

Cours d'Informatique MP2I

Antoine Domenech

2024-2025

TABLE DES MATIÈRES

Chapitre 0 – INTRODUCTION À L'INFORMATIQUE

0. Algorithmes et pseudo-codes	2
0. Problèmes	2
1. Algorithmes	3
2. Un langage de pseudo-code	4
<i>Variables et sauts conditionnels (p. 4). Fonctions (p. 6). Boucles (p. 7).</i>	
3. Convention de représentation des G.F.C.	9
4. Tableaux	10
1. Machines, programmes et langages	11
0. Machines, modèles	11
1. Programmes	12
2. Paradigmes	14

Chapitre 1 – PREUVES DE PROGRAMMES

0. Spécification d'une fonction	16
1. Terminaison	18
0. Variants et preuves de terminaison	18
1. Cas particulier des boucles Pour	20
2. Fonctions récursives	21
2. Correction	22
0. Correction partielle	22
<i>Sauts conditionnels (p. 22). Invariants de boucle (p. 22). Cas particulier des boucles Pour (p. 26).</i>	
1. Fonctions récursives	27
2. Correction totale	28

Chapitre 2 – MÉMOIRE

0. Représentation des types de base	30
0. Notion de données	30
1. Booléens	30
2. Différents types d'entiers	31
<i>Entiers non-signés (p. 31). Caractères (p. 33). Entiers signés (p. 33). Dépassements de capacité (p. 35).</i>	
3. Nombres à virgule (flottante)	36
<i>Notation mantisse-exposant pour l'écriture scientifique (p. 36). Erreurs d'approximation et autres limites (p. 37).</i>	
4. Tableaux	38
5. Chaînes de caractères	39
1. Compléments très brefs de programmation	40
0. Notion d'adresse mémoire	40
1. Portée syntaxique et durée de vie	40
<i>Définition et premiers exemples (p. 40). Exemples avancés (p. 43).</i>	
2. Organisation de la mémoire d'un programme	44
0. Bloc d'activation	44
1. Organisation de la mémoire virtuelle d'un programme	45
<i>Pile mémoire (p. 46). Tas mémoire (p. 46). À savoir faire (p. 47).</i>	

Chapitre 3 – COMPLEXITÉ

0. Complexité temporelle	50
0. Notion de complexité temporelle	50
<i>Opérations élémentaires (p. 51).</i>	
1. Notation de Landau	52
2. Exemples plus avancés	54
<i>Méthodologie (p. 54). Un exemple avec des boucles imbriquées (p. 54). Un exemple où il faut être très précis sur les itérations (p. 55). Écart incompressible entre les bornes (p. 56).</i>	
3. Pire cas, cas moyen, meilleur cas	56
4. Ordres de grandeur	57
1. Complexité spatiale	58
2. Cas particulier des fonctions récursives.....	59
0. Complexité temporelle des fonctions récursives	60
<i>Cas général : formule de récurrence (p. 60). Cas particuliers : arbre d'appels (p. 60).</i>	
1. Complexité spatiale des fonctions récursives	64
3. Complexité amortie	65
0. Méthodes de calcul	65
<i>Exemple fil rouge (p. 65). Méthode du comptable (p. 65). Méthode du potentiel (p. 67). Méthode de l'aggrégat (p. 69).</i>	
1. Remarques et compléments	70

Chapitre 4 – RÉCURSIVITÉ

0. Introduction.....	72
0. Notion de réduction	72
1. Notion de récursion	72
1. Exemples	73
0. Affichage d'un triangle	73
1. Tours de Hanoï	74
2. Exponentiation rapide	76
3. À propos des boucles	77
<i>Transformation boucle <-> récursion (p. 77). Récursivité terminale (hors-programme) (p. 78).</i>	
2. Comment concevoir une fonction récursive	78
3. Analyse de fonctions récursives	79
4. Compléments	79
0. Élimination des appels redondants	79
1. Avantages et inconvénients de la récursion	79
2. Querelle sémantique	80
3. Quand « déplier les appels » ?	80

Chapitre 5 – STRUCTURES DE DONNÉES (S1)

0. Tableaux dynamiques	82
0. Tableaux C vs listes OCaml	82
1. Fonctionnement	83
<i>Ajout dans un tableau dynamique (p. 84). Création, suppression (p. 85).</i>	
2. Analyse de complexité	85
3. Complément mi hors-programme : RETRAIT	85
1. Interfaces et types abstraits	87
0. Aspects pratiques	87
<i>Librairies (p. 87). Interface (p. 88). Étapes de la compilation (p. 90).</i>	
1. Aspects théoriques	91

2. Structures de données séquentielles	93
0. Types de base	93
1. Listes linéaire	93
<i>Implémentation par tableaux de longueur fixe (p. 93). Implémentation par tableaux dynamiques (p. 94). Implémentation par listes simplement chaînées (p. 94). Variantes des listes chaînées (p. 97).</i>	
2. Piles	97
<i>Implémentations par tableaux (p. 98). Implémentation par listes simplement chaînées (p. 99).</i>	
3. Files	100
<i>Implémentations par tableaux circulaires (p. 100). (Hors-programme) Amélioration avec des tableaux dynamiques (p. 103). Implémentations par listes simplement chaînées (p. 103). Implémentation par listes doublement chaînées (p. 104). Par double pile (p. 104).</i>	
4. Variantes des files	105
3. Aperçu des structures de données S2	105

Chapitre 6 – BONNES PRATIQUE DE PROGRAMMATION

0. Écrire mieux	108
1. Tester mieux	108

Chapitre 0

INTRODUCTION À L'INFORMATIQUE

Notions	Commentaires
Notion de programme comme mise en oeuvre d'un algorithme. Paradigme impératif structuré, paradigme déclaratif fonctionnel, paradigme logique.	On ne présente pas de théorie générale sur les paradigmes de programmation, on se contente d'observer les paradigmes employés sur des exemples. La notion de saut inconditionnel (instruction GOTO) est hors programme. On mentionne le paradigme logique uniquement à l'occasion de la présentation des bases de données.
Caractère compilé ou interprété d'un langage.	Transformation d'un fichier texte source en un fichier objet puis en un fichier exécutable. [...]

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Graphe de flot de contrôle. [...]	[...]

Extrait de la section 1.3 du programme officiel de MP2I : « Validation, test ».

SOMMAIRE

0. Algorithmes et pseudo-codes	2
0. Problèmes	2
1. Algorithmes	3
2. Un langage de pseudo-code	4
<i>Variables et sauts conditionnels (p. 4). Fonctions (p. 6). Boucles (p. 7).</i>	
3. Convention de représentation des G.F.C.	9
4. Tableaux	10
1. Machines, programmes et langages	11
0. Machines, modèles	11
1. Programmes	12
2. Paradigmes	14

Définition 1 (src : CNRTL).

« Informatique (Subt., Fem.) : Science du traitement rationnel, notamment par des machines automatiques, de l'information considérée comme le support des connaissances humaines et des communications dans les domaines technique, économique et social. »

0 Algorithmes et pseudo-codes

0.0 Problèmes

Définition 2 (Problème).

Un problème est composé de deux éléments :

- une instance (aussi appelée entrée).
- une question ou une tâche à réaliser sur cette instance.

Exemple.

- PGCD :

Entrée : x, y deux entiers strictement positifs.

Tâche : calculer leur plus grand diviseur commun.

- TRI :

Entrée : des éléments comparables selon un ordre \leq .

Tâche : trier ces entiers par ordre \leq croissant.

- PLUS COURT CHEMIN :

Entrée : une carte de réseau de transport.

un point de départ sur ce réseau, et un d'arrivée.

Tâche : calculer un plus court chemin du départ à l'arrivée.

- PUZZLE :

Entrée : un cadre et des pièces de puzzle.

Question : peut-t-on agencer les pièces dans le cadre selon les règles ?

- #PUZZLE :

Entrée : un cadre et des pièces de puzzle.

Tâche : calculer de combien de façons il est possible d'agencer légalement les pièces dans le cadre.

- SEUIL-PUZZLE :

Entrée : un cadre et des pièces de puzzle.

un entier $n \in \mathbb{N}$

Question : Existe-t-il plus de n (inclus) façons d'agencer légalement les pièces ?

On parle de problème de :

- **décision** si la réponse est binaire (Oui/Non).
- **optimisation** si la réponse est une solution "meilleure" que tout autre.
- **dénombrement** si la réponse est un nombre de solutions.
- liste non exhaustive.

Exemple. Associer un type (ou plusieurs) à chacun des problèmes précédents.

0.1 Algorithmes

Le terme « algorithme » a de nombreux sens. La définition que j'en donne ici est celle que nous utiliserons cette année ; mais ce n'est forcément pas celle du langage courant¹. Cela ne signifie pas que l'une des deux est meilleure que l'autre : un même mot a des sens différents dans des contextes différents, c'est normal, et cela nous arrivera même entre différentes sections d'un même chapitre.

Définition 3 (Algorithme).

Un algorithme est la description finie d'une suite d'opérations résolvant un problème. On veut de plus avoir :

- Terminaison : la suite d'opération décrite est atteinte toujours sa fin. On dit que l'algorithme termine.
- Précisément définie : les opérations sont définies précisément, rigoureusement et sans ambiguïté.
- Entrée : un algorithme a 0 ou plusieurs quantités de nature connue et spécifiée qui lui sont données avant ou pendant son exécution.
- Sortie : un algorithme renvoie une ou plusieurs quantités dont le lien avec les entrées est spécifié.
- Éléментарité : les opérations décrites doivent être assez simples pour être réalisées par une personne avec un papier et un crayon en temps fini.

Remarque.

- On rapproche souvent les algorithmes des recettes de cuisine.
- Aucune notion de langage informatique n'est présente dans cette définition. Langage naturel², idéogrammes, schéma, pseudo-code, Python, C, OCaml, SQL : tout pourrait convenir.
- Attention à ne pas confondre la terminaison avec le fait que la description elle-même est finie.
- Le caractère précisément défini demande que l'on sache *quoi* faire, alors que l'élémentaire demande que l'on sache *comment* le faire.
- L'éléментарité dépend du public-cible. « Diviser un entier par un autre » est une étape élémentaire pour vous, pas pour des élèves de CE2. De même pour la précision : elle peut dépendre du contexte.
- Cette définition d'algorithme ne contient aucun critère d'efficacité en temps (ou en espace mémoire).

Exemple. L'algorithme d'Euclide pour résoudre PGCD est un exemple connu. Dedans, « assigner $x \leftarrow y$ » signifie « écraser la valeur de x en la remplaçant par celle de y ».

Algorithme 1 : Algorithme d'Euclide.

Entrées : m et n deux entiers strictement positifs

Sorties : $\text{pgcd}(m, n)$

- 1 E1 (calcul du reste) : diviser m par n , et noter r le reste
 - 2 E2 (égal à 0 ?) : si $r = 0$, renvoyer n
 - 3 E3 (réduction) : assigner $m \leftarrow n, n \leftarrow r$. Recommencer en E1.
-

1. Il me semble que de plus en plus, le terme a une connotation d'obscur, d'inconnu, de traitement caché. Ce n'est pas le contexte dans lequel je me place dans ce cours. Notez que je n'affirme pas ladite connotation est absurde.

2. Le langage naturel est la langue des humains-es.

On peut représenter cet algorithme sous une forme visuelle :

FIGURE 1 – Graphe de Flot de Contrôle de l'algorithme 1.

Exercice. Vérifier qu'il s'agit bien d'un algorithme. Le faire tourner sur l'instance $m = 15, y = 12$.

0.2 Un langage de pseudo-code

Pour utiliser une machine automatique, on doit transformer l'algorithme en une suite d'instruction que la machine comprend. La première et principale étape consiste en générale à écrire cet algorithme dans un *langage de programmation*. Ceux-ci ont de nombreuses différences, mais aussi énormément de concepts communs. On parle de **pseudo-code** pour décrire quelque chose qui ressemble à des langages de programmation, manie ces concepts communs, mais limite les détails techniques propres à chaque langage.

Le pseudo-code que je vais présenter ici est très **impératif**, c'est à dire que l'on programme en décrivant des modifications successives de la mémoire à effectuer.

0.2.0 Variables et sauts conditionnels

Définition 4 (Variable, version intuitive).

Une variable peut-être vue comme une boîte qui a :

- un identifiant, aussi appelé nom de variable.
- un contenu. Celui-ci peut-être par exemple :
 - une valeur précise indiquée.
 - le résultat d'un calcul.
 - la valeur renvoyée par un autre algorithme sur une entrée précise.

On note $n \leftarrow \alpha$ le fait que la variable n contienne dorénavant le contenu α .

Exemple.

$x \leftarrow 3$	on stocke 3 dans x
$x \leftarrow (9 - 4)$	x contient dorénavant 5 et non plus 3
$y \leftarrow 2x$	on stocke 10 dans y
$x \leftarrow y^3$	x contient dorénavant 1000 et non plus 5.

Définition 5 (Saut conditionnel).

Un saut conditionnel permet d'exécuter différentes instructions selon une valeur logique, souvent le résultat d'un test. On note :

si <i>valeur logique</i> alors instructionsDuAlors sinon instructionsDuSinon
--

Pseudo-code

G.F.C associé.

Les instructions du Alors s'effectuent si la valeur logique s'évalue à Vrai, celles du Sinon... sinon. Ces instructions sont respectivement nommées le **corps du Alors/Sinon**.

Lorsque le corps du Sinon est vide, on n'écrit tout simplement pas le Sinon (et son corps).

Exemple.

```

1  x ← 3
2  si x ≥ 2 alors
3    | x ← 3x
4    | x ← x - 1
5  sinon
6    | x ← 1.5x
7    | x ← -2x

```

À la fin de l'exécution de ce code, x contient 8.

La notion de conditionnel est central en informatique. C'est elle qui permet de faire en sorte qu'un programme puisse avoir des comportements différents en fonction des informations qui lui sont données ! Imaginez un moteur de recherche internet qui renvoie toujours les mêmes résultats, peu importe l'entrée...

Remarque.

- Les barres verticales sont très utiles à la relecture. Je vous encourage *très très fortement* à les mettre dès que vous êtes sur papier, y compris lorsque vous écrivez du vrai code.
- En anglais, Si/Alors/Sinon se dit If/Then/Else.
- Par abus du langage, on parle souvent du « corps du Si » pour désigner le corps du Alors.
- Un saut conditionnel est lui-même une instruction, et on peut donc les imbriquer :

Résumé grossier du fonctionnement de M. Domenech aux portes ouvertes.

```

1  si l'élève veut faire 12h de maths alors
2    | si l'élève aime ou veut découvrir l'informatique alors
3    | | orienter en MP2I
4    | sinon
5    | | orienter en MPSI
6  sinon
7    | orienter en PCSI (10h de maths quand même)

```

Notez que 10h vs 12h de maths n'est pas un excellent critère. Dans le groupe nominal « résumé grossier », il se pourrait que le qualificatif soit plus important que le substantif.

- Considérez l'imbriication suivante : « Si b_0 Alors Si b_1 Alors $instr_0$ Sinon $instr_1$ ». Ici, il n'est pas possible de savoir de quel Si $instr_1$ est le Sinon. On parle du problème du *Sinon pendant* (Dangling else d'Alan Turing) : il y a une ambiguïté sur l'imbriication.

Sur papier, une écriture avec retour à la ligne, indentation (décalage horizontal du corps) et barre verticale permet de l'éviter.

Les langages de programmation, eux, ont chacun une règle qui force une unique façon d'interpréter cette imbrication. Il s'agit généralement soit d'indiquer explicitement la fin du saut conditionnel, soit de la règle du parenthésage à gauche d'abord ; nous en reparlerons.

- Les sauts inconditionnels (« goto ») sont hors-programme et prohibés.

0.2.1 Fonctions

Lorsque vous avez découvert les fonctions en maths, on vous les a possiblement présenté comme des machines qui prennent quelque chose en entrée et le transforme en autre chose (en termes savants, on dit qu'elles associent un antécédant à une image). C'est la même idée en informatique.

Définition 6 (Fonction, version intuitive).

Une **fonction** est transforme des **arguments** en une nouvelle valeur. On dit aussi qu'elle transforme des **entrées** en **sorties**.

En pratique, une fonction est un bloc de pseudo-code qui attend certaines variables en entrée, décrit des modifications à faire dessus, et indique une valeur à renvoyer à la fin. On dit que les instructions contenues dans la fonction forment le **corps** de la fonction.

On dit qu'une fonction est **appelée** lorsqu'on l'utilise pour calculer la sortie associée à une entrée. On dit qu'elle **renvoie** sa sortie. On note généralement comme en maths : $f(x_0, x_1, \dots)$ est la valeur renvoyée par la fonction f sur l'entrée (x_0, x_1, \dots) .

Exemple. Voici deux fonctions. Que font-elles ?

Entrées : x une variable contenant un entier.

Sorties : L'entier 2.

1 **renvoyer** 2

Entrées : x une variable contenant un entier.

Sorties : L'entier x^2 .

1 $y \leftarrow x$

2 **renvoyer** $x * y$

Exercice. Proposer une autre façon d'écrire le corps de cette seconde fonction qui évite la création d'une variable dans la fonction.

Notez que la notion de corps induit une notion de « monde extérieur » : on distingue le (pseudo-)code qui se trouve dans la fonction du (pseudo-)code qui se trouve ailleurs.

Exemple. Indiquer ce que vaut `main()` avec les fonctions ci-dessous :

Entrées : x une variable contenant un entier.

Sorties : y un entier

1 $y \leftarrow 2x$

2 $x \leftarrow 0$

3 **renvoyer** y

Fonction `fun`

Entrées : Rien.

Sorties : Mystère.

1 $x \leftarrow 3$

2 $z \leftarrow \text{fun}(x)$

3 **renvoyer** x

Fonction `main`

Il y a deux réponses possibles : 0 (car `fun` a remis x à 0) et 3 (car `fun` ne travaille pas sur le x d'origine mais sur une copie). Cela correspond respectivement aux deux notions suivantes :

Définition 7 (Passages et effets secondaires).

Soit F une fonction et x un de ses arguments. On dit que x est :

- **appelé par référence** si toute modification de x par le corps de F modifie également le x initial (la variable qui avait été donnée en entrée à F lors de son appel). Tout se comporte comme si le corps de la fonction et le monde extérieur partagent le même x .
- **appelé par valeur** si, au contraire, toute modification de x par le corps de F ne modifie pas le x initial. Tout se comporte comme si le corps de la fonction manipule une copie du contenu x , distincte du x du monde extérieur.

Sauf mention contraire explicite dans ce cours, tous les arguments sont appelés par valeur. La mention contraire principale concernera les tableaux.

En particulier, dans l'exercice précédent, la réponse que donnerait tout langage de programmation raisonnable³ est 3.

3. Et toute étudiant-e de MP2I raisonnable ;)

Définition 8 (Effets secondaires, version courte).

L'ensemble des modifications effectuées par une fonction sur son monde extérieur est appelé les **effets secondaires** de la fonction.

Remarque.

- On parle aussi de **passage par référence/valeur**. Nous verrons également le **passage par pointeur**, qui à notre niveau ne sera pas distinct du passage par référence.
- Le passage par valeur est coûteux : l'appel de la fonction doit consommer du temps et de l'espace mémoire afin de créer une copie de l'argument concerné. Les langages de programmation réservent donc généralement le passage par valeur aux variables de taille raisonnable (comme des entiers) et passent tout ce qui est potentiellement gros (comme une collection de valeurs plus simples) par référence.
Par exemple, Python passe les entiers par valeur et les listes par référence.
- L'exemple de `fun` et `main` montre également un exemple où une fonction en appelle une autre, cela nous arrivera très fréquemment. **Le rôle principal des fonctions est de décomposer et structurer un gros code en petites unités élémentaires simples à comprendre.**
Nous verrons également qu'il peut-être très comode pour une fonction de s'appeler elle-même, ce que l'on nomme la **réursion**.

Définition 9 (Prototype, signature).

- La donnée de l'identifiant (le nom) d'une fonction ainsi que de ses types d'entrées et de sortie est appelée la **signature** de la fonction.
- Si l'on ajoute les identifiants des entrées, on parle de **prototype**.

Exemple.

- Signature de PGCD : PGCD : entier , entier \rightarrow entier
- Prototype de PGCD : PGCD : *m* entier, *n* entier \rightarrow entier

En mathématiques, on note $pgcd : \mathbb{N}^* \times \mathbb{N}^* \rightarrow \mathbb{N}^*$. C'est la signature!

0.2.2 Boucles

De même que l'informatique sans Si/Alors/Sinon n'est pas très intéressante, il nous manque pour l'instant une capacité très importante : pouvoir répéter des instructions un nombre inconnu de fois. C'est par exemple ce que faisait l'algorithme d'algorithme grâce à son étape E3 qui indiquait « retourner en E1 ». On dit que E1-E2-E3 forment une boucle.

Définition 10 (Boucle à précondition).

Une boucle à précondition, ou boucle TantQue (« While ») répète des instructions tant qu'une valeur logique (souvent une condition) est vraie :

tant que <i>valeur logique</i> faire instructions
--

Pseudo-code

G.F.C associé.

Les instructions répétées sont appelées le **corps**. On dit qu'on effectue une **itération** de la boucle lorsque l'on exécute les opérations du corps.

Exemple.

Entrées : x un entier.

```

1  $p \leftarrow 0$ 
2 tant que  $x < 436$  et  $p < 10$  faire
3   |  $x \leftarrow 2x$ 
4   |  $p \leftarrow p + 1$ 
5 renvoyer  $x$ 
    Fonction demo

```

Pour calculer à la main $\text{demo}(1)$, on peut écrire un tableau où on décrit le contenu des variables à la fin de chaque itération :

	p	x
Avant la première iter.	0	1
Fin de la première iter.	1	2
Fin de la seconde.	2	4
Fin de etc	3	8
Fin de etc	4	16
Fin de etc	5	32
Fin de etc	6	64
Fin de etc	7	128
Fin de etc	8	256
Fin de etc	9	512

Après cette dernière itération, on quitte la boucle car $x \geq 436$ et donc la valeur logique d'entrée dans la boucle devient fausse.

Remarque.

