
Notons SI(T) la séquence infixe de tout arbre binaire T ; conséquence de la définition précédente, un arbre binaire <<A,n,B>,p,C> et sa rotation droite <A,n,<B,p,C>> partagent la même séquence infixe SI(A).(n).SI(B).(p).SI(C). Ainsi les deux arbres binaires sont soit 2 ABR soit ne sont pas des ABR.
fin

Cette notion de rotation s'étend à tout sommet s d'un noeud en "réalisant" la rotation de l'arbre dont il est racine. Ceci est illustré par le dessin suivant :



Voici une définition algorithmique de la rotation gauche d'un noeud :

```
fonction rotationGauche(n : noeud)
debut
      s ← parent(n) ;
      p ← enfantDroit(n) ;
      o ← enfantGauche(p) ;

      si estEnfantGauche(n) alors
            changerEnfantGauche(s,p)
      sinon
            changerEnfantDroit(s,p)
      ;
      changerEnfantDroit(n,o) ;
      changerEnfantGauche(p,n)
fin
```

2.3 Une complexité logarithmique nécessite l'équilibrage

L'annonce de ce chapitre et de cette section est que ces complexités en temps sont logarithmiques (en le nombre n du nombre d'éléments de l'A.B.R.). Il en découle les définitions suivantes :

Définition A.B.R équilibré

Un ensemble **T** d'A.B.R est *équilibré* si il existe une constante K tel que tout arbre T de **T** vérifie : hauteur(T) < K.ln(taille(T))
Ceci peut être noté : hauteur(T)=Θ(ln(taille(T))

Comme nous le verrons en TP, la difficulté réside alors dans la rapidité algorithmique de `Equilibrer`, c'est à dire l'algorithme qui équilibre un arbre provenant d'un A.B.R équilibré qui vient d'être faiblement déséquilibré par l'ajout (ou la suppression) d'un élément.

Il existe plusieurs définitions différentes d'arbres équilibrés.

Exemple 1 arbres parfaits

La définition la plus simple d'arbre équilibré est d'imposer que tout sommet interne ait deux enfants et que toutes les feuilles soient à même distance de la racine. Il vient facilement :

hauteur(T) ≤ ln(taille(T))

Cette contrainte est trop forte car seuls les ensembles de cardinalité de la forme 2^h-1 peuvent être représentés.
fin_exemple

Exemple 2 : arbres presque complets

La définition la plus simple d'arbre équilibré de taille quelconque est d'imposer que tout sommet à distance de la racine strictement inférieure à hauteur(T)**-1** ait exactement 2 enfants. Il est facile de vérifier alors:

hauteur(T) ≤ ln(taille(T)) + 1

Cette contrainte est malheureusement trop forte : il **n'existe** pas d'algorithmes `Equilibrer` en temps logarithmique.
fin_exemple

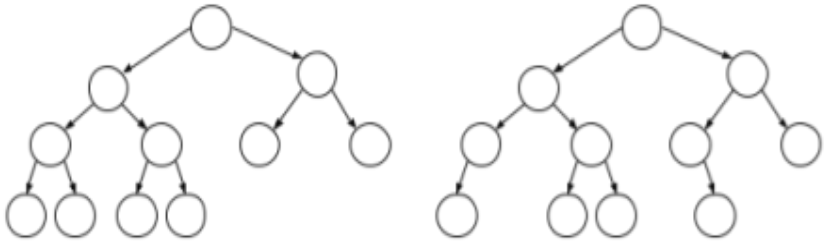
Exemple 3 arbres complets

Une autre définition d'équilibrage est d'imposer que le 2-intervalle formé par ses positions (voir définition itérative de l'A.B) soit un intervalle d'entiers. En plus clair : tous les niveaux sont remplis, sauf le dernier qui est entièrement rempli sur sa gauche.
Pour des raisons similaires à celles des arbres presque complets :

- tout arbre complet T vérifie : hauteur(T) ≤ ln(taille(T)) + 1
- **il n'existe pas** d'algorithme `Equilibrer` en temps Θ(ln(taille(T)))

fin_exemple

Exemple 4 Le dessin ci-dessous présente à gauche un arbre binaire complet et à droite presque complet. Leurs tailles sont identiques et égales à 11 ; leurs hauteurs égales à 3.