- Si la valeur logique du TantQue reste éternellement vraie, on ne quitte jamais la boucle et on effectue de sitérations à l'infini. On parle de **boucle infinie**. C'est un des bugs les plus courants.
- Plusieurs fois dans l'année, nous referons des tableaux d'itération comme celui-ci. Nous noterons avec des primes (p' et x') les quantités en fin d'itération afin d'éviter des confusions.
- On peut tout à fait faire le tableau avec les quantités en début d'itération. Cela évite la ligne « avant la première itération », mais demande souvent de rajouter une ligne à la fin « début de l'itération qui n'a pas lieu car on quitte la boucle ».

Dans ce cours et toute cette année, je travaillerai plutôt avec les fins.

Exercice. Calculer l'image de 14 par la fonction suivante :

Fonction Collatz	
Entrées : x un entier strictement positif.	
Sorties : Mystère mystère.	
1	tant que $x > 1$ faire
2	si x est pair alors
3	$x \leftarrow x/2$
4	sinon
5	$x \leftarrow 3x + 1$
6	renvoyer x

CultureG : les valeurs successives de x durant ces itérations forment ce que l'on appelle une suite de Syracuse^{4,5}. Il est conjecturé et presque prouvé⁶ que toutes les suites de Collatz contiennent tôt ou tard l'entier 1.

4. Pas Syracuse comme la ville grecque, voyons. Syracuse comme l'université de Syracuse, bien sûr. Qui tient son nom de la ville de Syracuse, New-York (l'état, pas la ville), États-Unis. Évidemment.

5. L. Collatz, lui, est un mathématicien ayant participé à populariser l'étude de ces suites.

6. « Presque » au sens où T. Tao a prouvé que cette conjecture est vraie pour presque tous les entiers. La définition de « presque tous » est subtile. Retenez qu'en pratique elle est vraie.

Définition 11 (Boucle itérative).

Une boucle itérative, ou boucle Pour (« For »), permet de répéter des instructions un nombre connu de fois tout en mémorisant le numéro de la répétition en cours. C'est en fait un simple raccourci pour une boucle TantQue :

```
pour compteur allant de 0 inclus à n
  exclu faire
    | instructionsDuCorps
```

Pseudo-code d'un For

```
compteur ← 0
tant que compteur < n faire
  | instructionsDuCorps
  | compteur ← compteur + 1
```

Pseudo-code du While équivalent

G.F.C. associé.

On parle là aussi du **corps** de la boucle.

Remarque.

- Notez que la version While et la version G.F.C. explicitent toutes deux le fait que **la mise à jour du compteur a lieu à chaque itération, en toute fin d'itération**.
- On croise souvent les boucles PourChaque, qui permettent d'effectuer le corps de la boucle sur chacun des éléments d'une collection d'objets (e.g. « PourChaque *x* entier de $\llbracket 8; 35 \rrbracket$). C'est par exemple ce que fait `for x in ...` de Python.

Définition 12 (break, continue).

On crée deux nouvelles instructions pour les boucles : **break** permet de quitter instantanément la boucle en cours, et **continue** permet de passer immédiatement à l'itération suivante (en ayant mis à jour le compteur).

Ces deux instructions ne s'appliquent *que* à la boucle en cours (et pas aux éventuelles sur-boucles en cas d'imbrication).

Attention, ces deux instructions sont une erreur commune de bugs des débutant·es. La plus « dangereuse » des deux, `continue`, est hors-programme pour cette raison.

0.3 Convention de représentation des G.F.C.

Le flot d'un code désigne l'enchaînement de ses instructions. Un graphe de flot de contrôle est une représentation du déroulement de l'exécution (pseudo-)code. Le but est de représenter quelle instruction peut mener à quelle instruction, c'est à dire de représenter le flot.

Dans les schémas précédents, on a utilisé les conventions suivantes :

- Le début du flot est marqué par un triangle (base en bas), et par une double flèche y menant. Au départ de la flèche, on indique les variables d'entrées.
- Les fins du flot sont marquées par un triangle (base en haut), et une double flèche en sortant. À la fin de la flèche, on indique la sortie / valeur renvoyée par cette fin du flot.

- Les instructions sont écrites dans des rectangles. On peut utiliser un même rectangle pour tout un corps.
- Le passage d'un point à un autre est représenté par une flèche simple.
- Les embranchements (sauts conditionnels, entrée/sortie de boucle) sont représentés par un "losange", coin vers le bas. Dans le losange on indique la valeur logique qui est interrogée. Sur les flèches sortantes, on indique Vrai/Faux (ou Oui/Non) pour indiquer quelle flèche correspond à quelle option.
- On représente les appels de fonction par un cercle contenant le nom de la fonction. On détaille éventuellement le GFC de la fonction à part.

Exemple. Cf G.F.C. de l'algorithme d'Euclide.

0.4 Tableaux

On a parfois besoin de stocker une très grande quantité de données, au point où créer une variable par donnée ne serait pas pratique. Par exemple, si je veux stocker la note d'anglais au bac de chaque élève de CPGE scientifique de Camille Guérin, il me faudrait $48 \times 5 = 240$ variables.

Définition 13 (Tableaux, version intuitive).

On peut penser un tableau comme un long meuble muni de L tiroirs numérotés. Chacun des tiroirs est une variable.

Un peu plus formellement, un **tableau** est défini par sa **longueur**, souvent notée L , qui est son nombre de **cases**. Chaque case se comporte comme une variable. On peut accéder à chaque case à l'aide de son indice, compris entre 0 et $L - 1$ inclus : si T est un tableau et i un indice valide, on note $T[i]$ la case d'indice i de T .

Le contenu de toutes les cases d'un même tableau doit avoir la même nature (tous des entiers, ou bien tous des nombres à virgule, etc).

Dans ce cours, si un tableau à L cases contient dans l'ordre les valeurs x_0, x_1, \dots, x_{L-1} , on le notera en pseudo-code $[x_0; x_1; \dots; x_{L-1}]$.

Lors des appels de fonction, un tableau se comporte comme si son contenu était passé par référence.

Exemple.

<pre> 1 T ← tableau de longueur 3 2 T[0] ← 42 3 T[1] ← -20 4 T[2] ← T[0] + T[1] </pre>	À la fin de l'exécution, T est $[42; -20; 22]$.
--	--

Exemple. La fonction `mainbis()` ci-dessous est la fonction constante égale à 0, car les cases du tableau se comportent comme si passées par référence :

Entrées : T un tableau d'entiers.

```

1 y ← 2 × T[0]
2 T[0] ← 0
3 renvoyer y

```

Fonction `funbis`

Entrées : Rien.

```

1 T ← tableau d'entiers non-vide
2 z ← funbis(T)
3 renvoyer T[0]

```

Fonction `mainbis`

Remarque.

- La longueur d'un tableau est fixe. Les tableaux ne sont *pas* redimensionnables. En particulier, les listes Python ne sont pas des tableaux. C'est une structure plus complexe, qui permet plus de choses mais est un peu plus lente : ce sont des tableaux dynamiques (de pointeurs), que nous verrons comment construire cette année.
- Très souvent, nous serons amenés à traiter les unes après les autres les cases d'un tableau ; on dit qu'on **parcourt** le tableau. Les boucles sont l'outil adapté pour cela :

<pre> 1 i ← 0 2 tant que i < L faire 3 traiter T[i] 4 i ← i+1 </pre>	<pre> 1 pour i allant de 0 inclus à L exclu faire 2 traiter T[i] </pre>
---	---

Avec une boucle For

Avec une boucle While

- Le contenu des cases d'un tableau T peut tout à fait être des tableaux : dans ce cas, la case $T[i]$ de T est elle-même un tableau, et on peut donc en demander la case j . On demande donc $(T[i])[j]$, que l'on préfère écrire $T[i][j]$.

1 Machines, programmes et langages

1.0 Machines, modèles

Définition 14 (Ordinateur (src : wikipedia.fr)).

Un ordinateur est un système de traitement de l'information programmable.

Exemple.

- L'ordinateur PC-prof de la salle B004. C'est le sens commun du terme « ordinateur », et c'est sur ces ordinateurs-ci que nous apprendrons à programmer.
- Vos téléphones, consoles de jeu.
- Les télévisions, voitures, vidéoprojecteurs, etc modernes.
- Un métier Jacquart.
- Certains jeux, comme Minecraft ou BabaIsYou (on peut simuler un ordinateur dedans).
- Les machines à laver modernes (si si).

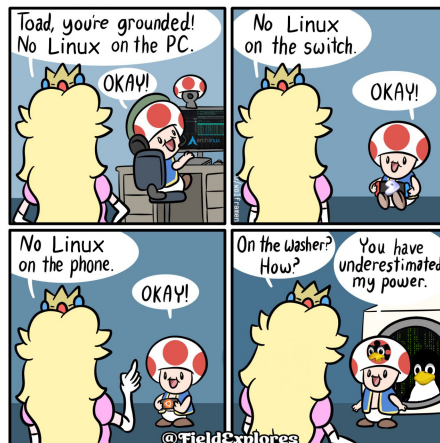


FIGURE 2 – Inspiré de faits réels.

Définition 15 (Modèle de calcul).

Un modèle de calcul est un ensemble de règles qui permettent de simuler les étapes d'un calcul. C'est souvent une abstraction d'une machine de calcul (e.g. un ordinateur) du monde réel, afin de prouver des propriétés sur la machine et son fonctionnement.

Remarque. Le modèle est une *simplification*. Par exemple, on lui suppose souvent une mémoire infinie⁷.

7. Et moi qui me trouve chanceux avec mes $2 \times 8 \times \text{Go}$ de RAM, alors que je pourrais viser l'infini...

Exemple.

- Machine à RAM : représente un ordinateur muni d'une mémoire vive (RAM) infinie.
- Machine de Turing : idem, sauf que la mémoire est un ruban qu'il faut dérouler pour aller d'un point à un autre de la mémoire. Ce modèle représente les premiers ordinateurs modernes et a eu un très fort impact théorique et historique.
- Lambda-calcul : a inspiré la famille des langages de programmation fonctionnels. Très théorique, ne correspond pas à une machine réelle.

1.1 Programmes

Voici un schéma (très) simplifié des ordinateurs modernes⁸ :

FIGURE 3 – B-A-BA de l'architecture d'un ordinateur.

Définition 16 (Programme.).

Un **programme**, aussi appelé exécutable, est une suite d'instructions à effectuer par le processeur. Ces instructions sont écrites en **langage machine**.

Remarque. Un compilateur ne peut lire *que* du langage machine, qui est une suite de 0 et de 1 !

Exemple. Ce sont des opérations très basiques. Voici un exemple "lisible" par des humains, où l'on a "traduit" les suites de 0 et de 1 du langage machine en mots-clefs (on parle de **langage assembleur**) :

<code>mov r3, #17</code>	<i>stocke 17 dans la case mémoire r3</i>
<code>add r3, r3, #42</code>	$r3 \leftarrow r3 + 42$

Le langage assembleur, et plus encore le langage machine, est très dur à lire et permet assez peu d'abstraction. C'est pour cela que nous ne programmerons pas en assembleur mais dans des langages de programmation, pensés pour être plus proches des humains.

Définition 17 (Langage de programmation).

Un langage de programmation est défini ensemble de règles syntaxiques propres. L'objectif est de rédiger des programmes de manière plus lisible que le langage machine. « Coder dans un langage de programmation » signifie « écrire un texte qui respecte les règles du langage. »

8. Non-contractuel, j'ai vraiment simplifié comme un bourrin.

Définition 18 (Compilateur).

Pour que le fichier écrit dans un langage de programmation devienne un programme, il faut le traduire en langage machine : un tel traducteur est appelé un **compilateur**.

Lorsque l'on traduit d'un langage de programmation vers un autre langage de programmation (et non vers le langage machine), on dit que l'on **transpile** (pour distinguer de compile).

Exemple.

- C, OCaml, Rust, C++, Haskell, Fortran, etc sont des
- Le langage machine est techniquement un langage de programmation (c'est même le seul qui n'a pas à être compilé).
- L'assembleur (la traduction est assez directe, il suffit⁹ de traduire mot à mot en 0-1).
- On considère souvent que les langages comme le HTML, qui ne servent pas à exprimer des calculs mais à décrire un agencement, ne sont pas des langages de programmation. Pour HTML, on parle de langage de balisage.
Cela ne signifie pas que travailler sur ces langages est moins "noble" ou plus simple. Cela signifie juste que ce n'est pas la même chose car le but (et les moyens offerts) est radicalement différent.

Remarque.

- Pour un même langage, différents compilateurs peuvent exister. Ils peuvent proposer des façons de traduire différentes, qui optimisent des critères distincts.
Par exemple, pour C, il en existe au moins 3 : gcc (que nous utiliserons), clang et compcert.
- Un compilateur peut faire des modifications à votre code afin de l'accélérer. Par exemple, si un compilateur détecte qu'une boucle ne sert qu'à calculer $\sum_{i=1}^n i$, il peut supprimer la boucle et la remplacer par $\frac{n(n+1)}{2}$.
- Différents processeurs ont des langages machines différents. Si l'on veut distribuer un logiciel, il faut donc en prévoir une version compilée par langage machine que l'on veut pouvoir supporter. C'est pour cela que vous trouvez par exemple des versions arm et des version x86 des logiciels sur leurs pages de téléchargement (en réalité il faut aussi une version par système d'exploitation : différents Linux, MacOS, Windows, etc).
- Une fois un code compilé, il n'est plus nécessaire d'avoir le compilateur (ou le fichier source du code) pour lancer le programme : seule compte la version compilée.

Définition 19 (Interpréteur).

Un interpréteur est un programme qui lit du code écrit dans un langage de programmation et l'exécute au fur et à mesure de sa lecture.

Exemple. Python, Java, OCaml (qui dispose d'un compilateur *et* d'un interpréteur).

Remarque.

- Le code n'est donc jamais traduit en un programme exécutable et a besoin de l'interpréteur pour fonctionner.
- Le fait que les lignes sont lues les unes et interprétées après les autres permet beaucoup moins d'optimisations qu'un compilateur (qui lit tout le fichier avant de compiler). Un code interprété est donc typiquement plus long qu'un code compilé (et consomme souvent plus de mémoire car il faut faire tourner l'interpréteur en plus du code).
- L'avantage est que l'on peut exécuter uniquement petit morceau du code grâce à l'interpréteur. Cela permet de tester un morceau choisi en ignorant les erreurs qui pourraient se trouver ailleurs. À l'inverse, un compilateur refusera de compiler un fichier tant qu'il reste un non-respect des règles du langage dedans.

9. Modulo simplification pédagogique.

- Un autre avantage est qu'il n'y a pas besoin de recompiler pour chaque langage processeur ! Il faut juste avoir accès à un interpréteur (lui-même un programme compilé en général) pour ce langage processeur.
- Différents interpréteurs peuvent exister pour un même langage. Python en a 3, chacun écrits dans un langage différent. Vous avez probablement utilisé sans le savoir cpython, codé en C, qui est l'interpréteur le plus courant.
- On parle de **langage interprété** (respectivement **langage compilé**) pour désigner un langage qui dispose d'un compilateur (respectivement interpréteur).

1.2 Paradigmes

Définition 20 (Paradigmes de programmation).

Il existe différentes façons de penser la programmation :

- **paradigme impératif** : programmer, c'est décrire une suite de modifications de la mémoire.
- **paradigme fonctionnel** : programmer, c'est décrire une le calcul à effectuer comme une suite d'évaluation de fonctions mathématiques.
- **paradigme logique** : programmer, c'est décrire très précisément le résultat attendu à l'aide d'un ensemble de formules logiques.

Un langage peut correspondre à différents paradigmes.

Ce sont des concepts qui se comprennent mieux en pratiquant.

Exemple.

- Pseudo-code ci-dessus, C, Python, Rust, C++, OCaml : paradigme impératif.
- OCaml, Haskell, Erlang, Elixir : paradigme fonctionnel.
- Prolog, un peu SQL : paradigme logique.

Chapitre 1

PREUVES DE PROGRAMMES

Notions	Commentaires
Terminaison. Correction partielle. Correction totale. Variant. Invariant.	La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine.

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Spécification des données attendues en entrée, et fournies en sortie/retour.	On entraîne les étudiants à accompagner leurs programmes et leurs fonctions d'une spécification. Les signatures des fonctions sont toujours précisées.

Extrait de la section 1.2 du programme officiel de MP2I : « Discipline de programmation ».

SOMMAIRE

0. Spécification d'une fonction.....	16
1. Terminaison	18
0. Variants et preuves de terminaison	18
1. Cas particulier des boucles Pour	20
2. Fonctions récursives	21
2. Correction.....	22
0. Correction partielle	22
<i>Sauts conditionnels (p. 22). Invariants de boucle (p. 22). Cas particulier des boucles Pour (p. 26).</i>	
1. Fonctions récursives	27
2. Correction totale	28

0 Spécification d'une fonction

Rappels du Chapitre 0 (Introduction) :

- La signature d'une fonction est la donnée de son identifiant, du type de chacune de ses entrées et du type de sa sortie.
- Le prototype d'une fonction est la même chose à ceci près que l'on indique en plus le nom de chacun des variables.

```
1 bool est_positif(int x) {
2   return (x >= 0);
3 }
```



- Signature : $\text{est_positif} : \text{int} \rightarrow \text{bool}$
- Prototype : $\text{est_positif} : (x : \text{int}) \rightarrow \text{bool}$

Remarque.

- La façon d'écrire les prototypes « à la mathématiques » ci-dessus ressemble à ce que l'on utilisera en OCaml.
- En C, il est valide d'annoncer qu'une fonction existera¹ en déclarant uniquement sa signature ou son prototype : `bool est_positif(int);` ou `bool est_positif(int x);`.

Définition 1 (Spécification).

La **spécification** d'une fonction est la description de ce qu'elle est censée faire, c'est à dire qu'elle indique :

- Entrées :
 - pour chacune d'entre elles, son type.
 - les éventuelles conditions supplémentaires que ces entrées doivent respecter. Par exemple : « x , un entier > 3 » ou encore « T , un tableau d'entiers triés par ordre croissant ».
 De telles conditions sont appelées des **pré-conditions**.
- Sortie :
 - le type de la valeur renvoyée.
 - les conditions attendues sur la valeur renvoyée (généralement en lien avec les entrées).
 - les éventuelles modifications que fait la fonction sur « le monde extérieur », c'est à dire sur des variables (ou objets mémoire) que la fonction n'a pas créé elle-même : modification d'un argument passé par référence, modification d'une variable globale, enregistrement d'un fichier sur le disque dur, affichage sur un écran, etc. On parle d'**effets secondaires** (« *side effects* »).

On appelle **post-conditions** la donnée des conditions attendues sur la valeur renvoyée et des effets secondaires.

Remarque.

- On peut voir la spécification comme un contrat entre l'utilisateur de la fonction et celle-ci : l'utilisateur de la fonction s'engage à vérifier les pré-conditions, en échange de quoi la fonction s'engage à vérifier les post-conditions.
Si les pré-conditions ne sont pas remplies lors d'un appel, la fonction n'est tenue à rien et peut renvoyer tout et n'importe quoi. Cependant, il est pertinent de plutôt lever un message d'erreur et interrompre le programme dans ce cas (beaucoup plus simple à déboguer!).
- Quand il n'y a pas d'effets secondaires, on ne les spécifie pas (au lieu de spécifier leur absence).
- L'anglicisme « effets de bord » pour « effets secondaires » existe.
- On essaye de limiter les effets secondaires au strict nécessaire, et on veille à toujours être exhaustif quand on les liste. Il s'agit de l'une des sources de bogues les plus communes!

1. On s'engage à en fournir le code source tôt ou tard.

Exemple. La fonction `max` est bien spécifiée et respecte cette spécification. par contre, `apres_moi_le_deluge` a des effets secondaires mal spécifiés... et pire encore : tout à fait évitables!!

```

4  /** Maximum de deux entiers
5   * Entrées : a et b deux entiers relatifs
6   * Sortie : le maximum de a et b
7   */
8  int max(int a, int b) {
9      if (a > b) {
10         return a;
11     }
12     else {
13         return b;
14     }
15 }
16
17 /** Maximum d'un tableau
18 * Entrées : T un tableau de L entiers
19 *           L un entier positif
20 * Sorties : le maximum de T
21 * Eff. Sec. : euh?...
22 */
23 int apres_moi_le_deluge(int L, int T[]) {
24     int i = 1;
25     while (i < L) {
26         T[i] = max(T[i-1], T[i]);
27         i = i+1;
28     }
29     // à la fin, T[l-1] = max(T)
30     return T[L-1];
31 }

```

Exercice. Parmi les 4 fonctions `maxi_vX` ci-dessous, lesquelles respectent leur spécification?

```

34 /** Maximum d'un tableau
35 * Entrées : n un entier >= 1
36 *           T un tableau de n
37 *           ↪ entiers
38 * Sortie : un élément maximal
39 *           ↪ de T
40 */
41 int maxi_v0(int n, int T[]) {
42     int sortie = T[0];
43     int i = 0;
44     while (i < n) {
45         if (T[i] > sortie) {
46             sortie = T[i];
47             i = i+1;
48         }
49     }
50     return sortie;
51 }

```

```

51 /** Maximum d'un tableau
52 * Entrées : n un entier >= 1
53 *           T un tableau de n
54 *           ↪ entiers
55 * Sortie : un élément maximal
56 *           ↪ de T
57 */
58 int maxi_v1(int n, int T[]) {
59     int sortie = 666;
60     int i = 0;
61     while (i < n) {
62         sortie = max(sortie, T[i]);
63         i = i+1;
64     }
65     return sortie;
66 }

```

```

66  /** Maximum d'un tableau
67  * Entrées : n un entier >= 1
68  *          T un tableau de n
69  *          ↪ entiers
70  * Sortie : un élément maximal
71  *          ↪ de T
72  */
73  int maxi_v2(int n, int T[]) {
74      int sortie = T[0];
75      for (int i = 1; i < n; i =
76          ↪ i+1) {
77          sortie = max(sortie, T[i]);
78      }
79      return sortie;
80  }

```

```

79  /** Maximum d'un tableau
80  * Entrées : n un entier >= 1
81  *          T un tableau de n
82  *          ↪ entiers
83  * Sortie : un élément maximal
84  *          ↪ de T
85  */
86  int maxi_v3(int n, int T[]) {
87      int sortie = T[0];
88      int i = 0;
89      while (i < n) {
90          sortie = max(sortie, T[i]);
91          i = i+1;
92      }
93      return sortie;
94  }

```

1 Terminaison

Convention 2 (Notation prime).

Lorsque l'on analyse une boucle, si x désigne une valeur au début d'une itération, on notera x' la valeur associée en fin d'itération (c'est à dire juste avant le début de l'itération suivante). De même lorsque l'on analyse une fonction récursive : le prime désigne la valeur au début de l'appel récursif suivant.

Cette convention est personnelle, et même si elle est partagée par plusieurs de mes collègues, je vous recommande très chaudement de la rappeler en début d'une copie de concours avant de l'utiliser.

1.0 Variants et preuves de terminaison

Définition 3 (Terminaison).

On dit qu'une fonction termine si pour toutes entrées vérifiant les pré-conditions de la fonction, la suite d'instruction exécutée par la fonction lors d'un appel sur ces entrées est finie.

Remarque. En pratique, on aimerait aussi que la suite d'instruction soit finie même si, par erreur, on ne remplit pas les pré-conditions...

Définition 4 (Variant - semestre 1).

Un **variant de boucle** (TantQue ou Pour) est une quantité :

- entière
- minorée (resp. majorée) tant que l'on ne quitte pas la boucle.
- strictement décroissante (resp. strictement croissante) d'une itération sur l'autre.

Remarque. Le terme quantité est volontairement général : il peut s'agir de la valeur d'une variable, de la différence de deux variables, ou de toute autre opération savante impliquant au moins une variable.

Exemple.

```

103 int somme(int n) {
104     int s = 0;
105     int i = 0;
106     while (i < n+1) {
107         s = s+i;
108         i = i+1;
109     }
110     return s;
111 }

```



Dans la boucle `while` de fonction `somme` ci-contre, la quantité i est un variant. En effet, elle est :

- entière car i est un entier.
- strictement croissante car si l'on note i' la valeur de i à la fin d'une itération, on a $i' = i + 1$.
- majorée car d'après la condition de boucle, $i < n + 1$ (où n ne varie jamais).

Théorème 5 (Preuves de terminaison).

Toute boucle qui admet un variant termine.

Toute fonction dont toutes les boucles terminent *et* dont tous les appels de fonction terminent termine.

Démonstration. Traitons les deux points séparément :

- Considérons une boucle qui admet un variant V . Quitte à adapter la preuve, supposons ce variant str. décroissant minoré et notons m un tel minorant. Numérons les itérations de la boucle : $0, 1, \dots$ et notons V_0, V_1, \dots les valeurs du variant V au début de chacune de ces itérations. Comme la suite des V_i est strictement décroissante et que ses termes sont des entiers, on pour tout $i \in \mathbb{N} : V_i - 1 \geq V_{i+1}$. Il s'ensuit que si les itérations existent au moins jusqu'au rang $i = (V_0 - m) + 1$, on a $V_{(V_0 - m) + 1} \leq m - 1$. Or cela est impossible car m est un minorant : en particulier, les itérations s'arrêtent avant d'atteindre ce rang. C'est à dire que la boucle termine.
- Les seules instructions vues jusqu'à présent susceptibles de ne pas terminer sont les boucles. Si toutes les boucles de la fonction terminent, et que toutes les fonctions appelées (donc toutes *leurs*) boucles terminent, la fonction termine bien.

NB : nous verrons dans quelques mois la non-terminaison des appels récursifs... mais c'est un cas particulier de la non-terminaison d'un appel de fonction. L'énoncé du théorème est donc valide.

□

Exemple. La fonction `somme` de l'exemple précédent termine.

Exemple.

```

128 int sup_log_2(int x) {
129     int n = 0;
130     int p = 1;
131     while (p < x) {
132         p = 2*p;
133         n = n+1;
134     }
135     return n;
136 }

```



Dans la boucle `while` de fonction `sup_log_2` ci-contre, la quantité $V = x - p$ est un variant. En effet, elle est :

- entière car x et p sont des entiers.
- strictement décroissante. En effet, comme $p > 0$ (il l'est au début de la première itération et n'est ensuite que multiplié par 2), on a $p' > p$. Or, $x' = x$, donc $V' < V$.
- minorée car d'après la condition de boucle, $0 < x - p$.

Donc la boucle termine. Comme la fonction n'a pas d'autres boucles ni d'appels de fonctions, elle termine.

Profitions de cet exemple pour mentionner une autre façon de comprendre $\lceil \log_2 x \rceil$: c'est le nombre de fois qu'il faut diviser x par 2 pour atteindre 1 ou moins. C'est donc réciproquement l'exposant de la plus petite puissance de 2 qui atteint x ou plus.

Remarque.

- Étant donné une fonction, il peut parfois être impossible de prouver qu'elle termine et impossible de prouver qu'elle ne termine pas. On termes **savants** de MPI, on dit que « le problème de l'arrêt n'est pas décidable ».
- Ce n'est pas un problème qui se pose à notre modeste niveau.
- Les preuves de variant vous rappellent peut-être les récurrences mathématiques : il y a un lien profond entre les deux que nous apercevrons plus en détails au second semestre.

1.1 Cas particulier des boucles Pour

Pour étudier la terminaison d'une boucle Pour, on la dépie ainsi :

<pre> 1 pour <i>i</i> allant de 0 inclu à <i>n</i> exclu faire 2 corpsDeLaBoucle </pre>	\longrightarrow	<pre> 1 <i>i</i> \leftarrow 0 2 tant que <i>i</i> < <i>n</i> faire 3 corpsDeLaBoucle 4 <i>i</i> \leftarrow <i>i</i> + 1 </pre>
---	-------------------	---

C'est à dire que l'on se ramène à la boucle `while` dont le `for` est un raccourci. On fait bien attention au fait que :

- Le compteur de boucle (*i* ci-dessus) est initialisé juste avant la première itération.
- La mise à jour du compteur a lieu à la toute fin d'une itération.

Remarque.

- Parfois, la mise à jour du compteur n'est pas une simple incrémentation de 1. Ça ne change rien, on procède pareil.
 - Dans certains langages (dont le C), il est possible que le corps de la boucle Pour modifie le compteur. C'est une mauvaise pratique qu'il ne faut pas faire : il faut utiliser un `while` dans ce cas !
 - Je parlerais de « **vraie boucle Pour** » pour désigner une boucle où le compteur n'est pas modifié par le corps, où sa mise à jour est une incrémentation d'une quantité constante (pas forcément 1), et où le compteur est borné par une quantité fixée (cohérente avec la monotonie du compteur).
- Ce n'est pas un terme canonique, et je vous conseille de le redéfinir si vous souhaitez l'utiliser.

Proposition 6 (Terminaison des vraies Pour).

Une « vraie boucle Pour » admet toujours un variant. En particulier, elle termine.

Démonstration. D'après la définition de vraie boucle Pour, le compteur est un variant immédiat. □

Exemple.

```

113 int sommebis(int n) {
114     int s = 0;
115     for (int i = 0; i < n+1;
116         i = i+1) {
116         s = s+i;
117     }
118     return s;
119 }

```



La boucle pour de fonction `sommebis` ci-contre est une « vraie boucle Pour », c'est à dire que son compteur *i* est entier, strictement croissante (car $i' = i + 1$) et borné par $n + 1$ constant. La boucle termine donc.

Puisqu'il n'y a pas d'autres boucles ni d'appels de fonctions, il s'ensuit que `sommebis` termine.

1.2 Fonctions récursives

Pour prouver la terminaison d'une fonction récursive, on doit montrer que la suite d'appels récursifs termine. Pour cela, on utilise un **variant d'appels récursifs** : c'est comme un variant, sauf qu'on demande la stricte monotonie non pas d'une itération à l'autre mais d'un appel à l'autre. La minoration correspond en général au cas de base de la fonction récursive.

Exemple.

```

173  /** Renvoie x^n
174   * Entrée : x un entier
175   *         n entier >= 0
176   * Sortie : x^n
177   */
178  int exp_rap(int x, int n) {
179      assert(n >= 0);
180
181      /* Cas de base */
182      if (n == 0) { return 1; }
183      else if (n == 1) { return x; }
184
185      /* Sinon, cas récursifs */
186      int x_puiss_demi = exp_rap(x, n/2);
187      if (n % 2 == 0) {
188          return x_puiss_demi * x_puiss_demi;
189      } else {
190          return x * x_puiss_demi * x_puiss_demi;
191      }
192  }
```

Prouvons que cette fonction récursive termine. Commençons par remarquer qu'elle ne contient ni boucles ni appels à une autre fonction : il suffit de montrer que la suite d'appels récursifs termine. Pour cela, prouvons que la quantité n est un variant d'appels récursifs :

- n est un entier (ligne 179).
- n est minoré par 0 d'après les pré-conditions. Notons que si n atteint 0 ou 1 alors la fonction termine aux lignes 183-184.
- n décroît strictement d'un appel au suivant. d'après le code et le point précédent la fonction peut s'appeler récursivement si $n > 2$. Dans ce cas, elle s'appelle récursivement avec l'argument n' qui vaut $\frac{n}{2}$. On a bien $n' < n$.

Comme la suite d'appels récursifs admet un variant, elle termine, et d'après l'analyse initiale la fonction `exp_rap` termine.

Remarque.

- Dans cet exemple, le variant est un argument. Mais cela peut aussi être la différence de plusieurs arguments (par exemple pour une dichotomie) ou le nombre d'éléments d'une liste.
- Au second semestre, nous verrons les *inductions structurelles* qui permettent souvent de prouver une terminaison récursive de manière plus proche du code, et donc plus simple à comprendre.
- Il est souvent pertinent de faire la minoration avant la stricte décroissance. Par exemple, dans cette preuve, cela m'a permis d'utiliser implicitement le fait que $n \geq 0$ ce qui était nécessaire pour conclure que $n' < n$.
- S'il y a plusieurs appels récursifs, il faut prouver la terminaison de chacun d'entre eux et donc prouver la stricte monotonie pour chacun d'entre eux.

2 Correction

2.0 Correction partielle

Définition 7 (Correction partielle).

On dit qu'une fonction est **partiellement correcte** si pour toutes entrées vérifiant les pré-conditions et sur lesquelles la fonction termine, le résultat de l'appel de la fonction vérifie les post-conditions.

Remarque.

- C'est équivalent à « pour toutes entrées vérifiant les pré-conditions, soit l'appel de la fonction ne termine pas soit son résultat vérifie les post-conditions ».
- Comme pour la terminaison, on voudrait que la fonction soit bien élevée même lorsque ses pré-conditions ne sont pas remplies : le mieux est que dans ce cas elle termine rapidement en renvoyant un message d'erreur (et n'ait pas d'effets secondaires!!).

2.0.0 Sauts conditionnels

Pour prouver la correction partielle d'une fonction qui ne contient que des déclarations/modifications de variables et des sauts conditionnels, il suffit de faire une disjonction de cas sur les `if-then-else`.

Exemple. Prouvons la correction partielle de la fonction ci-dessous :

```

4  /** Maximum de deux entiers
5   * Entrées : a et b deux entiers relatifs
6   * Sortie : le maximum de a et b
7   */
8  int max(int a, int b) {
9      if (a > b) {
10         return a;
11     }
12     else {
13         return b;
14     }
15 }
```



Notons tout d'abord que comme annoncé, la fonction n'a pas d'effets secondaires (les deux arguments sont passés par valeur). Pour prouver la post-condition de la sortie, procédons comme le code (ligne 9) par disjonction de cas :

- Si $a > b$, on entre dans le `then` ligne 10 : on renvoie a , qui est bien $\max(a, b)$ comme annoncé.
- Sinon, on entre dans le `else` (ligne 13) : on renvoie b , qui est bien $\max(a, b)$ comme annoncé.

La fonction est donc partiellement correcte.

Remarque. Il est rare que, comme dans cet exemple, les sauts conditionnels demande la majeure partie du travail de la preuve : c'est généralement un point facile à prouver sur lequel on passe rapidement pour ne pas alourdir la rédaction.

2.0.1 Invariants de boucle

Convention 8 (Somme vide).

Lorsque l'on écrit $\sum_{k=a}^b \dots$, on considère que si $a > b$ alors la somme est vide est vaut 0. De même avec le symbole produit \prod : unproduit vide vaut 1.

Définition 9 (Invariant de boucle).

Un **invariant de boucle** (TantQue ou Pour) est une propriété logique qui :

- est vraie juste avant la toute première itération de la boucle (on appelle cela l'**initialisation**).
- si elle est vraie au début d'une itération, est vraie à la fin de cette même itération (on parle de **conservation**).

Remarque.

- Un invariant, comme un variant, doit impliquer des variables de la fonction pour être utile.
- C'est en fait une récurrence ! L'avantage de ce formalisme-ci par rapport à une récurrence sur les itérations est que l'on a pas besoin d'introduire un numéro d'itération superflu. L'autre avantage est que l'on compare le début d'une itération à la fin de celle-ci, là où une récurrence comparerait au début de l'itération suivante... laquelle n'existe peut-être pas.
- Comme pour les preuves de terminaison, on utilise la notation prime pour marquer une quantité en fin d'itération.
- Pour prouver la conservation dans une boucle `while`, il est souvent utile d'utiliser le fait que l'on est encore en train d'itérer la boucle et que donc la condition de la boucle est vraie : elle est une hypothèse que l'on peut appliquer.
- Une très bonne pratique est de ne pas seulement prouver que quelque chose est un invariant, mais aussi de préciser ce qu'il prouve à la fin de la dernière itération. On nomme cette étape la **conclusion**.

Exemple.

```

98  /** Somme des n premiers entiers.
99  * Entrées : n >= 0
100  * Sortie : somme des n premiers entiers de N*.
101  * Eff. Sec : aucun.
102  */
103  int somme(int n) {
104      int s = 0;
105      int i = 0;
106      while (i < n+1) {
107          s = s+i;
108          i = i+1;
109      }
110      return s;
111  }
```



Montrons que $I: \ll s = \sum_{k=0}^{i-1} k \gg$ est un invariant.

- Initialisation : avant la première itération de la boucle, on a $s = 0$. On a également $i - 1 < 0$, d'où la somme de I est vide et vaut donc 0. On a bien $0 = 0$: l'invariant est initialisé.
- Conservation : supposons l'invariant I vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que $I' : \ll s' = \sum_{k=0}^{i'-1} k \gg$.
On a :

$$\begin{cases} s' = s + i & \text{(ligne 108)} \\ i' = i + 1 & \text{(ligne 109)} \end{cases}$$

Or d'après I , on a $s = \sum_{k=0}^{i-1} k$. Donc :

$$\begin{aligned}
s' &= s + i \\
&= \left(\sum_{k=0}^{i-1} k \right) + i && \text{d'après } I \\
&= \sum_{k=0}^i k \\
&= \sum_{k=0}^{i'-1} k && \text{car } i' = i + 1
\end{aligned}$$

On a prouvé que I' est vrai : l'invariant se conserve.

- Conclusion : On a prouvé que I est un invariant. En particulier, comme à la fin de la dernière itération on a $i' = n + 1$, on a en sortie de boucle :

$$\begin{aligned}
s &= \sum_{k=0}^{n+1-1} k \\
&= \sum_{k=0}^n k
\end{aligned}$$

Remarque.

- J'ai volontairement beaucoup détaillé les étapes de calcul de cette preuve, afin de limiter les difficultés calculatoires. On aurait pu aller plus vite.
- Toutefois, je vous demande de ne pas aller plus vite que moi sur la phrase d'introduction de la conservation : « *supposons l'invariant [...] vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que [...]'* ». En effet, je veux que vous ayez chacun des éléments de cette phrase sous les yeux avant de vous lancer dans la conservation en elle-même.²

Proposition 10.

Pour prouver la correction partielle d'une fonction qui contient des boucles, on fait appel à des invariants de boucle afin de prouver qu'une certaine propriété est vraie au moment de la sortie de la boucle.

Autrement dit, les invariants jouent le rôle d'un lemme : ils *ne* prouvent *pas* la correction partielle, mais on espère pouvoir utiliser leur résultat pour en déduire la correction partielle.

Exemple. Dans l'exemple précédent, l'invariant choisi prouve qu'en sortie de boucle, on a $s = \sum_{k=0}^n k$. Comme s n'est pas modifié entre la sortie de la boucle et le `return s`, on peut conclure que la sortie renvoyée vérifie la condition annoncée. De plus, il n'y a pas d'effets secondaires (une seule entrée, passée par valeur) comme annoncés : les post-conditions sont vérifiées, donc la fonction est partiellement correcte.

Exemple. Prouvons la correction partielle de la fonction ci-dessous. On utilisera la propriété suivante pour qualifier $\lceil \log_2 x \rceil$ (avec $x \geq 1$) : c'est l'unique entier ℓ tel que $2^{\ell-1} < x \leq 2^\ell$.

2. Comme, peut-être, lors de votre apprentissage des récurrences en Terminale.

```

122  /** Partie entière supérieure de log2(x)
123   * Entrées : x entier >= 1
124   * Sortie : la partie entière supérieure du
125   *          logarithme en base 2 de x
126   * Eff. Sec : aucun.
127   */
128  int sup_log_2(int x) {
129      int n = 0;
130      int p = 1;
131      while (p < x) {
132          p = 2*p;
133          n = n+1;
134      }
135      return n;
136  }

```

On veut prouver l'encadrement voulu pour la ligne 136, c'est à dire en fin de boucle while.

Puisque l'on a quitté la boucle, on a alors $p \geq x$. En déroulant les étapes à la main, on se rend compte que I : « $p = 2^n$ » est un invariant³ : nous allons donc le prouver. Nous aurons aussi besoin de l'invariant J : « $\frac{p}{2} < x$ » qui donnera l'autre moitié de l'encadrement voulu⁴.

- Initialisation : avant la première itération, on a $p = 1$ et $n = 0$, or $1 = 2^0$ donc J est bien initialisé. De plus, d'après les pré-conditions on a $x \geq 1$. Comme $\frac{p}{2} = \frac{1}{2} < 1$, on a bien l'initialisation de I .
- Conservation : supposons les invariants I et J vrais au début d'une itération quelconque et montrons-les vrais à la fin de cette itération, c'est à dire montrons que I' : « $p = 2^{n'}$ » et que J' : « $\frac{p'}{2} < x$ ».

On a :

$$\begin{cases} p' = 2p & \text{(ligne 133)} \\ n' = n + 1 & \text{(ligne 134)} \end{cases}$$

En appliquant I dans la définition de p' , on obtient :

$$p' = 2 \times 2^n = 2^{n+1} = 2^{n'}$$

C'est à dire I' . Pour prouver J' , utilisons le fait qu'au début de chaque itération on a $p < x$. Donc d'après la définition de p' :

$$\frac{p'}{2} = p < x$$

C'est à dire J' . Les deux invariants se conservent.

- Conclusion : On a prouvé que I et J sont des invariants. En particulier, en sortie de boucle on a $p = 2^n$ et $x > \frac{p}{2} = 2^{n-1}$.

L'évaluation de J en sortie de boucle nous donne la minoration de l'encadrement attendu. Pour obtenir la majoration, remarquons qu'on quitte la boucle car $p \geq x$, c'est à dire d'après I car $2^n \geq x$.

On a ainsi prouvé l'encadrement demandé : la fonction renvoie bien $\lceil \log_2 x \rceil$. Remarquons enfin qu'elle n'a aucun effets secondaires comme annoncé : la fonction est donc partiellement correcte.

Remarque. Les invariants peuvent aussi servir à prouver une terminaison (on prouve que la minoration/majoration du variant est un invariant) ou des non-terminaisons (on prouve que la condition d'un while est toujours fausse).

3. Qui sera très utile puisque l'on veut encadrer x par des puissances de 2 : nous avons la majoration grâce à cet invariant !

4. Celui-ci est peut-être un peu moins évident : on peut le trouver en essayant de conclure avec uniquement I , on se rend alors compte qu'il nous manque la moitié de l'encadrement mais qu'elle forme un invariant pas méchant.

2.0.2 Cas particulier des boucles Pour

Comme pour la terminaison, on déplie les boucles Pour :

<pre> 1 pour <i>i</i> allant de 0 inclu à <i>n</i> exclu faire 2 corpsDeLaBoucle </pre>	\longrightarrow	<pre> 1 <i>i</i> ← 0 2 tant que <i>i</i> < <i>n</i> faire 3 corpsDeLaBoucle 4 <i>i</i> ← <i>i</i> + 1 </pre>
---	-------------------	---

En particulier :

- Le compteur est créé juste avant la première itération, et l'initialisation des invariants se fait donc avec cette valeur.
- Le compteur est mis à jour juste avant la fin de l'itération, et donc sa valeur en fin d'itération (i') n'est plus celle en début (i).
- On quitte la boucle car le compteur dépasse : en sortie de boucle, le compteur contient la valeur mise à jour du compteur (dans l'exemple ci-dessus, i vaut n en sortie de boucle).

Attention, cette valeur mise à jour n'est pas forcément la borne du `for` : pensez par exemple à `for (int i = 0; i < 3; i = i+2)` .

On se contente généralement d'affirmer cette valeur en sortie de boucle au lieu de la prouver.

Exemple. Prouvons la correction partielle de la fonction ci-dessous :

```

147 int exp_simple(int a, int n) {
148     int p = 1;
149     for (int i = 0; i < n; i = i+1) {
150         p = a*p;
151     }
152     return p;
153 }
```



On veut prouver qu'en sortie de boucle, $p = a^n$. Pour cela, montrons que $I : \langle p = a^i \rangle$ est un invariant.

- Initialisation : Juste avant la première itération, on a $p = 1$ et $i = 0$. On a bien $1 = a^0$, d'où l'initialisation.
- Hérédité : supposons l'invariant I vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que $I' : \langle p = a^{i'} \rangle$.

Or :

$$\begin{aligned}
 p' &= a \times p && \text{d'après ligne 151} \\
 &= a \times a^i && \text{d'après } I \\
 &= a^{i+1} \\
 &= a^{i'} && \text{d'après ligne 150}
 \end{aligned}$$

C'est à dire I' : l'invariant est conservé.

- Conclusion : On a prouvé que I est un invariant. En particulier, comme en sortie de boucle $i = n$, on a alors :

$$p = a^n$$

C'est exactement la condition attendue sur p . De plus, la fonction n'a pas d'effets secondaires, comme annoncé. Elle est donc partiellement correcte.

2.1 Fonctions récursives

Pour les fonctions récursives, on prouve les invariants par... récurrence ! Il s'agit de prouver que la spécification est vérifiée, en supposant que l'appel récursif la vérifie déjà (hérédité). Il faut bien sûr également prouver que les cas de base la vérifient (initialisation).