Arbre binaire complet Arbre binaire presque complet

Après ces 3 tentatives infructueuses, rassurons nos lecteurs. Plusieurs familles d'arbres équilibrés existent comportant une solution algorithmique `Equilibrer` en temps logarithmique. Il s'agit par exemple, des arbres AVL, des arbres rouge noir ou aussi des 2-3 arbres. Ce dernier exemple sera traité en TP.

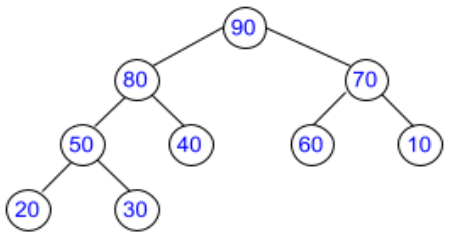
3 Un arbre binaire dédié au TRI : le TAS

Un des rares algorithmes simples permettant de trier un tableau de n entiers selon un complexité espace Θ(1) et en temps Θ(n.ln(n)) est le tri par tas. Cet algorithme utilise un type abstrait appelé le TAS qui se caractérise par le fait suivant :
`extraireMax Ensemble -> Element * Ensemble` est de complexité temps Θ(ln(n))

L'obtention de Θ(ln(n)) est obtenu en implémentant le type abstrait TAS à l'aide d'un arbre binaire appelé (aussi !) un Tas, c.a.d vérifiant deux propriétés :

1. tout contenu d'un parent est strictement supérieur aux contenus de ses enfants
2. l'arbre est complet.

Exemple 4 voici un exemple de tas représentant l'ensemble {10,20,30,40,50,60,70,80,90}. L'arbre binaire étant contraint à être complet, sa représentation est faite à l'aide d'un tableau et d'un entier indiquant la taille du tas.



taille : 9

valeurs :

1	2	3	4	5	6	7	9	9	10
90	80	70	50	40	60	10	20	30	77

Chapitre 13 :
les 2,3 arbres

Les 2,3 arbres ne sont pas exactement des arbres binaires de recherche mais en sont très proches. Ils fournissent une classe d'arbres très faciles à équilibre.

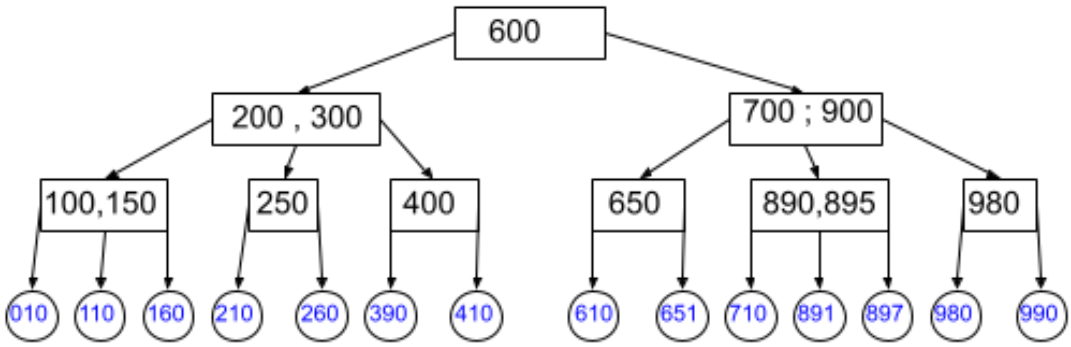
Trois idées caractérisent les 2-3 arbres :

1. les valeurs de l'ensemble à représenter sont placées uniquement sur les feuilles de l'arbre.
2. toutes les feuilles sont placées à même distance de la racine.
3. chaque sommet interne possède 2 ou 3 enfants.

Les sommets internes possèdent des balises de façon à ce que la séquence infixe composée des balises et des feuilles soit triée.

1) Définitions

Exemple : Voici un exemple de 2,3 arbre :



Le 2,3 arbre T1 représentant l'ensemble
{10,110,160,210,260,390,410,610,651,710,891,897,980,990}

Voici une définition plus formelle d'un 2,3 arbre.

Définition

Un 2,3 arbre est un arbre T :

- tout sommet possède aucun enfant ou sinon 2 ou 3 enfants
- toutes les feuilles sont à une même distance de la racine.
- seules les feuilles possèdent une valeur. La valeur d'une feuille f est notée $valeur_T(f)$.
- tout sommet interne s, pour lequel nous notons ici :
  - m son nombre d'enfants
  - $(t_1,..,t_m)$ sa séquence d'enfants

  possède une séquence de m-1 entiers strictement ordonnées appelées balises notée $balises_T(s)=(b_1,..,b_{m-1})$ telle que pour tout feuille f descendant d'un enfant t_i avec $i\in[1,m]$ vérifie :
  - $b_{i-1} < valeur_T(f)$, si $1<i$.