Exemple.

```

173  /** Renvoie x^n
174   * Entrée : x un entier
175   *          n entier >= 0
176   * Sortie : x^n
177   */
178  int exp_rap(int x, int n) {
179      assert(n >= 0);
180
181      /* Cas de base */
182      if (n == 0) { return 1; }
183      else if (n == 1) { return x; }
184
185      /* Sinon, cas récursifs */
186      int x_puiss_demi = exp_rap(x, n/2);
187      if (n % 2 == 0) {
188          return x_puiss_demi * x_puiss_demi;
189      } else {
190          return x * x_puiss_demi * x_puiss_demi;
191      }
192  }

```

Prouvons que cette fonction est partiellement correcte. Commençons par noter qu'elle n'a pas d'effets secondaires, comme demandé.

Montrons par récurrence forte sur $n \in \mathbb{N}$ la propriété H_n : « pour tout x entier, `exp_rap(x, n)` renvoie x^n ».

- **Initialisation** : si $n = 0$ (resp. si $n = 1$), la fonction renvoie 1 qui vaut bien x^0 (resp. x qui vaut bien x^1). D'où l'initialisation.
- **Hérédité** : soit n entier strictement supérieur à 1 tel que la propriété soit vraie jusqu'au rang $n-1$. Montrons-la vraie au rang n , c'est à dire montrons que `exp_rap(x, n)` renvoie x^n .

Comme $n > 1$, l'exécution ne rentre pas dans les if-else (ni dans le assert) et on va donc en ligne 187. Par hypothèse de récurrence, `x_puiss_demi` contient $x^{\lfloor \frac{n}{2} \rfloor}$.

Or, d'après le cours de maths :

- Si n est pair, on a $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$. Donc :

$$x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor} = x^n$$

- Sinon, on a $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$. Donc :

$$x \cdot x^{\lfloor \frac{n}{2} \rfloor} \cdot x^{\lfloor \frac{n}{2} \rfloor} = x \cdot x^{n-1} = x^n$$

Or, cette disjonction de cas et ces calculs sont exactement les liens 188-192 du code : la fonction renvoie donc bien dans tous les cas x^n . D'où l'hérédité.

- **Conclusion** : On a prouvé par récurrence forte que pour tout n , pour tout x , la fonction renvoie bien le résultat annoncé.

Les post-conditions sont donc vérifiées : la fonction est partiellement correcte.

Remarque. S'il y a plusieurs appels récursifs, on doit utiliser l'hypothèse de récurrence sur chacun d'entre eux.

2.2 Correction totale

Définition 11 (Correction totale).

On dit qu'une fonction est **totale**ment correcte lorsqu'elle termine et qu'elle est partiellement correcte.

Exemple.

- La fonction `somme` des exemples précédents est totalement correcte : on a prouvé sa terminaison page 19 et correction aux pages 23 et 24.
- La fonction `sup_log_2` aussi : terminaison page 19 et correction page 24.
- La fonction `exp_rap` aussi : terminaison page 21 et correction page 27.
- La fonction `max` termine car elle ne contient ni boucle ni appels de fonction, et on a prouvé sa correction page 22 : elle est aussi totalement correcte.
- La fonction `patience` ci-dessous est partiellement correcte mais ne termine pas, donc n'est pas totalement correcte :

```

156  /** Renvoie true
157  * Entrée : rien
158  * Sortie : true
159  */
160  bool patience(void) {
161      while (true) {
162          int sert_a_rien = 0;
163      }
164      return true;
165  }
```

Remarque. La correction totale (que ce soit la terminaison ou la correction partielle) d'une fonction dépend de la correction totale des fonctions qu'elle appelle ! Il faut donc commencer par les prouver (ou les admettre s'il s'agit de fonctions fournies par des bibliothèques que l'on a pas codées ou par l'énoncé.)

Exercice. Prouver que la fonction `maxi_v3` ci-dessous est totalement correcte :

```

79  /** Maximum d'un tableau
80  * Entrées : n un entier >= 1
81  *          T un tableau de n entiers
82  * Sortie : un élément maximal de T
83  */
84  int maxi_v3(int n, int T[]) {
85      int sortie = T[0];
86      int i = 0;
87      while (i < n) {
88          sortie = max(sortie, T[i]);
89          i = i+1;
90      }
91      return sortie;
92  }
```

Chapitre 2

MÉMOIRE

Notions	Commentaires
Représentation des flottants. Problèmes de précision des calculs flottants.	On illustre l'impact de la représentation par des exemples de divergence entre le calcul théorique d'un algorithme et les valeurs calculées par un programme. Les comparaisons entre flottants prennent en compte la précision.


Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Utilisation de la pile et du tas par un programme compilé.	On présente l'allocation des variables globales, le bloc d'activation d'un appel.
Notion de portée syntaxique et durée de vie d'une variable. Allocation des variables locales et paramètres sur la pile.	On indique la répartition selon la nature des variables : globales, locales, paramètres.
Allocation dynamique.	On présente les notions en lien avec le langage C : malloc et free [...]

Extrait de la section 5.1 du programme officiel de MP2I : « Gestion de la mémoire d'un programme ».

SOMMAIRE

0. Représentation des types de base	30
0. Notion de données	30
1. Booléens	30
2. Différents types d'entiers	31
<i>Entiers non-signés (p. 31). Caractères (p. 33). Entiers signés (p. 33). Dépassements de capacité (p. 35).</i>	
3. Nombres à virgule (flottante)	36
<i>Notation mantisse-exposant pour l'écriture scientifique (p. 36). Erreurs d'approximation et autres limites (p. 37).</i>	
4. Tableaux	38
5. Chaînes de caractères	39
1. Compléments très brefs de programmation.....	40
0. Notion d'adresse mémoire	40
1. Portée syntaxique et durée de vie	40
<i>Définition et premiers exemples (p. 40). Exemples avancés (p. 43).</i>	
2. Organisation de la mémoire d'un programme	44
0. Bloc d'activation	44
1. Organisation de la mémoire virtuelle d'un programme	45
<i>Pile mémoire (p. 46). Tas mémoire (p. 46). À savoir faire (p. 47).</i>	

 Ce chapitre, surtout sa dernière section, est rempli de simplifications.

0 Représentation des types de base

Dans cette section, nous allons voir comment les données (entiers, nombres à virgule, etc) sont représentées en mémoire. Nous n'allons *pas* voir où ces représentations sont rangées en mémoire.

Ce qui est vu dans cette section est valable en C et en OCaml.

0.0 Notion de données

Définition 1 (Donnée informatique).

Une donnée informatique est un élément fini d'information que l'on peut lire, stocker transmettre.

Définition 2.

- L'**unité élémentaire** de stockage des données est le **chiffre binaire (bit en anglais)**. Il peut prendre deux valeurs : 0 et 1.
- Un groupe de 8 bits est appelé un **octet**. Les ordinateurs modernes ne stockent pas les bits uns par uns dans la mémoire mais des octets : on dit que c'est le **plus petit adressable (byte en anglais)**.
- Les processeurs modernes, pour aller plus vite, ne font pas leurs calculs sur des octets mais sur un des groupes plus gros. On parle de **mot machine**, et ils font aujourd'hui souvent 8 octets / 64 bits

Remarque. Le terme du « plus petit adressable » provient du fait que puisque les bits sont stockés 8 par 8, on ne peut pas accéder à un des bits de l'octet sans avoir lu en même temps les autres. C'est une question de câblage électronique : les « fils » ne ciblent pas chaque bit individuellement mais des groupes de 8 bits adjacents.

Remarque. Des (suites d')octets en eux-mêmes ne veulent rien dire. Il faut connaître la façon de les déchiffrer. C'est le rôle d'un **type de données** : c'est une description de comment encoder et décoder des données d'une certaine nature. Vous connaissez déjà des types : `int` , `bool` .

0.1 Booléens

Le type booléen permet de stocker les deux valeurs logiques : Vrai (« true ») et Faux (« false »).

Proposition 3 (Encodage des booléens).

Il suffit en théorie d'1 bit pour encoder un booléen.

En C, on utilise un plus petit adressable (1 octet).

En OCaml, un mot machine (8 octets).

Remarque.

- En OCaml, *tout* est manipulé à travers un mot machine, dont les booléens.
- En C comme en OCaml, il existe des astuces pour stocker un booléen dans chacun des bits d'un octet / mot machine.

Définition 4 (Opérateurs logiques).

Voici les opérations usuelles des booléens :

x	false	true
Non(x)	true	false

(a) NON(x)

x \ y	false	true
false	false	false
true	false	true

(b) ET(x,y)

x \ y	false	true
false	false	true
true	true	true

(c) OU(x,y)

x \ y	false	true
false	false	true
true	true	false

(d) XOR(x,y)

FIGURE 2.1 – Tables (à double entrée sauf pour la négation) des opérations logiques usuelles.

Pour permettre d'alléger les écritures de formules logiques, on se donne des règles de priorité : NON est plus prioritaire que ET qui est plus prioritaire que XOR qui est plus prioritaire que OU.

Exemple. « NON false ET false OU true » s'évalue à true, car cette expression se parenthèse ainsi d'après les règles : « ((NON false) ET false) OU true »

Exercice. Évaluez NON (NON false ET NON true OU NON false ET true).

0.2 Différents types d'entiers

Il faut distinguer deux grandes familles de types d'entiers :

- les **entiers non-signés** sont des types qui permettent de manipuler uniquement des entiers positifs.
- les **entiers signés** sont des types qui permettent de manipuler des entiers relatifs.

0.2.0 Entiers non-signés

Théorème 5 (Écrire en base b).

Soit b un entier supérieur ou égal à 2 appelé **base**. Pour tout $x \in \mathbb{N}$, il existe une unique décomposition de x de la forme :

$$x = \sum_{i=0}^{N-1} c_i \cdot b^i$$

qui vérifie :

- pour tout i , $c_i \in \llbracket 0; b \rrbracket$. Les c_i sont appelés les **chiffres**.
- $a_{N-1} \neq 0$.

On appelle la donnée des c_i l'**écriture en base b** de x . Pour indiquer la base, on note généralement : ${}_b c_{N-1} c_{N-2} \dots c_1 c_0$

On dit que c_{N-1} est le **chiffre** de poids fort et que c_0 est le **chiffre de poids faible**.

Convention 6 (Base 10 implicite).

Lorsque l'on ne précise pas la base, on utilise implicitement la base 10.

Remarque.

- On parle de **binaire** pour la base 2, **décimal** pour la base 10, **hexadécimal** pour la base 16.
- En base 16, les chiffres peuvent valoir plus que 9 : on note a le chiffre correspondant au nombre décimal 10, b pour 12, ..., f pour 16.
- L'écriture $\overline{c_{N-1}c_{N-2}\dots c_1c_0}^b$ existe aussi.
- Vous êtes habitué-es à l'écriture en base 10 : $2048 = 2 \cdot 10^3 + 0 \cdot 10^2 + 4 \cdot 10^1 + 8 \cdot 10^0$.

Exemple.

$$\begin{aligned} {}^2\overline{1100\ 0010} &= 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &= 128 + 64 + 2 \\ &= 194 \end{aligned}$$

Définition 7 (Représentation des entiers non-signés).

Pour représenter un entier non-signé, on stocke son écriture en base b dans un ou plusieurs octets. le bit de poids faible est tout à droite. Au besoin, on complète à gauche par des 0 :

1	1	0	0	0	0	1	0
---	---	---	---	---	---	---	---

Le nombre d'octets utilisés peut être fixé par le type (4 octets pour `unsigned int` en C, 8 octets en OCaml).

Exemple. Voici l'entier 3452 représenté sur deux octets :

0	0	0	0	1	1	0	1	0	1	1	1	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Remarque.

- Les 32 bits du type `unsigned int` C peuvent dépendre de la machine, mais 32 est la valeur la plus commune.
- En incluant `stdint.h`, on a accès en C à des types dont le nombre de bits est garanti : `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`.

Proposition 8 (Capacité des types non-signés).

Sur k bits, on peut représenter de cette façon exactement les entiers de $\llbracket 0; 2^k \rrbracket$.

⚠ Conséquence importante : on ne peut pas représenter *tous* les entiers positifs en C ni en OCaml!

Remarque. C'est long d'écrire en binaire. Pour raccourcir, on utilise souvent la base 16. En effet, 4 chiffres consécutifs en base 4 correspondent à 1 chiffre en base 16 : la traduction est donc facile¹. Convertissons par exemple ${}^2\overline{0000\ 1101\ 0111\ 1100}$

$$\begin{array}{cccc} \overline{0000} & \overline{1101} & \overline{0111} & \overline{1100} \\ \text{0} & \text{d} & \text{7} & \text{c} \end{array}$$

$$\text{Donc } {}^2\overline{0000\ 1101\ 0111\ 1100} = {}^{16}\overline{0d7c}.$$

Exercice. Réciproquement, convertissez ${}^{16}\overline{9c}$ en binaire.

1. Beaucoup plus qu'entre la base 10 et la base 2 (ou la base 16).

0.2.1 Caractères

Le type des **caractères** permet de stocker des « lettres » : une lettre de l'alphabet, un chiffre décimal, une espace, etc. Dans l'idée, il permet de stocker « une touche du clavier ».

Attention, je parle bien de lettres uniques et non de suites de lettres : ce second cas est plus compliqué, et on parle pour lui de chaînes de caractères (strings).

Définition 9 (Représentation des caractères).

Pour représenter un caractère, on le fait correspondre à un entier non-signé et on encode ensuite cet entier non-signé.

La façon la plus commune actuellement de réaliser cette correspondance est la table ASCII. Elle donne une correspondance entre 128 caractères et les 128 entiers 7-bits.

En C, le type `char` est le type des correspondances des caractères ASCII. C'est un type d'entiers non-signés, avec lequel on peut faire des calculs.

Remarque.

- ASCII signifie **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange. En conséquence, les caractères qui n'intéressent pas les américains (caractères accentués par exemple) n'apparaissent pas dedans.
- Aujourd'hui, on utilise beaucoup plus de caractères que l'ASCII : nos caractères sont maintenant encodés par l'Unicode (UTF-8), qui donne des correspondances entre *beaucoup* plus de caractères et plus d'octets.
- Dans la table ASCII, les lettres de l'alphabet sont les unes après les autres (les majuscules vont de 'A'=65 à 'Z'=90, et les minuscules de 'a'=97 à 'z'=122).

Exemple. Comme les `char` sont des entiers et que les lettres se suivent, on peut écrire :

```
1 char lettre = 'a';
2 while (lettre <= 'z') {
3     fairedetrucs;
4
5     // passer à la lettre suivante
6     lettre = lettre +1;
7 }
```



0.2.2 Entiers signés

Pour représenter des entiers signés, on utilise le **complément à 2**. Pour simplifier les explications, commençons par présenter le **complément à 10**, et je noterai entre guillemets les codes (la façon de représenter les nombres) :

- 0) On fixe le nombre de chiffres que la représentation utilisera. Dans cet exemple, je travaillerai avec 3 chiffres décimaux.
- 1) La première moitié des codes possibles encodent les positifs de manière transparente :

Code	Entier représenté	Code	Entier représenté
"000"	0
"001"	1	"497"	497
"002"	2	"498"	498
...	...	"499"	499

- 2) La seconde moitié des codes encode les nombres négatifs. Pour encoder $-|x|$, on calcule $10^3 - |x|$ et on utilise le résultat comme code :

Code	Entier représenté	Code	Entier représenté
"500"	-500
"501"	-499	"997"	-3
"503"	-498	"998"	-2
...	...	"999"	-1

3) Fin.

Remarque.

- En complément à 10 sur 3 chiffres, on peut donc représenter $\llbracket -500; 500 \rrbracket$. Plus généralement, en complément à 10 sur k chiffres on peut représenter $\llbracket -10^k/2; 10^k/2 \rrbracket$.
- Le premier chiffre n'est pas un chiffre de signe au sens où il ne stocke pas le signe. Par contre, on peut en déduire le signe.
- Si l'on dispose d'un code "xyz", pour savoir s'il encode un entier positif ou strictement négatif il suffit de savoir si ce code fait partie de la première moitié des codes ou de la seconde... et donc il suffit de regarder le premier chiffre !

Et le complément à 2 ? C'est pareil mais en base 2 ! Par exemple sur 1 octet, on représente :

Code	Entier représenté	Code	Entier représenté
"0000 0000"	0	"1000 0000"	-128
"0000 0001"	1	"1000 0001"	-127
...
"0111 1110"	126	"1111 1110"	-2
"0111 1111"	127	"1111 1111"	-1

On représente souvent les compléments sous forme visuelle :

(a) Complément à 10 sur 3 chiffres

(b) Complément à 2 sur 8 chiffres

FIGURE 2.2 – Représentation visuelle des compléments

0.2.3 Dépassements de capacité

Sur des types utilisant un nombre fixé d'octets (comme en C ou en OCaml²), on ne peut donc utiliser qu'un nombre fini d'octets. Mais que se passe-t-il si, par exemple, on additionne deux nombres du type et que le résultat de l'addition est trop grand pour le type ?

Par exemple (avec des `uint8_t`), on peut poser l'addition :

$$\begin{array}{r} 1101 \quad 0011 \\ + \quad 0011 \quad 0101 \\ \hline 1 \quad 0100 \quad 1000 \end{array}$$

Le résultat de cette addition ne tient pas sur un octet. Sur la plupart des machines et des langages (dont C et OCaml), le résultat serait simplement tronqué en « enlevant ce qui déborde » et on obtiendrait ²0100 1000.

Définition 10 (Dépassement de capacité).

On parle de **dépassement de capacité (overflow)** lorsque le résultat attendu d'une opération sort de l'ensemble des valeurs représentables par son type.

On parle parfois de **surpassement** pour qualifier un dépassement « par le haut » et de **sous-passement** « par le bas ».

Proposition 11 (Gestion des dépassements entiers en C/OCaml).

Dans les deux langages au programme, les dépassements d'entiers sont gérés ainsi :

- types d'entiers non-signés : sur un type d'entiers k bits, les calculs sont effectués modulo 2^k . Cela revient à « ignorer ce qui déborde ».
- types d'entiers signés : les *encodages* sont calculés en « ignorant ce qui déborde ». Cela donne un caractère circulaire au type (cf TD).

NB : en C, cette façon de gérer n'est pas obligatoire. Elle est là sur la plupart des machines mais il est dangereux de faire un programme qui repose dessus.

⚠ Les dépassements de capacité sont une source **très** commune d'erreur. Il faut y prêter attention !

Exemple. Voici une liste d'exemples de bogues de dépassements de capacité³ :

- Explosion du vol 501 de la fusée Ariane 5, le 4 juin 1995 (à cause d'un écrêtage dans une conversion 64bits -> 16 bits et d'un mauvais choix dans le code de comment réagir à une erreur imprévue) : https://fr.wikipedia.org/wiki/Vol_501_d%27Ariane_5#Conclusions.
- La machine de radiothérapie Therac-25 a causé la mort de cinq patients par violent surdosage de radiation. Parmi les causes, une variable entière qui débordait et passait sous le seuil qui déclenchait des tests. Il y a beaucoup d'autres causes, dont et surtout la façon dont le logiciel a été conçu, relu, documenté, etc : <https://fr.wikipedia.org/wiki/Therac-25#Causes>.
- Bug de l'an 2038 : dans un ordinateur, la date est stockée en nombre de secondes depuis le 1er janvier 1970. Sur les machines où cette date est stockée en complément à 2 sur 32bits (comme `int` en C...), il y aura un débordement le 19 janvier 2038 à 3h14min8s. La date deviendra alors (débordement « circulaire ») le 13 décembre 1901, 20h45 et 52s. Cf https://fr.wikipedia.org/wiki/Bug_de_l%27an_2038.
Notez que la norme demande d'utiliser un entier non signé, ce qui sur 32 bits repousse le bug à 2106.
- Et beaucoup d'autres.

2. Mais pas comme en Python

3. En réalité, un bogue n'est presque jamais uniquement dû à un dépassement de capacité. Il se combine avec d'autres manquements, que ce soit dans le code (une fonction critique ne s'assure pas du respect de ses pré-conditions), dans le processus de validation du code (le code n'a pas été assez bien testé, ni assez prouvé, ni assez bien relu) et/ou dans la gestion de l'équipe de développement.

0.3 Nombres à virgule (flottante)

On veut représenter des nombres à virgule. Remarquons tout d'abord que dans tout intervalle non-vide, il y a une quantité infinie de nombres rationnels (sans parler des réels). On ne pourra donc pas représenter tous les nombres d'un intervalle, aussi "petit" soit-il.

0.3.0 Notation mantisse-exposant pour l'écriture scientifique

Commençons à nouveau par expliquer en base 10. Un nombre $x \in \mathbb{R}$ est dit **nombre décimal** s'il existe un $p \in \mathbb{N}$ tel que $x * 10^p \in \mathbb{Z}$. Autrement dit, si il a un nombre fini de chiffres après la virgule.

Exemple.

- Tous les entiers sont décimaux.
- 0.1 est décimal car $0.1 * 10^1 = 1 \in \mathbb{Z}$.
- 0.5 est décimal car $0.5 * 10^1 = 5 \in \mathbb{Z}$.
- $\frac{1}{3}$ n'est pas décimal même si il est rationnel.
- $\sqrt{2}$ n'est pas décimal même si il est algébrique (racine d'un polynome à coefficients entiers).
- π et $\exp(1)$ ne sont pas décimaux.

Définition 12 (Écriture scientifique en base 10).

Pour tout x décimal, il existe une unique écriture de x de la forme $x = \pm m * 10^e$ avec :

- $m \in [1; 10[$ (et est décimal).
- $e \in \mathbb{Z}$.

On appelle m la mantisse et e l'exposant.

On parle parfois de **virgule flottante** pour désigner le fait qu'on « déplace » la virgule lorsqu'on passe un nombre en écriture scientifique.

Exemple.

- $243 = 2.43 * 10^2$
- $0.00759 = 7.59 * 10^{-3}$

Pour représenter un nombre décimal, on peut donc donner : le signe, la mantisse et l'exposant.

Revenons à la base 2. On va appliquer exactement cette idée : écrire en écriture scientifique, et communiquer mantisse, signe, exposant.

Définition 13 (Nombre dyadique).

Un nombre réel x est dit dyadique s'il existe un $p \in \mathbb{Z}$ tel que $x * 2^p \in \mathbb{Z}$. Autrement dit, s'il a un nombre fini de chiffres après la virgule en base 2.

Exemple.

- Tous les entiers sont dyadiques.
- 0.1 n'est **pas** dyadique.⁴
- 0.5 est dyadique car c'est 2^{-1} .
- $\frac{1}{3}, \sqrt{2}, \pi, \exp(1)$ ne sont pas dyadiques.

4. Le prouver est un petit exo de maths sympa. On peut par contre prouver que si un nombre est dyadique alors il est décimal : c'est la deuxième partie d'un petit exo sympa.

Définition 14 (Écriture scientifique en base 2).

Pour tout x dyadique, il existe une unique écriture de x de la forme $x = \pm m * 2^e$ avec :

- $m \in [1; 2[$ (et est dyadique).
- $e \in \mathbb{Z}$.

On appelle m la mantisse et e l'exposant.

L'idée est alors la suivante. On va stocker des nombres dyadiques sur 64 bits, ainsi :

- Le premier bit est un bit de signe 0 pour positif, 1 pour négatif.
- Ensuite l'exposant. On le stockera sous une forme particulière qui simplifie certains calculs.
- Enfin la mantisse. Toutefois, il est inutile de stocker le bit de poids fort de la mantisse : on sait qu'il vaut 1 puisque $m \in [1; 2[$. On stocke donc uniquement la partie après la virgule de la mantisse!⁵

On obtient le format suivant (aussi appelé **norme IEEE 754**) :

Définition 15 (Représentation des nombres dyadiques).

On encode des nombres dyadiques sur 64 bits de la façon suivante :

	signe s	exposant E	mantisse M
64 bits :	1 bit	11 bits	52 bits

Qui représente : $\pm m * 2^e$ avec :

- + si $s = 0$, - sinon.
- E code un entier non-signé sur 11 bits, et $E = e + 1023$. On représente donc les puissances $e \in \llbracket -1023; 1024 \rrbracket$. 1023 est appelé la **constante d'excentrement**.
- $m = 1, M$. C'est à dire que la mantisse stockée ne stocke que ce qu'il y a après la virgule. On a donc en réalité accès à 53 chiffres significatifs.

On parle de représentation en **double précision**, car autrefois on utilisait moitié moins de bits (32). Le double correspondant en C est **double**.

Définition 16 (Valeurs réservées).

Certaines paires de valeur (M, E) sont réservées :

- $M = 0...0$ et $E = 0...0$ encodent ± 0 . Par conséquent, $\pm \overline{1, 0...0} * 2^{-1023}$ n'est pas représentable.
- $M = 0...0$ et $E = 1...1$ encodent $\pm \infty$. Par conséquent, $\pm \overline{1, 1...1} * 2^{1024}$ n'est pas représentable.
- un autre M avec $E = 1...1$ encode NaN (Not a Number).

0.3.1 Erreurs d'approximation et autres limites

Cette représentation double précision a des limites. Elles proviennent du fait que puisque la mantisse est de taille finie, on manipule des approximations⁶.

- Propagation d'erreurs : toutes les erreurs listées ci-dessous se propagent lors des calculs, comme les approximations en physique.
- Beaucoup de nombres sont approximés et non exacts, dont tous les nombres non-dyadiques. Rappelons que 0.1 n'est pas dyadique : on en manipule donc un arrondi, ce qui mène à des erreurs. Si possible, il vaut lui préférer 0.125 qui est lui exact.
- Faux négatif sur les comparaisons : parfois, des comparaisons comme « $a+b=c$ » s'évaluent à **false** alors qu'elles sont mathématiquement justes. Un exemple classique est $0.1 + 0.2 == 0.3$.⁷

5. Ça a l'air de rien, mais on gagne ainsi 1 bit dans la mantisse, c'est à dire un chiffre significatif stockable de plus.

6. Comme en physique : si on a trop peu de chiffres significatifs, le résultat est peu précis.

7. En même temps, **aucun** de ces 3 nombres n'est dyadique. Partant de là, l'égalité eut été un coup de chance incroyable.

- Faux positif sur les comparaisons : de même.
- La somme de deux nombres représentables ne l'est pas forcément. Par exemple, 2^{30} l'est, 2^{-30} l'est, mais il faut un M de 60 chiffres pour stocker leur somme. En fait, il se passe un débordement de la mantisse : un flottant déborde en perdant de la précision.
- Non associativité : on n'a pas forcément $(a+b)+c == a+(b+c)$. Par exemple $(2^{-30} + 2^{30}) - 2^{30}$ ne renvoie pas le même résultat que $2^{-30} + (2^{30} - 2^{30})$.

Corollaire 17.

Quand on compare des flottants, on le fait toujours à epsilon près, avec epsilon la précision voulue!

Par exemple, au lieu de tester $a \neq b$, on teste $|a-b| > \epsilon$.

0.4 Tableaux

Définition 18 (Représentation des tableaux).

Soit τ un type qui se représente sur r bits. Un tableau T de longueur L dont les cases sont de type τ est représenté en mémoire comme :

Représentation de $T[0]$	Représentation de $T[1]$...	Représentation de $T[L-1]$
-----------------------------	-----------------------------	-----	-------------------------------

FIGURE 2.3 – Représentation du tableau T

Autrement dit, pour représenter T , on utilise $L \times r$ bits. Les r premiers bits stockent $T[0]$, les r suivants $T[1]$, etc. Les cases sont contiguës en mémoire.

Remarque.

- Si l'on connaît le type τ , on connaît sa taille r . Si l'on a accès à l'adresse mémoire du début D du tableau, on peut alors aisément calculer :
 - l'adresse de $T[0]$: c'est D .
 - l'adresse de $T[1]$: c'est $D + r$.
 - l'adresse de $T[2]$: c'est $D + 2r$.
 - Ainsi de suite. $T[i]$ a pour adresse $D + ir$.

C'est pour cela que les tableaux sont une structure de donnée aussi efficace : il est très rapide d'accès au i -*me* élément, car cet accès demande une unique calcul à partir de l'adresse du début et de l'indice.

Notez que pour que cette propriété soit vraie, il *faut* que toutes les cases du tableau aient la même taille! C'est pour cela que l'on autorise pas les mélanges de type dans un tableau.

- **Lorsque l'on passe un tableau à une fonction, tout se comporte comme si le contenu du tableau était passé par référence.**

En fait, le tableau est affaibli en pointeur : au lieu de passer le tableau par valeur, on passe à la place l'adresse de sa première case. D'après le point précédent, c'est exactement ce dont on a besoin. Mais puisqu'on a fourni à la fonction l'emplacement mémoire du tableau initial, les cases qu'elle modifie sont celles du tableau.

- Cette façon de manipuler des tableaux est celle de C. Elle a deux inconvénients : la longueur du tableau n'est pas stockée, et le tableau n'est pas redimensionnable (si ça se trouve, les cases mémoire qui suivent la fin du tableau sont déjà prises et on ne peut donc pas agrandir el tableau). OCaml (et Python) utilisent une structure qui permet de stocker la longueur du tableau en plus de ses valeurs⁸. Nous verrons avec les tableaux dynamiques comment crée des tableaux redimensionnables (comme en Python).

8. Dans l'idée, on crée une case fictive au début du tableau qui sert à stocker la longueur.

- Il faut que tous les éléments d'un tableau aient la même taille. En conséquence, si on veut faire un tableau de tableaux, il y a deux grandes façons de faire :
 - Imposer que tous les sous-tableaux aient la même longueur et le même type.
 - Ne pas stocker les sous-tableaux dans les cases, mais l'adresse de leur début (toutes les adresses ont la même taille). Il faut alors stocker les sous-tableaux ailleurs.

(a) Tableaux imbriqués

(b) Tableaux d'adresses de tableaux

FIGURE 2.4 – Les deux grandes façons de faire un tableau de tableaux.

0.5 Chaînes de caractères

Définition 19 (Chaînes de caractères).

Une chaîne de caractère est une succession de caractères, c'est à dire un texte.

On les représente généralement comme une succession de caractères contigus en mémoire. On utilise un caractère particulier, `\0` qui marque la fin (et ne veut rien dire d'autre) :

Premier caractère	Second caractère	...	Dernier caractère	<code>\0</code>
-------------------	------------------	-----	-------------------	-----------------

Remarque. Ceci est la façon de faire de C (qui a l'avantage de la simplicité). Là aussi, on peut stocker la longueur de la chaîne en dur, et/ou faire une représentation plus compliquée qui permet plus d'opérations.

1 Compléments très brefs de programmation

1.0 Notion d'adresse mémoire

La mémoire d'un ordinateur peut être pensée comme un « très gros tableau » : c'est une succession de cases mémoires (dont la taille est un plus petit adressable), ayant chacune un indice. Celui-ci est appelé une **adresse**.

Autrement dit, une adresse indique où est rangée une donnée en mémoire.

Définition 20 (Pointeur).

Une variable qui stocke une adresse est appelée un **pointeur**.

En C, si le contenu de la case mémoire pointée est de type `type`, le type du pointeur sera `type*`.

Nous reparlerons de cette notion en TP. Tout ce qu'il faut retenir pour l'instant, c'est qu'en plus d'utiliser des variables, on peut manipuler des adresses mémoire explicites.

On parle alors d'**objet mémoire** pour désigner quelque chose qui a un contenu et une adresse.

1.1 Portée syntaxique et durée de vie

On a déjà vu comment les fonctions, if-then-else et boucles structurent un code source en différents corps (le corps de la fonction, le corps du if, etc). les lignes qui appartiennent exactement aux mêmes corps sont appelés un **bloc syntaxique**.

Remarque.

- Cette appellation provient de la *syntaxe* : un corps est délimité par quelque chose qui indique son début et quelque chose qui indique sa fin. En C, ces indicateurs sont les `{` et `}`.
- Pour les fonctions, il y a une subtilité : il faut considérer que la déclaration des arguments de la fonction est aussi dans le bloc syntaxique.
- Il y a une notion d'imbrications : par exemple, si une boucle est dans une fonction, on dira que le corps de la boucle est imbriqué dans le bloc de la fonction.

1.1.0 Définition et premiers exemples

Définition 21 (Portée).

La **portée** d'un identifiant (c'est à dire d'un nom de variable ou de fonction) est l'ensemble des endroits où cet identifiant est callable.

Proposition 22 (Portée locale vs globale).

Il y a deux types de portées :

- **locale** : si un identifiant est créé dans un bloc syntaxique, il n'est utilisable qu'à partir de sa création jusqu'à la fin de son bloc syntaxique. On peut l'utiliser dans des blocs qui sont imbriqués dans son corps.
- **globale** : si un identifiant est créé en dehors de tout bloc syntaxique (c'est à dire en dehors de toute fonction), il est utilisable de sa déclaration à la fin du fichier (peu importe les blocs).

Exemple.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int somme = -10;
6  int Lmax = 100000;
7
8
9  int somme_tableau(int T[], int len) {
10     if (len > Lmax) {
11         // faire planter le programme
12         exit(EXIT_FAILURE);
13     }
14
15     int somme = 0;
16     int indice = 0;
17     while (indice < len) {
18         int tmp = 666;
19         somme = somme + T[indice];
20         indice = indice + 1;
21     }
22
23     return somme;
24 }
25
26
27 int main(void) {
28     int T[] = {5, 85, -20};
29     int sortie_somme = somme_tableau(T, 3);
30     printf("variable globale somme : %d\n", somme);
31     printf("sortie de somme_tableau : %d\n", sortie_somme);
32
33     return EXIT_SUCCESS;
34 }

```

Dans l'exemple ci-dessus :

- Dans l'exemple ci-dessus, le bloc syntaxique du `while` de `somme_tableau` commence ligne 18 et se finit ligne 20. Ainsi, la variable `foo` n'est utilisable que entre ces lignes : c'est sa portée.
- Celui de la fonction `somme_tableau` commence ligne 9 (déclaration des arguments) et finit ligne 24. La portée de `indice` est donc de la ligne 16 à la ligne 24. On peut bien utiliser cette variable dans le bloc du `while` puisque celui-ci est imbriqué dans le bloc de la fonction.
- La variable `Lmax` a une portée globale (ligne 6 jusqu'à la fin). On peut donc l'utiliser ligne 10 (dans le bloc de la fonction).

Définition 23 (Masquage).

On parle de **masquage** lorsqu'au sein de la portée d'un identifiant, on redéfinit une nouvelle variable/fonction qui a le même identifiant. L'identifiant d'origine n'est alors plus accessible tant que l'on est dans la portée du nouveau : seul le plus « récent » est utilisable.

Exemple. Dans l'exemple précédent, il y a un masquage du `somme` global (ligne 5) par le `somme` local (ligne 15). On ne peut pas accéder au premier tant que l'on est dans la portée du second (ligne 15-24).

Définition 24 (Durée de vie).

La **durée de vie** d'un objet mémoire (par exemple une variable) est l'ensemble des endroits où cet objet existe et a une valeur définie en mémoire. Cela ne coïncide pas nécessairement avec la portée.

Exemple.

- Dans l'exemple précédent, durant le masquage, on est bien dans la durée de vie du `somme` global (il a toujours sa même valeur en mémoire, il n'a pas été supprimé). Par contre, on est pas dans sa portée à cause du masquage.
- En C, lorsque l'on crée un tableau mais que l'on ne l'a pas encore initialisé (par exemple : `double T[10]`; crée un tableau de 10 nombres à virgule sans l'initialiser), on est dans sa portée puisque l'identifiant a été créé, mais pas dans sa durée de vie puisque le contenu du tableau n'a pas de valeur définie en mémoire.

Proposition 25 (Durée de vie statique, automatique et allouée).

Il y a trois durées de vie possibles :

- **statique** : l'objet est toujours défini en mémoire. C'est le cas des variables globales. Leur valeur est stockée dans une zone particulière de la mémoire, où elles existent dès le début du fichier (avant le début de leur portée, donc).
- **automatique** : l'objet n'existe en mémoire que pendant un bloc. C'est le cas des variables locales : elles "naissent" au début de leur bloc syntaxique et "meurt" automatiquement à la fin de celui-ci.
- **allouée** : on contrôle à la main la "naissance" et la "mort" d'un objet. Cela se fait en C via les fonctions `malloc` et `free`.

Exercice. Dans l'exemple `portee.c` précédent, indiquer des variables statiques et des variables automatiques.

Remarque.

- Si on oublie de libérer la mémoire d'un objet alloué, on provoque une fuite de mémoire : la zone restera allouée jusqu'à la fin du programme, même si on ne s'en sert plus.⁹
- Certains langages de programmation ne proposent pas de contrôler à la main la "mort" des objets de durée de vie allouée. À la place, ils essaient de détecter le moment à partir duquel ils ne seront plus utilisés (et les suppriment alors). On parle de **ramasse-miette**. C'est ce que font OCaml et Python. Cela a l'avantage d'éviter les erreurs du programmeur liée à la mort des objets, mais a l'inconvénient de ralentir le programme (car il faut détecter si une variable servira encore ou non).

Proposition 26.

Les objets à durée de vie automatique ont une propriété de type LIFO (*Last In First Out*) : ce sont ceux du dernier bloc ouvert qui seront supprimés en premier.

Cette propriété provient du fait que les blocs eux-mêmes vérifient cette propriété : le bloc qui a été ouvert le plus récemment est le prochain à fermer. Vous pouvez y penser comme à des parenthèses : quand vous croisez une parenthèse fermante, elle ferme la parenthèse encore ouverte la plus récente.

9. Pire encore : même si on ne peut plus libérer la mémoire, car cette libération avait besoin d'une variable locale à laquelle nous n'avons plus accès.

1.1.1 Exemples avancés

Avec les objets alloués, on peut créer d'autres exemples de désynchronisation entre la durée de vie et capacité à accéder au contenu :

```
1 int* ptr = (int*)
  ↪ malloc(<taille>);
2 free(p);
3 int x = p[0];
```

Lors de la ligne 4 de ce code, l'objet alloué n'existe plus mais on peut encore y accéder via `p`.

⚠ Attention, c'est l'objet pointé par `p` qui est mort. `p`, lui, va très bien et contient toujours la même adresse (sauf qu'il n'y a plus rien à cette adresse).

```
1 int* zombi(void) {
2     int var = 42;
3     return &var;
4 }
```

Cette fonction renvoie l'adresse d'une variable locale. Or celle-ci meurt à la fin de la fonction : on récupère donc un pointeur sur de la mémoire morte.

```
1 void fuite(unsigned int n) {
2     malloc(n*sizeof(char));
3     return;
4 }
```

Après un appel à cette fonction, l'objet mémoire créé existe toujours mais on n'a plus aucun moyen d'y accéder. Il est même impossible de libérer cette mémoire car on n'a pas mémorisé le pointeur qui y mène...

```
1 double* zombieRetour(void) {
2     double tab[5] = {3.14, -4.2,
  ↪     2.71, 5.0};
3     return tab;
4 }
```

Idem qu'à gauche, sauf qu'ici ça se voit moins (surtout si on a fait du Python). Le tableau est affaibli en pointeur lorsqu'on le renvoie, c'est donc exactement la même situation.

Pour enfoncer le clou dans le cadavre des zombies, considérons le code ci-dessous et demandons au compilateur son avis :

```
1 int main(void) {
2     double* tab = zombieRetour();
3     return EXIT_SUCCESS;
4 }
```

Ligne de compilation et résultat :

```
1 MP2I/Cours/Memoire: gcc error.c -Wall -Wextra
2 error.c: Dans la fonction « zombieRetour »:
3 error.c:5:10: attention: la fonction retourne l'adresse d'une variable
  ↪ locale [-Wreturn-local-addr]
4     5 |     return tab;
5       |           ^~~
```

Voilà. Tout est dit.

2 Organisation de la mémoire d'un programme

Dans cette section, nous allons voir comment est « rangée » la mémoire dans le langage C : où vont les octets des différentes variables d'un programme.

Pour OCaml, c'est plus compliqué : la notion de pile d'appels de fonction reste valide, mais le reste est plus différents (pour faire fonctionner le ramasse-miette).

2.0 Bloc d'activation

Vous vous encourage très fortement à aller jouer avec le site ci-dessous pour visualiser la mémoire et les blocs :

<https://pythontutor.com/c.html#mode=edit>

Définition 27 (Bloc d'activation).

Les variables créées par un même bloc syntaxique sont stockées les unes après les autres en mémoire. L'espace mémoire servant à stocker ces variables est appelé **bloc d'activation**.

Il contient donc les variables créées dans ce bloc syntaxique (mais pas dans ses blocs imbriqués).

Rappelons que si le bloc syntaxique est un bloc de fonction, les arguments en font partie.

Exemple.