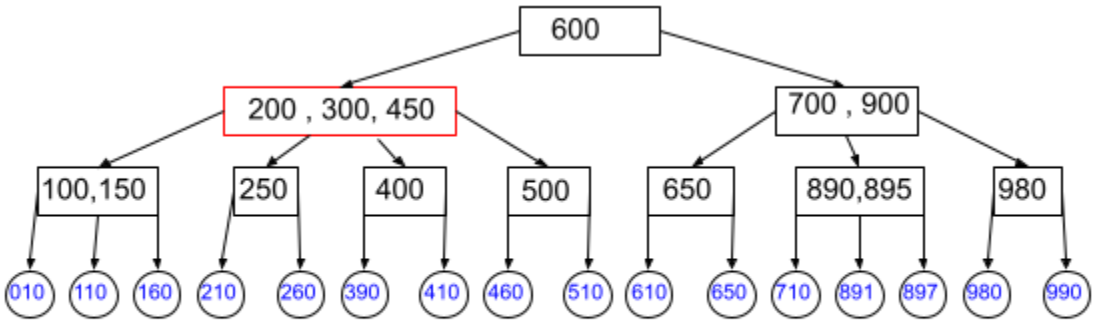
- $valeur_T(f) \leq b_i$ si i<m.

Afin de pouvoir insérer une nouvelle valeur ou supprimer un valeur dans un 2,3 arbre, nous manipulons des “presque 2,3 arbre” ainsi définis :

Définition
Un presque 2,3 arbre est un arbre T qui vérifie toutes les propriétés d’un 2,3 arbre exceptée une :

- il existe un unique sommet interne de T qui possède soit **1** unique enfant soit exactement **4** enfants. Ce sommet est appelé le sommet *déséquilibré*.

Exemple :
Voici un exemple de presque 2,3 arbre :



Un presque 2,3 arbre et son noeud déséquilibré (en rouge)

2) Implémentations

Voici quelques premières primitives permettant de manipuler les noeuds représentants les sommets d’un 2,3 arbre ou d’un presque 2,3 arbre :

```
estRacine        noeud → booléen
estInterne       noeud → booléen
estFeuille       noeud → booléen
estSentinelle    noeud → booléen
nbreEnfants      noeud → entier
valeur           noeud → élément
enfant           noeud x entier → noeud
balise           noeud x entier → élément
parent           noeud → noeud
```

Pour implémenter un tel noeud, nous proposons d’utiliser une structure contenant les 11 champs suivants :

```
estSentinelle                type booléen
nbreEnfants                  type entier
valeur                       type élément
```

```
parent                                type noeud
enfant1,enfant2,enfant3,enfant4   type noeud
balise1,balise2,balise3             type élément
```

3) Un premier algorithme

Voici un premier algorithme permettant de retourner la feuille pouvant contenir la valeur x :

```
fonction chercheFeuille(x : élément , T : arbre) : (booléen,noeud)
debut
      n ← enfant(sentinelleRacine(T),1) ;
      si estSentinelle(n) alors
            retourner (faux(),n) ;

      tantque non(estFeuille(n)) faire
            m ← nbreEnfants (n) ;
            si m = 1 OU x ≤ balise(n,1) alors
                  n ← enfant(n,1)
            sinon si m = 2 OU x ≤ balise(n,2) alors
                  n ← enfant(n,2)
            sinon si m = 3 OU x ≤ balise(n,3) alors
                  n ← enfant(n,3)
            sinon
                  n ← enfant(n,4)
      ;
      retourner (x=contenu(n),n)
fin
```

4) Autres algorithmes

Les algorithmes permettant de manipuler les 2,3 arbres ont des principes identiques à ceux des arbres binaires :

- on ajoute un élément (presque) à l'endroit où l'on ne l'y trouve pas
- on rééquilibre un presque 2,3 arbre en :
    - soit supprimant localement le déséquilibre
    - soit en remontant au parent le déséquilibre de son enfant

Ces algorithmes seront vus en TD.