```

1  int f(int a, int b) {
2      int c = 10;
3      int i = 0;
4      while (i < c) {
5          int d = 42;
6          int c = 50;
7          i = i+1;
8      }
9      int e = 2;
10     return c;
11 }
```

 bloc.c

Le bloc d'activation de la fonction contient un sous-bloc d'activation (boucle for) qui s'étend des lignes 3 à 6.

On peut représenter ces deux blocs sur un schéma où on les « empile » :

FIGURE 2.5 – Blocs d'activation de `f`

Remarque.

- Un appel de fonction ouvre un nouveau bloc (celui du "code" de la fonction appelée).
- Beaucoup de langages ne font pas un bloc d'activation par bloc syntaxique mais simplement un gros bloc d'activation pour toute la fonction.

2.1 Organisation de la mémoire virtuelle d'un programme

On parle de **mémoire virtuelle** (abrégié VRAM) pour désigner la mémoire telle que le système d'exploitation la présente à un programme. C'est une version « mieux rangée » de la mémoire réelle, que le système d'exploitation s'occupe de faire correspondre avec la mémoire réelle.¹⁰

Tout comme la mémoire réelle, la mémoire virtuelle est un « tableau géant » indicé par des adresses. Du point de vue d'un programme, la mémoire virtuelle ressemble à ceci :

FIGURE 2.6 – Mémoire virtuelle vue par un processus.

Légende :

- *Code* : c'est là où les instructions du programme (son... code) sont stockées.
- *Données statiques initialisées* : une donnée statique est une donnée qui est connue à la compilation. Les données statiques initialisées sont les variables globales, ainsi que le contenu des chaînes de caractères¹¹
- *Données statiques initialisées* : hors-programme.
- *Pile mémoire* : c'est là que sont stockés les différents blocs d'activation en cours, empilés les uns après les autres.
- *Tas mémoire* : là où on stocke les objets à durée de vie allouée.

Ces deux dernières zones évoluent durant le programme, au fur et à mesure que des variables sont créées ou détruites.

10. C'est la différence entre la description que je fais à autrui de mes brouillons, et l'état réel de mes brouillons. Tel un système d'exploitation, je suis capable de lire mes brouillons fort mal organisés et de les faire correspondre en direct à quelque chose de plus présentable.

11. Une chaîne de caractère est un `char*`, c'est à dire un pointeur vers une zone où les caractères sont rangés les uns à côté des autres. Cette zone se trouve dans les données statiques initialisées.

2.1.0 Pile mémoire

Quand on entre dans un nouveau bloc syntaxique, on crée son bloc d'activation associé et on le met au bout de la pile mémoire. Quand on quitte un bloc syntaxique, on supprime son bloc d'activation.

FIGURE 2.7 – Schéma de l'évolution de la pile mémoire

Exercice. Représenter de même l'évolution de la pile mémoire lors de l'exécution du programme dont le code C est portée. c.

Proposition 28 (Organisation de la pile mémoire).

Les blocs d'activation sont stockés sur la pile mémoire suivant le principe LIFO. La pile est donc toujours remplie de manière contigu : elle n'a pas de « trou ».
La taille de la pile évolue durant l'exécution.

Remarque. La pile mémoire se comporte comme la structure de donnée pile (que nous verrons plus tard).

2.1.1 Tas mémoire

Les objets à durée de vie allouée sont stockés dans le tas. Quand le programme demande à allouer (« réserver », « faire naître ») x octets, on trouve x octets dans le tas et on les fournit au programme. Ils ne seront pas réutilisables tant que le programme n'aura pas explicitement libéré cette mémoire.

Exemple. Considérons le programme suivant, et montrons un exemple possible d'évolution du tas pour ce programme :

- 1 $x \leftarrow$ allouer 1 cases
- 2 $y \leftarrow$ allouer 3 cases
- 3 $z \leftarrow$ allouer 2 cases
- 4 Désallouer y

Proposition 29 (Non-contigüité du tas mémoire).

Le tas mémoire n'est pas rempli contigüement.

Remarque.

- Cette différence avec la pile mémoire provient de fait que l'on a pas accès à une bonne propriété comme LIFO : les objets alloués peuvent être libérés dans n'importe quel ordre.
- Il existe des stratégies pour essayer de limiter la création de « petits trous » dans le tas. On parle de stratégie d'allocation.
- Le tas mémoire n'a rien à voir avec la structure de donnée tas.

2.1.2 À savoir faire

L'objectif premier de cette section sur la mémoire d'un programme est que vous soyez capable de faire des schémas montrant l'évolution de la pile mémoire et une évolution possible du tas mémoire durant l'exécution d'un programme.¹²

Ce sont des notions importantes à avoir en tête lorsque l'on programme en C avec des pointeurs. En fait, **pour comprendre ce que l'on fait en C, il faut être capable de visualiser la mémoire.**

12. Notez que tout ce qui n'est pas marqué hors-programme est au programme. En particulier, j'ai déjà vu un sujet de concours MPI poser la question « dans quelle zone mémoire sont stockées les variables globales ? ».

Chapitre 3

COMPLEXITÉ

Notions	Commentaires
Analyse de la complexité d'un algorithme. Complexité dans le pire cas, dans le cas moyen. Notion de coût amorti.	On limite l'étude de la complexité dans le cas moyen et du coût amorti à quelques exemples simples.

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
[...] Organisation des activations sous forme d'arbre en cas d'appels multiples.	[...] Les récurrences usuelles : $T(n) = T(n-1) + an$, $T(n) = aT(n/2) + b$, ou $T(n) = 2T(n/2) + f(n)$ sont introduites au fur et à mesure de l'étude de la complexité des différents algorithmes rencontrés. On utilise des encadrements élémentaires <i>ad hoc</i> afin de les justifier ; on évite d'appliquer un théorème-maître général.

Extrait de la section 2 du programme officiel de MP2I : « Récursivité et induction ».

SOMMAIRE

0. Complexité temporelle.....	50
0. Notion de complexité temporelle	50
<i>Opérations élémentaires (p. 51).</i>	
1. Notation de Landau	52
2. Exemples plus avancés	54
<i>Méthodologie (p. 54). Un exemple avec des boucles imbriquées (p. 54). Un exemple où il faut être très précis sur les itérations (p. 55). Écart incompressible entre les bornes (p. 56).</i>	
3. Pire cas, cas moyen, meilleur cas	56
4. Ordres de grandeur	57
1. Complexité spatiale	58
2. Cas particulier des fonctions récursives	59
0. Complexité temporelle des fonctions récursives	60
<i>Cas général : formule de récurrence (p. 60). Cas particuliers : arbre d'appels (p. 60).</i>	
1. Complexité spatiale des fonctions récursives	64
3. Complexité amortie	65
0. Méthodes de calcul	65
<i>Exemple fil rouge (p. 65). Méthode du comptable (p. 65). Méthode du potentiel (p. 67). Méthode de l'aggrégat (p. 69).</i>	
1. Remarques et compléments	70

0 Complexité temporelle

0.0 Notion de complexité temporelle

Définition 1 (Complexité temporelle).

La complexité temporelle d'un algorithme est une estimation du temps qu'il prend à s'exécuter. On peut la mesurer de deux grandes façons :

- En comptant le nombre de certaines opérations choisies. Par exemple, on peut exprimer la complexité en nombre d'additions effectuées. Si la ou les opérations ne sont pas choisies au hasard
- En comptant le nombre d'opérations élémentaires, c'est à dire du nombre d'étapes que prendra le calcul sur un processeur. Cette méthode impose souvent de faire des approximations pour simplifier.

Elle dépend souvent de l'entrée, on l'exprime souvent comme une fonction de l'entrée ou de la taille de l'entrée. Cette fonction est souvent nommée $C()$ ou $T()$.

Remarque.

- Généralement, ce qui nous intéresse est la complexité sur les grandes entrées. L'objectif est de prévoir (à peu près) combien de temps l'exécution d'un programme prendra, pour savoir si'il est raisonnable de le lancer ou s'il faut trouver une autre solution.
- Sauf mention contraire, une complexité est demandée en opérations élémentaires.

Exemple. Considérons la fonction ci-dessous, et calculons sa complexité en nombre d'additions $A()$:

```

9  /** Renvoie la somme des
    ↳ entiers de 0 à n */
10 unsigned somme_entiers(unsigned
    ↳ n) {
11     unsigned somme = 0;
12     unsigned i = 1;
13     while (i <= n) {
14         somme = somme + i;
15     }
16     return somme;
17 }
```

La fonction n'effectue aucune addition en dehors de sa boucle, ni aucun appel de fonctions. À chaque itération de sa boucle, elle effectue deux additions. La boucle itère n fois (car i parcourt l'ensemble $\llbracket 1; n \rrbracket$). D'où $A(n) = 2n$.

Exemple. Considérons les deux fonctions ci-dessous. On note A_{ep} la complexité en nombre d'additions de `est_premier`, et A_{spi} celle de `somme_premiers_inf`.

```

20 /** Teste si n est premier */
21 bool est_premier(int k) {
22     if (k <= 1) { return false; }
23
24     int d = 2;
25     while (d*d <= k) {
26         if (k % d == 0) { return
27             ↳ false; }
28         d = d + 1;
29     }
30     return true;
31 }
```

```

33 /** Renvoie la somme des nombres
    ↳ premiers inférieurs à x */
34 int somme_premiers_inf(int x) {
35     int somme = 0;
36     int n = 0;
37     while (n < x) {
38         if (est_premier(x)) {
39             somme = somme + x;
40         }
41         n = n + 1;
42     }
43     return somme;
44 }
```

Étudions tout d'abord `est_premier`. Elle ne fait aucune addition en dehors de sa boucle (ni aucun appel de fonction). Elle fait une addition par double. La boucle itère au plus $\lfloor \sqrt{k} \rfloor$ fois (d varie au plus de $\llbracket 2$ jusqu'à $\lfloor \sqrt{k} \rfloor + 1$ inclus). Donc $A_{ep}(k) \leq \lfloor \sqrt{k} \rfloor$.

Passons maintenant à `somme_premiers_inf`. Elle ne fait aucun appel de fonction ni addition en dehors de sa boucle. Sa boucle itère pour n allant de 0 à $x - 1$ inclus. À chaque itération, elle effectue au moins 1 addition et au plus $1 + A_{ep}(n)$. Donc au total, on peut encadrer A_{spi} ainsi :

$$\begin{aligned} \sum_{n=0}^{x-1} 1 &\leq A_{spi}(x) \leq \sum_{n=0}^{x-1} (1 + A_{ep}(n)) \\ x &\leq A_{spi}(x) \leq x + \sum_{n=0}^{x-1} (\lfloor \sqrt{n} \rfloor) \\ x &\leq A_{spi}(x) \leq x (1 + \lfloor \sqrt{x} \rfloor - 1) \end{aligned}$$

Remarque.

- Ici, on a encadré et non calculé la valeur précise de la complexité. La complexité précise de `est_premier` était en effet difficile à exprimer. C'est un cas très courant ! On va en fait essayer de *borner* les complexités, avec une borne supérieure et une borne inférieure.
- Vous trouvez peut-être que mon passage de l'avant dernière à la dernière ligne majore très grossièrement. Cela aurait pu être la cas¹, mais en l'occurrence pas tant que ça. Pour vous en convaincre, utilisez le fait que² :

$$\int_{d-1}^d \sqrt{x} dx \leq \sqrt{d} \leq \int_d^{d+1} \sqrt{x} dx$$

- L'expression de la complexité dépend de ce en fonction de quoi on l'exprime. Il y a deux grandes options :
 - Exprimer la complexité en fonction de la *valeur* des arguments (ce que l'on a fait ci-dessous).
 - Exprimer la complexité en fonction de la *taille* des arguments, c'est à dire en fonction du nombre de bits nécessaires pour représenter les arguments. Cette deuxième façon de faire prend tout son sens lorsque l'on essaye de comparer la difficulté de différents problèmes (cf MPI).
- Or, la taille des arguments est ne correspond pas à la valeur ! Penser à la taille de l'encodage binaire d'un entier, par exemple.

À retenir : il faut toujours indiquer en "fonction de quoi" on exprime la complexité.

0.0.0 Opérations élémentaires

Définition 2 (Opération élémentaire).

ne **opération élémentaire** est une opération « de base », au sens où elle n'est pas combinaison d'autres. Cela inclut par exemple :

- La lecture ou l'écriture d'une case mémoire.
- Une opération logique (NON, ET, OU, etc) ou arithmétique (+, -, %, etc) ou une comparaison du contenu de deux variables.
- Démarrer un appel de fonction ou renvoyer une valeur.
- Afficher 1 caractère.
- Et d'autres.

1. Dans beaucoup de calculs de complexité, il est facile d'obtenir une borne supérieure très large mais difficile de la rendre plus précise. Le monde est mal fait.

2. Si c'est un exo de maths trop dur pour l'instant, revenez-y au second semestre.

Remarque. Voici des exemples d'opérations qui *ne* sont pas élémentaires :

- Tester l'égalité de deux tableaux (cela demande de tester l'égalité de chacune des cases, c'est à dire de faire plusieurs tests d'égalité de variables).
- Afficher un texte (cela demande plusieurs affichages d'1 caractère).

Proposition 3.

En réalité, toutes les opérations élémentaires ne prennent pas le même temps, et le temps d'une même opération peut différer selon la machine qui l'exécute. Il est donc inutile d'essayer de compter le nombre exact d'opérations élémentaires. À la place, on cherche à compter **l'ordre de grandeur** du nombre d'opérations élémentaires.

Ainsi, au lieu de dire « on fait 47 opérations élémentaires », on dira « on effectue un nombre constant d'opérations élémentaires ».

Remarque. L'un des buts de la théorie de la complexité est d'analyser un algorithme indépendamment de la machine ou du langage qui l'exécute, afin de permettre des comparaisons des algorithmes et non des implémentations. Cela ne signifie pas que les comparaisons des langages / machines n'est pas pertinente ; ces dernières ce font juste différemment.

Exemple. Reprenons `somme_entiers`. En dehors de sa boucle, elle effectue un nombre constant d'opérations élémentaires. De même, chaque itération effectue un nombre constant d'opérations élémentaires. Or, la boucle itère n fois, donc la complexité $C(n)$ est linéaire en n .

0.1 Notation de Landau

Vous verrez ces notions plus proprement et plus en profondeur en mathématiques. Ici je m'efforce de faire une approche avant tout intuitive, quitte à piétiner certains points de rigueur.³

Définition 4 (Notations de Landau).

Soient (u_n) et (v_n) deux suites de réels. On dit que :

- (u_n) est dominé par (v_n) si quand n est grand, on a $|u_n| \leq K \cdot |v_n|$ (avec $K > 0$ une constante). Cela revient à dire que $\frac{|u_n|}{|v_n|}$ est majoré.
On note alors $u_n = O(v_n)$.
- (v_n) domine (u_n) si quand n est grand, on a $K \cdot |u_n| \leq |v_n|$ (avec $K > 0$ une constante). Cela revient à dire que $\frac{|v_n|}{|u_n|}$ est minoré (par une constante strictement positive).
On note alors $v_n = \Omega(u_n)$.
C'est en réalité équivalent au fait que u_n est dominé par v_n .
- (u_n) est de l'ordre de (v_n) si $u_n = O(v_n)$ et $v_n = O(u_n)$. Cela revient à dire que $\frac{|v_n|}{|u_n|}$ tend vers une constante strictement positive.
On note alors $u_n = \Theta(v_n)$.

En résumé : $O()$ indique l'ordre de grandeur de majoration, $\Omega()$ celui d'une minoration, et Θ les deux à la fois.

Exemple.

- $n - 50 = O(n)$; mais aussi $n + 42 = O(n)$ ou encore $3n + 12 = O(n)$
- $n + 3\sqrt{n} = O(n)$ (prendre $K = 4$ comme constante).
- $n + 3\sqrt{n} = \Omega(n)$; et donc $n + 3\sqrt{n} = \Theta(n)$.
- Pour la fonction `somme_premiers_inf`, on a prouvé que $A_{spi}(x) = O(x\sqrt{x})$ et que $A_{spi}(x) = \Omega(x)$.

Remarque.

3. J'ai essayé l'approche rigoureuse les années précédentes. Cela n'a pas marché.

- **Le but de ces notations est de ne garder que l'ordre de grandeur du terme dominant dans une majoration / minoration.** En effet, comme on l'a vu avec les opérations élémentaires, les calculs de complexité négligent souvent des « détails » de la réalité. Plutôt que de calculer une complexité fausse, on préfère se contenter de calculer l'ordre de grandeur de la vraie complexité (ou d'une borne de celle-ci).
- Il ne faut pas penser que $O(\dots)$ est une valeur précise. Deux suites différentes peuvent être dominée à l'aide d'un même $O(\dots)$. Par exemple, $(n+1)_{n \in \mathbb{N}}$ et $\left(\frac{n}{3} + 100\right)_{n \in \mathbb{N}}$ sont toutes les deux dominées par $(n)_{n \in \mathbb{N}}$, donc toutes les deux $= O(n)$.
- En maths, vous définirez l'équivalence \sim . Ce n'est pas la même chose que le $\Theta()$, ne confondez pas.
- On essaye de mettre une borne la plus « simple » possible dans le $O()$ (ou Ω ou Θ). Par exemple, on préfère $O(n^2)$ plutôt que $O(\pi n^2 - \sqrt{2.71} \log_5 n)$. Au risque de me répéter, le but est de simplifier en ne mémorisant que l'ordre de grandeur du terme dominant.

Proposition 5 (Relation usuelles de domination).

Je note ici $x \ll y$ le fait que $x = O(y)$. Soient $a, b \in \mathbb{R}_+^*$ et $c \in]1; +\infty[$. On a :

$$\log(n)^a \ll n^b \ll c^n \ll n!$$

Démonstration. Il s'agit des croissances comparées (une version plus faible des puissances comparées, pour être précis). Vous les prouverez en maths. \square

Proposition 6 (Opérations sur les dominations).

J'indique ici quelques relations utiles. Vous en verrez plus en maths.

- La domination est transitive, c'est à dire que si $u_n = O(v_n)$ et $v_n = O(z_n)$ alors $u_n = O(z_n)$.
- On peut sommer une quantité finie de dominations, c'est à dire que :
 - Si $u_n = O(y_n)$ et $v_n = O(z_n)$, alors $u_n + v_n = O(|y_n| + |z_n|)$.
 - Idem avec $\Omega()$.
 - Idem avec Θ .

En particulier, $u_n = O(z_n)$ et $v_n = O(z_n)$ entraîne $u_n + v_n = O(z_n)$.

- On peut sommer une quantité variable de dominations uniquement si elles cachent une même constante K . En conséquence, on peut par exemple sommer les dominations des itérations successives d'une boucle !
- On peut multiplier une quantité finie de dominations, c'est à dire que :
 - Si $u_n = O(y_n)$ et $v_n = O(z_n)$, alors $u_n \cdot v_n = O(|y_n| \cdot |z_n|)$.
 - Idem avec $\Omega()$.
 - Idem avec Θ .
- On peut multiplier une quantité variable de dominations uniquement si elles cachent la même constante K .

Remarque. Si cette histoire de « quantité variable de domination » vous perturbe (ou que vous ne comprenez pas pourquoi on ne peut pas forcément sommer dans ce cas), attendez le cours de maths sur le sujet.

Exemple. On a en fait déjà utilisé ces propriétés sont le dire dans le calcul de la complexité en opérations élémentaires

0.2 Exemples plus avancés

0.2.0 Méthodologie

La méthode utilisée pour ces exemples est toujours la même :

- Calculer d'abord la complexité des appels de fonction.
- Calculer la complexité en dehors des boucles.
- Pour chaque boucle, calculer la complexité d'une itération (condition + corps), puis le nombre d'itérations⁴, et en déduire la complexité de la boucle entière.
En cas de boucle imbriquées, on commence par les boucles intérieures.

0.2.1 Un exemple avec des boucles imbriquées

On considère la fonction ci-dessous :

```

47  /** Échange *a et *b */
48  void swap(int* a, int* b) {
49      int tmp = *a;
50      *a = *b;
51      *b = tmp;
52      return;
53  }
54
55
56  /** Trie tab par ordre croissant en temps quadratique */
57  void tri_bulle(int tab[], int len) {
58      int deja_trie = 0;
59      while (deja_trie < len) {
60
61          int indice = 0;
62          // Cette boucle fait remonter le plus grand élément
63          // non-encore trié et le place à sa place finale
64          while (indice + 1 < len - deja_trie) {
65              if ( tab[indice] > tab[indice+1] ) {
66                  swap( &tab[indice], &tab[indice+1] );
67              }
68              indice = indice + 1;
69          }
70
71          deja_trie = deja_trie + 1;
72      }
73      return;
74  }

```

On veut calculer sa complexité T en nombre de comparaisons en fonction de la longueur du tableau⁵.
Commençons par remarquer que la fonction `swap` n'effectue aucune comparaison. On ne prendra donc pas en compte ses appels dans les calculs qui suivent.

Étudions maintenant La fonction `tri_bulle` :

- Elle n'effectue aucune comparaison en dehors de ses boucles.
- À chaque itération de la boucle intérieure (lignes 60-65), elle effectue deux comparaisons (une dans la condition, une dans le `if` du corps). Elle effectue une dernière comparaison lorsqu'elle quitte la boucle. Comme cette boucle itère $\text{len} - \text{deja_trie}$ fois, elle effectue au total $2(\text{len} - \text{deja_trie}) + 1$ comparaisons.

4. S'il n'y a pas de difficulté particulière, on peut se contenter d'affirmer le nombre d'itérations sans le justifier.

5. C'est un critère assez courant pour évaluer un tri.

- La boucle principale (ligne 56-69) effectue une comparaison dans sa condition, et contient la boucle précédente. La variable `deja_trie` parcourt $\llbracket 0; \text{len} - 1 \rrbracket$. On effectuera une dernière comparaison quand on quittera la boucle. Donc au total, cette boucle fait :

$$\begin{aligned}
 1 + \sum_{\text{deja_trie}=0}^{\text{len}-1} (2(\text{len} - \text{deja_trie}) + 1) &= 1 + \text{len} - 2 \sum_{\text{deja_trie}=0}^{\text{len}-1} (\text{len} - \text{deja_trie}) \\
 &= 1 + \text{len} + 2\text{len}^2 - 2 \sum_{\text{deja_trie}=0}^{\text{len}-1} \text{deja_trie} \\
 &= 1 + \text{len} + 2\text{len}^2 - (\text{len} - 1)(\text{len}) \\
 &= 1 + \text{len}^2
 \end{aligned}$$

- On peut en déduire que :

$$T(\text{len}) = \text{len}^2 + 1 = \Theta(\text{len}^2)$$

Remarque.

- On aurait pu prouver $T(\text{len}) = O(\text{len}^2)$ plus facilement : au lieu de donner la valeur précise du nombre de comparaisons de la boucle intérieure effectuée, on dit qu'elle effectue $O(\text{len})$. Chaque itération de la boucle principale effectue donc $O(\text{len})$ comparaisons, or cette boucle itère len fois, donc $T(\text{len}) = O(\text{len}^2)$.
- « Il y a deux boucles imbriquées donc la complexité est quadratique » n'est pas une preuve, et n'est pas toujours vrai.

0.2.2 Un exemple où il faut être très précis sur les itérations

On suppose que l'on dispose d'une fonction `affiche_entiers_l_bits` qui, en temps $\Theta(2^l)$ affiche tous les entiers pouvant s'écrire sur l bits. On considère la boucle ci-dessous :

```

187 int l = 0;
188 while (l <= n) {
189     printf("Voici les entiers binaires sur %d bits : \n\t", l);
190     affiche_entiers_l_bits(buffer, l);
191     l = l + 1;
192 }
```

(Vous pouvez ignorer le premier argument de `affiche_entiers_l_bits`, c'est un détail d'implémentation.)

Calculons la complexité de cette boucle. Notons tout d'abord qu'il y a un nombre constant d'opérations élémentaires en dehors de la boucle. (Pour la suite, je propose deux raisonnements.)

Version 1 : La boucle itère l de 0 à n inclus. Chaque itération effectue des opérations en temps constant et un appel en $\Theta(2^l)$, et est donc en $O(2^n)$. En sommant sur les itérations, on obtient une complexité T pour la boucle qui vérifie $T(n) = O(n2^n)$.

Dans cette version, on a donné une borne grossière à chaque itération, afin que la somme des itérations soit simple à calculer.

Version 2 : La boucle itère l de 0 à n inclus. Chaque itération effectue des opérations en temps constant et un appel en $\Theta(2^l)$, et l'itération a donc une complexité de l'ordre de 2^l . En sommant sur les itérations, on obtient que le tout est de l'ordre de $\sum_{l=0}^n 2^l = 2^{n+1} - 1$. D'où $T(n) = \Theta(2^n)$.

Ici, on a donné la valeur exacte de l'ordre de grandeur de chaque itération. Sommer sur les itérations a en conséquence été un peu plus compliqué, mais on a obtenu une valeur plus précise à la fin.

Remarque. L'intérêt des majorations grossières est de simplifier les calculs. On peut ainsi obtenir très rapidement des majorations qui prouvent qu'un code s'exécutera en temps raisonnable⁶.

Exercice. Reprendre la preuve de la complexité en nombre de comparaisons du tri bulle. Refaire avec une majoration grossière. Commenter.

0.2.3 Écart incompressible entre les bornes

Revenons à `est_premier` :

```

20  /** Teste si n est premier */
21  bool est_premier(int k) {
22      if (k <= 1) { return false; }
23
24      int d = 2;
25      while (d*d <= k) {
26          if (k % d == 0) { return false; }
27          d = d + 1;
28      }
29      return true;
30  }
```

Avec les calculs déjà effectués, on conclut que la complexité en additions A_{ep} vérifie $A_{ep} = \Omega(1)$ (car la boucle itère au moins une fois) et $A_{ep} = O(k)$.

On peut prouver qu'il existe des k aussi grand que l'on veut pour lesquels la boucle itère une seule fois (les entiers pairs). Similairement, il existe des k aussi grands que l'on veut pour lesquels la boucle itère \sqrt{k} fois (les entiers premiers). Il est donc impossible d'obtenir un Θ .

Remarque. Cet exemple peut-être vu comme un cas particulier de la différence entre « meilleur cas » et « pire cas ».

0.3 Pire cas, cas moyen, meilleur cas

Parfois, pour une même taille d'entrée, la complexité peut varier. `est_premier` en était un exemple, mais en voici un encore plus clair (et plus fort) :

```

202  /** Renvoie true si x dans tab,
203   * et false sinon.
204   */
205  bool mem(int x, int const tab[], int len) {
206      int indice = 0;
207      while (indice < len) {
208          if (tab[indice] == x) { return true; }
209          indice = indice + 1;
210      }
211      return false;
212  }
```

Pour len aussi grand que l'on veut, il existe des tableaux de longueur len sur lesquels cette fonction s'exécute en temps $\Theta(1)$ et d'autres sur lesquels elle s'exécute en temps $\Theta(\text{len})$.

Autrement dit, on ne peut même pas définir la complexité comme une fonction dépendant uniquement de len ! C'est pourtant ce que l'on voudrait : on voudrait pouvoir prévoir le temps d'exécution d'une fonction à partir de critères simples, comme la longueur.

6. Vous ferez une utilisation similaire de majorations grossières en mathématiques en deuxième année, pour prouver qu'une série (ou une série de fonction) est « raisonnable ».

Définition 7 (Pire cas, meilleur cas).

- La **complexité dans le pire des cas** est le nombre *maximal* d'opérations que la fonction exécute sur une entrée d'une taille donnée.
- La **complexité dans le meilleur des cas** est le nombre *minimal* d'opérations que la fonction exécute sur une entrée d'une taille donnée.
- La **complexité dans le cas moyen** est le nombre moyen d'opérations que la fonction exécute sur une entrée d'une taille donnée. La définition de « nombre moyen » demande donc de sommer (et moyenner) sur toutes les entrées possibles de cette taille. On pondère parfois ces entrées par une probabilité afin de mieux coller à une situation pratique.

Exemple. La complexité (pire des cas) de `mem` est $\Theta(\text{len})$. Sa complexité meilleur des cas est $\Theta(1)$. Sa complexité en cas moyen... est difficile à définir, car il faut déjà définir ce que signifie moyenner sur tous les tableaux de longueur `len` lorsqu'il y en a une infinité.⁷

Convention 8.

Sauf mention contraire, toute complexité demandée est un pire des cas.

Remarque.

- **Ne confondez pas** « pire cas » avec « majoration de la complexité » et « meilleur cas » avec minoration. Ce qui est vrai, c'est que le meilleur des cas est une minoration du pire des cas. Mais il est souvent possible d'obtenir de meilleurs minoration.
- **Ne confondez pas** « meilleur cas » avec « entrée de petite taille ». On s'intéresse à la complexité sur des *grandes* entrées!⁸

0.4 Ordres de grandeur

On considère une fonction dont la complexité est $C(n)$. On dit que la fonction est :

- de complexité constante si $C(n) = O(1)$.
- polylogarithmique en n si $C(n) = O(\text{poly}(\log(n)))$ avec *poly* une fonction polynomiale.
- linéaire en n si $C(n) = O(n)$. Similaire pour quadratique en n , cubique en n , polynomiale en n .
- exponentielle en n si $C(n) = O(e^{\text{poly}(n)})$ avec *poly* une fonction polynomiale.

Notez que je précise à chaque fois « en n », pour bien rappeler en fonction de quoi la complexité est exprimée.

Il est impossible de traduire une complexité en une durée exacte à la nano seconde près : trop de facteurs liés au langage et à la machine entrent en compte. On peut cependant appliquer la méthode suivante pour avoir une approximation très grossière :

- 1) Évaluer la valeur de la complexité sur l'entrée voulue
- 2) Diviser par 1Ghz la fréquence du processeur (c'est un arrondi du nombre d'opérations par s de votre processeur).
- 3) Multiplier le tout par quelque chose entre 2 et 500 (pour prendre en compte la constante cachée du $O()$).

Si le résultat est raisonnable, votre code terminera très vite ; sinon croisez fort les doigts.

7. Les « sommes infinies » posent des difficultés mathématiques, comme vous le verrez en maths.

8. Les petites entrées terminent rapidement en pratique.

Voici un tableau des arrondis des valeurs obtenues par les étapes 1) et 2) de la méthode précédente :

$T(x) \backslash x$	10	100	1000	10000	10^6	10^9
$\Theta(\log x)$	1ns	10ns	10ns	10ns	10ns	100ns
$\Theta(\sqrt{x})$	1ns	10ns	100ns	100ns	1 μ s	1ms
$\Theta(x)$	10ns	100ns	1 μ s	10 μ s	1ms	1s
$\Theta(x \log x)$	10ns	100ns	10 μ s	100 μ s	10ms	10s
$\Theta(x^2)$	100ns	10 μ s	1ms	100ms	10min	10 ans
$\Theta(x^3)$	1 μ s	1ms	1s	10 min	10 ans	∞
$\Theta(2^x)$	1 μ s	∞	∞	∞	∞	∞

FIGURE 3.1 – Ordre de grandeurs de temps de calculs pour quelques complexités et tailles d'entrées. Non-contractuel.

1 Complexité spatiale

Dans l'idée, la complexité spatiale est l'espace utilisé par une fonction. Cependant, pour des raisons théoriques que vous verrez peut-être plus tard, on utilise un modèle extrêmement simplifié. Il a le mérite de permettre des comparaisons entre des algorithmes de manière indépendante du langage et de la machine.

Définition 9 (Complexité spatiale).

La complexité spatiale d'une fonction est le nombre de cases mémoires dont elle a besoin pour s'exécuter. On ne compte pas dedans :

- Les entrées si elles sont en lecture seule.
- La sortie si elle est en lecture seule.

Autrement dit, on compte l'espace *supplémentaire* dont la fonction a besoin.

On la note souvent E ou S . Les notions de pire, moyen et meilleur cas s'appliquent ici aussi.

Remarque.

- Comme pour la complexité temporelle, seul l'*ordre de grandeur* a véritablement un sens en pratique.
- Cette définition a des limites quand on l'applique à un langage pratique. Par exemple, si un programme C alloue n cases mémoires avec `malloc` mais ne les utilise pas, cet espace doit-il compter ? Dans de tels cas, il ne faut pas hésiter à préciser : on peut par exemple distinguer l'espace *utilisé* de l'espace *demandé*.
- Si une même zone mémoire est utilisée par plusieurs appels de fonctions (cela correspond aux arguments passés par pointeur/référence), il ne faut pas la compter plusieurs fois puisque c'est le même espace qui est utilisé !

Exemple. Toutes les fonctions vues jusqu'à présent sauf `tri_bulle` sont en espace constant car elles créent un nombre fixe de variables, dont la taille est fixe.

Par contre, `tri_bulle` lit et modifie $O(\text{len})$ cases du tableau : elles ne sont donc ni des entrées en lecture seule ni la sortie en écriture seule, et comptent donc la complexité spatiale qui est donc un $O(\text{len})$.

En fait, pour avoir une complexité spatiale variable, il faut :

- soit créer/modifier un objet dont la taille est une variable (par exemple un tableau à n cases).

- soit faire de la récurrence.

Théorème 10 (Bornes temps-espace (version simplifiée)).

On considère une fonction qui termine. On note sa complexité temporelle $T(n)$ (en opérations élémentaires) et spatiale $E(n)$ (on compte uniquement l'espace utilisé). On suppose en outre que l'ordre de $E(n)$ n'est pas constant. On a :

$$E(n) \leq T(n) \leq 2^{E(n)}$$

Démonstration. Je vais prouver l'inégalité de gauche, et donner l'intuition de la preuve de celle de droite :

- Utiliser 1 case de la mémoire demande d'y faire une lecture/écriture, et donc au moins 1 opération élémentaire. D'où $E(n) \leq T(n)$.
- L'exécution d'une fonction est déterministe. En partant d'une ligne précise, et de valeurs précises des variables, il y a une seule exécution possible. On nomme (grosso modo) « état mémoire » d'une fonction la donnée de la ligne du code où on en est ainsi que des valeurs des variables. Toutes ces informations sont stockées dans la mémoire. Par déterminisme de l'exécution, si un programme passe deux fois par exactement le même état mémoire, elle est en train de faire une boucle infinie. Or, si on utilise $E(n)$ cases, il n'y a que $2^{E(n)}$ états mémoire possibles car l'unité élémentaire est le bit. D'où $T(n) \leq 2^{E(n)}$.

□

Remarque.

- La borne supérieure est hors-programme, mais l'idée de la preuve est très intelligente. On note généralement cette borne supérieure $T(n) \leq 2^{O(E(n))}$, ce qui est plus rigoureux. J'ai ici essayé de simplifier, quitte à tordre un peu la vérité.
- Notez que la preuve de la majoration n'utilise pas l'hypothèse $E(n)$ non-constant. En fait, l'hypothèse est uniquement là car ce que l'on appelle « espace constant » d'un point de vue pratique n'est pas un espace constant d'un point de vue théorique (notamment à cause des considérations de la taille des entiers⁹). Plus d'infos à ce sujet en MPI.
- Ce théorème n'est en fait généralement même pas écrit à l'aide de E et T , mais à partir des classes de complexité en temps $DTIME$ (plus à ce sujet en MPI) et $DSPACE$ (plus à ce sujet en école).

2 Cas particulier des fonctions récursives

On voudrait analyser la complexité d'une fonction récursive. Par exemple :

Fonction Hanoï

Entrées : i : le pic de départ ; j : le pic d'arrivée ; n : le nombre de disques à déplacer

```

1 si  $n > 0$  alors
2    $k \leftarrow$  pic autre que  $i$  ou  $j$ 
3   HANOÏ( $i, k, n-1$ )
4   Déplacer 1 disque de  $i$  à  $j$ 
5   HANOÏ( $k, j, n-1$ )
```

Cette fonction effectue 1 déplacement par appel, mais il faut prendre en compte tous les appels... On va distinguer deux choses :

9. D'un point de vue théorique, les entiers ne peuvent pas déborder et ont une taille variable. Subséquemment, dans la théorie l'espace constant n'existe (quasiment) pas. Ce n'est pas le point de vue que nous utilisons ici.

Définition 11 (Coût local).

- Le **complexité locale à l'appel** : ce sont les ressources utilisées par l'appel en cours.
- Le **complexité des appels récursifs** : ce sont les ressources utilisées par les appels récursifs.

Pour calculer la complexité total, il faudra prendre en compte ces deux éléments.

Remarque. Le terme de « coût » est parfois utilisé comme synonyme de « complexité ». On parle donc aussi de « coût local » et de « coût récursif ».

2.0 Complexité temporelle des fonctions récursives

Proposition 12 (Complexité temporelle récursive).

La complexité temporelle d'une fonction est la somme des complexités temporelles locales de chacun des appels récursifs.

Il « suffit » donc pour la calculer d'être capable de :

- Calculer la complexité locale d'un appel.
- Sommer sur les appels.

C'est cette deuxième étape qui est généralement la plus difficile.

2.0.0 Cas général : formule de récurrence

La complexité temporelle s'exprime naturellement comme une fonction récursive. Si cette suite est une suite facile que l'on sait résoudre grâce au cours de maths, c'est gagné.

Exemple. La complexité en déplacements $D(n)$ de HANOÏ vérifie $D(n) = 2D(n-1) + 1$ et $D(0) = 0$. C'est une suite arithmético-géométrique que l'on sait résoudre, et dont la solution est $D(n) = 2^n - 1$.

Remarque.

- Lorsque l'on raisonne en opérations élémentaires, on n'a jamais une véritable égalité. Par exemple, pour HANOÏ, on obtient $T(n) = 2T(n-1) + O(1)$, c'est à dire qu'il existe $K > 0$ tel que $T(n) \leq 2T(n-1) + K$. Ici, on a une *majoration* par un terme arithmético-géométrique. On en déduit que $T(n)$ est inférieur à la suite arithmético-géométrique en question, et on obtient $T(n) = O(2^n)$. On peut faire de même pour les minoration.

- Lorsque l'on raisonne en opérations élémentaires, on fait souvent ce que j'ai fait dans le point précédent : on donne uniquement la formule de récurrence ($T(n) = 2T(n-1) + O(1)$), mais pas le cas de base. Cela signifie implicitement que le cas de base est de complexité constante, et qu'il est atteint lorsque n passe en-dessous d'une valeur fixe. On peut prouver que le choix de cette valeur fixe ne change pas l'ordre de grandeur de la complexité.

Ainsi, dans l'exemple précédent, le cas de base est atteint en $n < 1$ et a un coût $O(1)$. Changer ce seuil à $n < 4$ donnerait $T(n) = O(2^{n-3}) = O(2^n)$.

Si la complexité n'est pas une suite « gentille » que l'on sait résoudre grâce au cours de maths, on peut tout de même essayer d'observer les premiers termes de la suite et de trouver une logique et de peut-être repérer une complexité qui ressemble à un classique que l'on sait traiter. De manière générale, *essayer de se ramener à ce que l'on maîtrise* est un excellent raisonnement en sciences !

2.0.1 Cas particuliers : arbre d'appels

Rappel du cours sur la récursivité : on peut représenter la suite des appels récursifs sous la forme d'un arbre.

On peut parfois utiliser cette représentation pour facilement sommer les complexités locales des appels. Cela revient en fait à réécrire la somme des coûts des appels en faisant des « paquets » d'appels qui ont les mêmes paramètres de complexité (ces paquets sont les lignes de l'arbre si on a bien représenté l'arbre).

Exemple. Voici l'arbre des coûts des appels de HANOÏ :

FIGURE 3.2 – Arbre des coûts en déplacements des appels de HANOÏ

Pour calculer à l'aide d'un tel arbre, on procède ainsi :

- 1) On calcule le coût d'un noeud de l'arbre (= on calcule la complexité locale d'un appel).
- 2) On compte le nombre de noeuds par ligne (= le nombre d'appels qui ont les mêmes paramètres pour la complexité), et on en déduit le coût d'une ligne de l'arbre.
- 3) On somme sur les lignes pour conclure.

Les étapes 2) et 3) sont les plus difficiles, car il est aisé d'y affirmer une formule fausse. Il est donc très important d'inclure une ligne au milieu de l'arbre qui montre la forme générale d'une ligne « quelconque », car cette ligne résume le raisonnement.

Exemple. Dans l'arbre précédent, sans compter la ligne du bas (qui effectue 0 déplacements) il y a n lignes. Il y a 2^i appels sur la ligne des appels qui ont $n-i$ en entrée. Ainsi, une ligne associée à i a un coût de 2^i . Les lignes vont de $i = 0$ à $i = n-1$. En sommant sur ces lignes, on obtient $D(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$.

Remarque. Parfois, l'arbre obtenu n'a pas une forme simple à décrire. On peut alors essayer de majorer par un arbre « imple à calculer » en « rajoutant des appels là où il en manque » :

FIGURE 3.3 – Arbre d'appels et majoration pour $T(n) = T(n-1) + T(n-2)$ et cas de base constant.

Diviser pour régner : les arbres d'appels de fonctions « Diviser pour Régner » sont particuliers, mais se résolvent de manière similaire. On se place dans le cas suivant : un problème de taille n est découpé en r sous-problèmes de taille n/c . Le coût local (séparation/fusion ou cas de base) est donné par la fonction f .

On admet que **l'on peut ignorer les éventuelles parties entières dans l'équation de récurrence sans que cela ne change l'ordre du grandeur du résultat**¹⁰. L'équation de complexité est donc :

$$T(n) = r.T(n/c) + f(n)$$

Voici un schéma de l'arbre d'appel¹¹, où L est la hauteur de l'arbre d'appels :

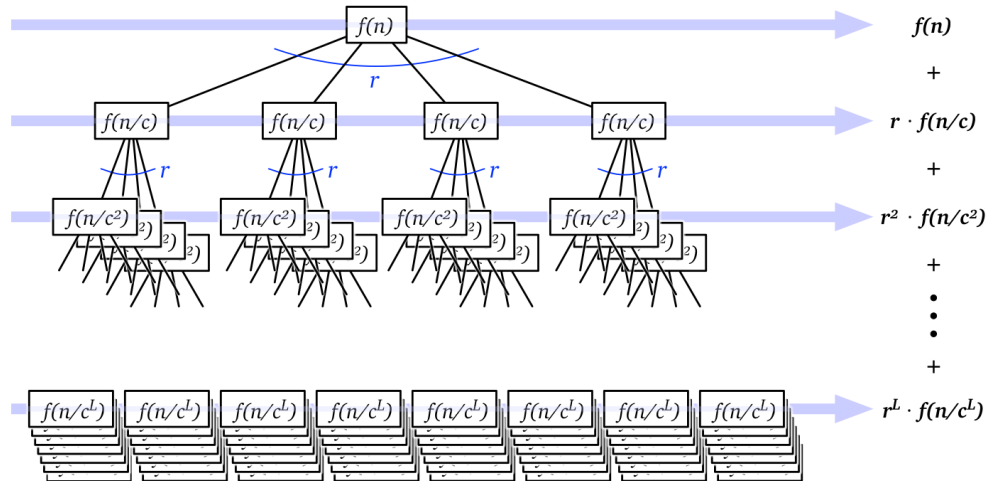


FIGURE 3.4 – Arbre de complexité diviser pour régner

Les coûts T_i de chacune des lignes sont indiqués le long de l'arbre (on a $T_i = r^i f(n/c^i)$). La complexité totale est la somme des coûts de ces lignes. Il existe 3 cas courants dans lesquels une majoration précise de cette somme est simple à calculer :

- Si les T_i décroissent (au moins) géométriquement, c'est à dire s'il existe $K > 1$ tel que $T_{i+1} \leq \frac{T_i}{K}$.
On a alors pour tout i , $T_i \leq \frac{T_0}{K^i}$ et donc :

$$\begin{aligned} T(n) &= \sum_{i=0}^L T_i \\ &\leq \sum_{i=0}^L \frac{T_0}{K^i} \\ &\leq f(n) \left(\sum_{i=0}^L \frac{1}{K^i} \right) && \text{en factorisant par } T_0 = f(n) \\ &= O(f(n)) && \text{car le terme entre parenthèses tend vers une constante positive} \end{aligned}$$

Autrement dit, si les coûts des lignes décroissent (au moins) géométriquement, le coût de la première ligne domine.

- Si les T_i croissent (au moins) géométriquement, c'est à dire s'il existe $K > 1$ tel que $T_{i+1} \geq K.T_i$.
On a alors pour tout i , $T_i \leq \frac{T_L}{K^{L-i}}$ et donc :

10. Si vous êtes accroché-es en maths, vous pouvez aller voir le lien suivant pour une preuve de cela : <https://courses.csail.mit.edu/6.046/spring04/handouts/akrabazzi.pdf>

11. Issu de l'excellent *Algorithms* de J. Erikson.

$$\begin{aligned}
T(n) &= \sum_{i=0}^L T_i \\
&\leq \sum_{i=0}^L \frac{T_L}{K^{L-i}} \\
&\leq r^L f(n/c^L) \left(\sum_{i=0}^L \frac{1}{K^{L-i}} \right) && \text{en factorisant par } T_L = r^L f(n/c^L) \\
&= O(r^L f(n/c^L)) && \text{car le terme entre parenthèses tend vers une constante positive}
\end{aligned}$$

Autrement dit, si les coûts des lignes croissent (au moins) géométriquement, le coût de la dernière ligne domine. Notez que dans ce cas, il faut calculer L . C'est généralement un $O(\log_c(n))$, car c'est généralement le premier i tel que n/c^i passe en dessous du seuil du cas de base.

- Si les T_i sont constants, on a :

$$\begin{aligned}
T(n) &= \sum_{i=0}^L T_i \\
&= \sum_{i=0}^L f(n) && \text{car } \forall i, T_i = T_0 = f(n) \\
&= O(Lf(n))
\end{aligned}$$

Notez qu'ici aussi il faut calculer L .

Résumons :

Proposition 13 (Méthode de calcul Diviser pour Régner).

Pour calculer une majoration du coût temporel d'un algorithme Diviser pour Régner dont l'équation de récurrence est $T(n) = r.T(n/c) + f(n)$, on utilise la méthode suivante :

- Si les coûts des lignes de l'arbre décroissent (au moins) géométriquement de haut en bas, alors on prouve que le coût de la racine est dominant.
- Si les coûts des lignes croissent (au moins) géométriquement de haut en bas, alors on prouve que le coût des cas de base est dominant.
- Si les coûts des lignes sont constants, on les somme simplement.

Remarque.

- **Il faut savoir identifier et gérer ces 3 cas !!!!!!!** Et pour ça, pas de secret, il faut connaître les 3 cas et s'entraîner en refaisant ces calculs.
- On peut aussi obtenir les minoration associées, mais généralement on se contente d'un $O()$.
- Il existe un théorème célèbre, appelé le « théorème maître ». C'est une application de la propriété précédente au cas où f est un polynôme. Je vous déconseille fortement d'essayer de l'apprendre par coeur : vous ferez des erreurs en mémorisant ces formules¹². Vous **n'**avez **pas** le droit de l'utiliser en MP2I/MPI. Je vous le donne ci-dessous à titre informatif :

Proposition 14 (Théorème Maître (hors-programme)).

On considère une formule de récurrence de la forme $T(n) = aT(n/b) + O(n^d)$ avec comme cas de base $T(1) = 1$. Alors :

- Si $a < b^d$, alors $T(n) = O(n^d)$ (« le coût de la racine domine »).
- Si $a > b^d$, alors $T(n) = O(n^{\log_b(a)})$ (« le coût des feuilles domine »).
- Si $a = b^d$, $T(n) = O(n^d \log(n))$ (« le coût des étages est constant »).

12. Le programme officiel est d'accord avec moi, et l'a mis hors-programme pour cette raison.

2.1 Complexité spatiale des fonctions récursives

Rappel du cours sur l'organisation de la mémoire : les variables non-allouées sont créées au début de leur bloc et supprimées à la fin.

En particulier, n'existe à tout moment en mémoire que les variables des appels récursifs qui ont commencé mais n'ont pas encore terminé. Dans l'arbre d'appels, commencer un appel revient à « descendre en suivant un trait » et terminer un appel revient à « remonter en suivant un trait ». Ainsi :

Proposition 15 (Complexité spatiale récursive).

La complexité spatiale d'une fonction récursive est le maximum d'espace nécessaire pour une suite d'appels de l'appel initial à un cas de base.

Autrement dit, c'est la somme maximale des complexités locales d'appels le long d'un chemin de la racine à une feuille de l'arbre d'appels.

Remarque. Ce « Autrement dit » ignore les difficultés liées au partage de la mémoire lors d'un passage par référence/pointeur que l'on a déjà évoquées.

Exemple. La fonction HANOÏ a une complexité spatiale en (n) . En effet, le coût local de chaque appel est constant, et il y a au plus n appels actifs à tout moment¹³. Comme cette borne sur le nombre d'appels est atteinte, c'est même un $\Theta()$.

Exercice. Calculer la complexité temporelle et spatiale de la fonction `tri_fusion` ci-dessous :

```

15 (** Fusionne deux listes triées en une seule. *)
16 let rec fusionne (l0 : 'a list) (l1 : 'a list) : 'a list =
17   match (l0,l1) with
18   | [], _      -> l1
19   | _, []      -> l0
20   | h0::t0, h1::t1 -> if h0 < h1 then
21                         h0 :: (fusionne t0 l1)
22                         else
23                         h1 :: (fusionne l0 t1)
24
25
26 (** Separe la liste l en deux moitiés *)
27 let rec separe (l : 'a list) : 'a list * 'a list =
28   match l with
29   | []      -> [], []
30   | h :: [] -> [h], []
31   | h::hbis :: t -> let (u,v) = separe t in
32                     h::u, hbis::v
33
34 (** Trie l grâce à l'algorithme du tri fusion *)
35 let rec tri_fusion (l : 'a list) : 'a list =
36   match l with
37   | []      -> []
38   | h :: [] -> [h]
39   | _      -> let (u,v) = separe l in
40               fusionne (tri_fusion u) (tri_fusion v)

```



13. Cette preuve est plus convainquante avec un arbre d'appels dessiné. Je dirais même que la preuve est incomplète sans.

3 Complexité amortie

Jusqu'à présent, nous avons étudié la complexité d'un appel à une fonction. Le but désormais est d'étudier le coût d'une *suite* d'appels, et plus précisément le coût d'un appel au sein de cette suite.

Considérons l'exemple initial suivant. Un-e MP2I veut aller faire des séances de sport dans une salle. Le tarif¹⁴ est le suivant :

- Si c'est la première fois, les frais de dossiers coutent 30€. La place coûte 10€ de plus.
- Les fois suivantes, la place coûte uniquement 10€.

Question : dans une succession de séances, combien coûte chaque séance ? On peut dire que dans le pire des cas, chaque séance coûte 40€... mais on veut bien que c'est grossier : dans une succession de séances, les frais de dossiers ne sont payés qu'une seule fois. On préfère dire que dans une succession de S séances, le coût de chaque séance est $10 + \frac{30}{S}$. On *amortit* le coût de l'abonnement sur toutes les séances (on dit aussi qu'on *lisse* le coût).

Définition 16 (Complexité amortie).

On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1. Un coût amorti est la donnée de coûts fictifs \hat{C}_i tels que :

$$\sum_{i=1}^n \hat{C}_i \geq \sum_{i=1}^n C_i$$

Autrement dit, la seule condition sur un coût amorti est que la somme des coûts amorti majore le coût réel.

3.0 Méthodes de calcul

3.0.0 Exemple fil rouge

Nous allons présenter trois méthodes qui cherchent à calculer un coût amorti qui soit une majoration assez précise. Nous les appliquerons à un exemple fil rouge : les tableaux dynamiques. Initialement, un tableau dynamique est un tableau à 1 case, qui n'est pas utilisé : nous appellerons. On ajoute un élément à la fin de celui-ci via la fonction ci-dessous :

Fonction Ajout

Entrées : T un tableau dynamique ; x un élément à ajouter à la fin

```

1 si T est entièrement plein alors
2   T' ← nouveau tableau deux fois plus grand que T
3   recopier T dans T'
4   remplacer T par T'
5 len ← nombre de cases utilisées de T
6 T[len] ← x
```

L'exemple fil rouge sera donc : calculer un coût amorti d'un appel à AJOUT au sein d'une suite d'appels à AJOUT. On comptera le coût en nombre d'écriture dans un tableau effectuées.

3.0.1 Méthode du comptable

On imagine qu'au lieu de dépenser du temps ou de l'espace (ou une autre ressource), la fonction dépense des *pièces*. L'objectif de la méthode du comptable est de montrer comment pré-payer des opérations coûteuses en augmentant le prix des opérations peu coûteuses. On amortit ainsi le coût des opérations coûteuses sur les autres.

14. Fictif.

Définition 17 (Méthode du comptable).

On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1.

La **méthode du comptable** est le fait d'ajouter à chaque opération une **accumulation** a_i et une **dépense** d_i . L'accumulation consiste à poser une ou plusieurs pièces « sur » la structure, en prévision d'un paiement élevé plus tard. Une dépense consiste à utiliser des pièces déjà déposées pour payer une opération coûteuse. On doit garantir qu'à tout moment :

$$\sum_{i=1}^n a_i \geq \sum_{i=1}^n d_i$$

C'est à dire qu'on ne dépense jamais une pièce que l'on a pas encore accumulée. La méthode du comptable définit alors le coût amorti suivant :

$$\hat{C}_i = C_i + a_i - d_i$$

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned} \sum_{i=1}^n \hat{C}_i &= \sum_{i=1}^n (C_i + a_i - d_i) \\ &= \left(\sum_{i=1}^n C_i \right) + \left(\sum_{i=1}^n a_i \right) - \left(\sum_{i=1}^n d_i \right) \\ &\geq \sum_{i=1}^n C_i \end{aligned} \quad \text{car } \sum_{i=1}^n a_i \geq \sum_{i=1}^n d_i$$

Il s'agit donc bien d'un coût amorti. □

Remarque. En pratique, il faut donner les accumulations, les dépenses, et une justification convainquante du fait que l'on ne dépense pas plus que ce que l'on a accumulé.

Exemple. Calculons un coût amorti pour un AJOUT au sein d'une suite d'AJOUT sur un tableau dynamique par la méthode du comptable. On fixe les accumulations et dépenses ainsi :

- Si l'appel à AJOUT n'a pas besoin de doubler le tableau, on paye 1 pièce (pour l'écriture de x), et on stocke 2 autres pièces sur la case que l'on vient d'écrire, afin d'être plus tard capable de payer un doublement du tableau.

Ainsi, toutes les cases qui ont été écrites depuis la dernière fois que le tableau a été doublé contiennent 2 pièces posées sur elles. Autrement dit, toutes les cases écrites de la deuxième moitié du tableau contiennent 2 pièces accumulées.

Le coût amorti pour un tel appel à AJOUT est donc $\hat{C}_i = 3$.

- Sinon, il faut payer 1 (pour écrire x) et en plus payer la recopie des cases, donc payer autant de cases que le tableau en a. On va pour cela dépenser les 2 pièces accumulées sur chacune des cases de la deuxième moitié du tableau ! En dépensant ce stock, le coût de la recopie est entièrement couvert. On accumule enfin 2 pièces sur la nouvelle case, pour les mêmes raisons que précédemment.

Le coût amorti pour un tel appel à AJOUT est donc $\hat{C}_i = 3$.

On a ainsi prouvé par la méthode du comptable qu'au sein d'une suite d'appels à AJOUT, un appel à AJOUT a un coût amorti $\hat{C}_i = 3$.

FIGURE 3.5 – Schéma de l'évolution du tableau dynamique et de l'accumulation de la méthode du comptable

Remarque.

- Dans la preuve précédente, on n'a pas toujours explicitement indiqué les valeurs de d_i , C_i et a_i . Je vous conseille de les indiquer si vous avez besoin de repères clairs pour avancer votre preuve (après tout, le choix de bons a_i et d_i sont tout l'enjeu de la méthode), mais de ne pas les mettre s'ils ne feraient qu'alourdir la rédaction. Par contre, ils ne doivent jamais être ambigus : on doit pouvoir comprendre quelle serait leur valeur !
- La méthode du comptable s'applique quand on arrive à bien prévoir quelle opération peut prépayer quelle autre opération. Ici, lors de l'ajout d'un élément on pré-paye pour la recopie future de cet élément ainsi que pour la recopie d'un élément de la première moitié du tableau.

3.0.2 Méthode du potentiel

La méthode du potentiel revient à faire accumuler et dépenser un potentiel « global », au lieu de petits stocks locaux de pièces. La différence est que l'on définit le potentiel par une formule (à trouver, qui doit répondre à nos besoins), et que c'est de cette formule que l'on déduit les accumulations/dépenses.

Définition 18 (Méthode du potentiel).

On considère une structure de données sur laquelle on applique une succession de n opérations. On note C_i les coûts réels de chacun de ces opérations successives. On numérote les opérations à partir de 1.

La **méthode du potentiel** est le fait d'associer à chaque état de la structure un potentiel ($P(0)$ est le potentiel initial, $P(1)$ le potentiel après la première opération, etc). On doit garantir que :

$$\forall i, P(i) \geq P(0)$$

La méthode du potentiel définit alors le coût amorti suivant :

$$\hat{C}_i = C_i + P(i) - P(i-1)$$

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned}
\sum_{i=1}^n \hat{C}_i &= \sum_{i=1}^n (C_i + P(i) - P(i-1)) \\
&= \left(\sum_{i=1}^n C_i \right) + \left(\sum_{i=1}^n P(i) - P(i-1) \right) \\
&= \left(\sum_{i=1}^n C_i \right) + P(n) - P(0) \\
&\geq \sum_{i=1}^n C_i \quad \text{car } P(n) \geq P(0)
\end{aligned}$$

Il s'agit donc bien d'un coût amorti. □

Remarque. Toute la difficulté est de trouver un bon potentiel. On veut une fonction qui :

- Augmente un peu lorsque l'on fait une opération peu coûteuse, pour « accumuler du potentiel ».
- Décroit d'un coup lors d'une opération coûteuse. Cette décroissance du potentiel va compenser le coût de l'opération : elle revient à « dépenser » le potentiel accumulé.

FIGURE 3.6 – Évolution typique d'un bon potentiel

Exemple. Calculons un coût amorti pour un AJOUT au sein d'une suite d'AJOUT sur un tableau dynamique (initialement vide) par la méthode du potentiel. Nommons E_0, \dots les états successifs du tableau, et numérotions les opérations à partir de 1 : ainsi, on passe de l'état E_{i-1} à l'état E_i par l'opération numéro i .

Un état E_i du tableau dynamique contient len_i le nombre de cases utilisées dans le tableau, ainsi que $dispo_i$ le nombre de cases dont le tableau dispose vraiment. On définit le potentiel suivant :

$$P(i) = 2len_i - dispo_i$$

Calculons le coût amorti associé :

- Si l'appel à AJOUT n'a pas besoin de doubler le tableau, le coût réel est 1 (on écrit x). Calculons la différence $P(i) - P(i-1)$. Comme le tableau n'a pas été doublé, $dispo_i = dispo_{i-1}$ et donc $P(i) - P(i-1) = 2(len_i - len_{i-1}) = 2$.

Le coût amorti pour un tel appel à AJOUT est donc $\hat{C}_i = 3$.

- Sinon, il faut payer 1 (pour écrire x) et en plus payer la recopie de toutes les cases du tableau, donc un coût total de $1 + dispo_{i-1}$. Comme le tableau a été doublé, $dispo_i = 2dispo_{i-1}$ et $len_{i-1} = dispo_{i-1}$. On a donc :

$$\begin{aligned}
P(i) - P(i-1) &= dispo_{i-1} - dispo_i + 2(len_i - len_{i-1}) \\
&= dispo_{i-1} - 2dispo_{i-1} + 2 \\
&= 2 - dispo_{i-1}
\end{aligned}$$

Le coût amorti pour un tel appel est $1 + \text{dispo}_{i-1} + 2 - \text{dispo}_{i-1}$, donc : $\hat{C}_i = 3$.

On a ainsi prouvé par la méthode du potentiel qu'au sein d'une suite d'appels à AJOUT, un appel à AJOUT a un coût amorti $\hat{C}_i = 3$.

Remarque. La méthode du potentiel s'applique lorsque l'on n'arrive pas à prévoir quelle opération pré-paye laquelle. À la place, on cherche comment doit évoluer le stock global de « pièces » et ce stock global que l'on définit.

3.0.3 Méthode de l'aggrégat

La méthode de l'aggrégat est plus mathématique. Elle consiste à calculer la somme $\sum_{i=1}^n C_i$ en la décomposant en « paquets » simples à calculer. On en déduit le coût total de la suite d'opérations, que l'on peut ensuite lisser sur les opérations.

Définition 19 (Méthode de l'aggrégat).

La **méthode de l'aggrégat** consiste à poser $\hat{C}_i = \frac{1}{n} \sum_{i=1}^n C_i$.

Toute la difficulté est donc d'avoir réussi à calculer cette somme. On peut utiliser une majoration très précise à la place de la somme.

Démonstration. Prouvons que le coût amorti proposé est bel et bien un coût amorti. On a :

$$\begin{aligned} \sum_{i=0}^{n-1} \hat{C}_i &= \sum_{i=0}^{n-1} \left(\frac{1}{n} \sum_{j=0}^{n-1} C_j \right) \\ &= n \left(\frac{1}{n} \sum_{j=0}^{n-1} C_j \right) && \text{car le contenu de la seconde somme ne dépend pas de } i \\ &= \sum_{j=0}^{n-1} C_j \end{aligned}$$

Il s'agit donc bien d'un coût amorti. □

Exemple. Calculons un coût amorti pour un AJOUT au sein d'une suite d'AJOUT sur un tableau dynamique (initialement vide) par la méthode de l'aggrégat. On numérote les opérations à partir de 1. Par une récurrence immédiate, on montre qu'après l'AJOUT numéro i , le tableau contient i valeurs. On montre de même que l'on double le tableau lorsque $i - 1$ est une puissance de 2, et que dans ce cas on recopie toutes les valeurs donc $i - 1$ valeurs.

Le coût C_i d'une opération est 1 (écriture de x), plus éventuellement le doublement. Donc :

$$C_i = \begin{cases} 1 + i - 1 & \text{si } i - 1 = 2^p \text{ avec } p \in \mathbb{N} \\ 1 & \text{sinon} \end{cases}$$

Donc :

$$\begin{aligned} \sum_{i=1}^n C_i &= \sum_{i=1}^n 1 + (i - 1) \cdot \mathbb{1}_{\langle i-1 \text{ est de la forme } 2^p \rangle} \\ &= n + \sum_{p=0}^{\lfloor \log_2 n \rfloor} 2^p \\ &= n + 2^{\lfloor \log_2 n \rfloor + 1} - 1 \\ &\leq 3n \end{aligned}$$

La méthode de l'aggrégat propose alors $\hat{C}_i = \frac{3n}{n} = 3$ comme coût amorti.

Remarque. Cette méthode est à utiliser quand on connaît une propriété de la structure qui permet de faire des « paquets » agréables, ou que l'on arrive à réorganiser la somme d'une façon qui lui donne plus de sens.

3.1 Remarques et compléments

- **L'une des applications classiques de la complexité amortie est le calcul de la complexité d'une boucle qui opère à chaque itération sur une structure de données.** Voici un exemple en Python, où `append` correspond à `AJOUT` :

```

1 lst_premiers = []
2 for x in range(0, n): # pour x allant de 0 inclus à n exclu
3     if is_prime(x):
4         lst_premiers.append(x)

```



Le calcul de la complexité de cette boucle est très simple si on utilise l'analyse amortie pour affirmer que l'on peut faire comme si chacun des `append` est en $O(1)$ (puisque l'on effectue une succession de `append` sur une même liste), et assez compliqué sinon.

- **Il ne faut pas hésiter à faire un schéma** pour illustrer la structure, les accumulations/dépenses ou le potentiel. Un bon schéma permet d'écrire ensuite une preuve plus concise et plus claire.¹⁵
- Les trois méthodes de calcul reviennent finalement à trouver une façon de majorer précisément une somme. Elles sont cependant expliquées de manière différente. Prenez la méthode qui vous parle le plus (sauf si dans votre situation précise une des méthodes est clairement plus simple que les autres).
- On cherche surtout à majorer et donc à calculer un $O()$. On pourrait adapter les raisonnements pour minorer et obtenir un $\Omega()$ ou un $\Theta()$, mais c'est plus rare.
- **L'état initial de la structure étudiée n'a pas beaucoup d'importance.** Quitte à majorer, on peut ajouter des opérations initiales « fictives » qui amènent l'état initial dans un état plus agréable, sans que cela ne change l'ordre de grandeur du résultat. En effet, les calculs d'un coût amorti lissent des coûts sur un nombre très grands d'opération. Les opérations fictives auront un coût asymptotiquement nul au sein de ce très grand nombre d'opération¹⁶.
- **Un coût amorti n'a de sens qu'au sein de la suite d'opérations où on l'a calculé.** On a prouvé dans ce cours qu'*au sein d'une suite d'AJOUT*, chaque `AJOUT` a un coût amorti constant. Mais on pourrait imaginer une nouvelle opération, `DÉMON` qui remplit le tableau dynamique jusqu'à ce que son nombre d'éléments soit une puissance de 2. Avec cette nouvelle opération, chacun des `AJOUT` va doubler le tableau ; et donc le meilleur coût amorti possible pour `AJOUT` au sein d'une suite d'opérations qui alterne `AJOUT` et `DÉMON` est un coût linéaire en la longueur du tableau.
- **Un coût amorti n'a rien à voir avec la complexité moyenne**¹⁷. Ce n'est juste pas la même définition ! Un coût amorti est défini sur *une suite d'opérations*, la complexité moyenne sur *une* opération (mais en moyennant sur les entrées possibles pour cette seule opération).

15. Ce conseil s'applique à l'entiereté de l'informatique (et de la plupart des sciences). Aidez à visualiser !

16. Mathématiquement, on dit que la somme partielle d'une série qui diverge vers $+\infty$ est négligeable devant son reste.

17. J'ai d'ailleurs fait très attention à ne jamais utiliser le mot « moyenne » dans toute la section sur la complexité amortie.

Chapitre 4

RÉCURSIVITÉ

Notions	Commentaires
Réversibilité d'une fonction. Réversibilité croisée. Organisation des activations sous forme d'arbre en cas d'appels multiples.	La capacité d'un programme à faire appel à lui-même est un concept primordial en informatique. [...] On se limite à une présentation pratique de la récursivité comme technique de programmation.

Extrait de la section 2 du programme officiel de MP2I : « Récursivité et induction ».

SOMMAIRE

0. Introduction	72
0. Notion de réduction	72
1. Notion de récursion	72
1. Exemples	73
0. Affichage d'un triangle	73
1. Tours de Hanoï	74
2. Exponentiation rapide	76
3. À propos des boucles	77
<i>Transformation boucle <-> récursion (p. 77). Récursivité terminale (hors-programme) (p. 78).</i>	
2. Comment concevoir une fonction récursive	78
3. Analyse de fonctions récursives.....	79
4. Compléments	79
0. Élimination des appels redondants	79
1. Avantages et inconvénients de la récursion	79
2. Querelle sémantique	80
3. Quand « déplier les appels » ?	80

0 Introduction

0.0 Notion de réduction

Rappels du cours introductif : Un **problème** est composé de deux éléments : la description d'une **instance** (une « entrée »), et une question/tâche à réaliser sur cette instance.

Définition 1 (Réduction).

On dit qu'une instance d'un problème A se *réduit* à une instance d'un problème B si résoudre cette instance de B suffit à résoudre cette instance de A .

On dit que le problème A se réduit au problème B si toute instance de A se réduit à une instance de B . On note parfois $A \preceq B$.

Remarque.

- Dit autrement, il s'agit d'utiliser le fait que l'on sait déjà résoudre un problème B afin de résoudre un problème A . C'est l'une des méthodes courantes en science : se ramener à ce que l'on sait faire !
- Vous approfondirez cette notion en MPI, notamment en imposant des conditions sur les liens entre les instances.
- Notez qu'il n'y a pas besoin de savoir *comment* B est résolu. D'un point de vue pratique, il suffit d'avoir accès à une fonction qui résout B (sans même savoir comment elle marche) afin de résoudre A .

Exemple.

- Posons A le problème de calculer le PGCD de deux entiers (une instance est la donnée de deux entiers), et B le problème de calculer le PPCM de deux entiers.
Alors A se réduit à B . En effet, soit (x, y) une instance de A . Utilisons (x, y) comme une instance de B . Or, d'après le cours de maths, on sait que $|xy| = \text{pgcd}(x, y) \cdot \text{ppcm}(x, y)$ et donc que $\text{pgcd}(x, y) = \frac{|xy|}{\text{ppcm}(x, y)}$.
Ainsi, résoudre l'instance (x, y) de B suffit à calculer $\text{pgcd}(x, y)$ et donc à résoudre l'instance (x, y) de A . Il s'ensuit que A se réduit à B .
- On peut montrer que réciproquement, PPCM se réduit à PGCD.

0.1 Notion de récursion

L'idée centrale de la récursion est de réduire une instance d'un problème à une autre instance, plus petite, de ce même problème.

Exemple.

- Le calcul d'une suite récurrente en maths : si par exemple $u_{n+1} = f(u_n)$, alors l'instance « calculer le terme $n + 1$ » se réduit à l'instance « calculer le terme n ».
- Le calcul des coefficients binomiaux : pour calculer $\binom{n}{k}$, d'après le cours de maths il suffit de calculer $\binom{n-1}{k}$ et $\binom{n-1}{k-1}$.
- Dessiner une fractale : une fractale est un dessin qui se répète dans lui-même. Ainsi, pour dessiner une fractale « grande », il faut commencer par dessiner des fractales plus « petites » dedans.

Exercice. Montrez que le problème de trouver le minimum d'un tableau se réduit au problème de trier un tableau.

Cette méthode de résolution de problèmes se généralise à l'écriture des fonctions :

Définition 2 (Fonction récursive).

Une **fonction récursive** est une fonction qui peut faire un ou plusieurs appels à elle-même. Le contenu d'une fonction récursive est composée de :

- Cas récursif : les appels que la fonction fait à elle-même, et ce qu'elle en fait.
- Cas de base : ce que fait la fonction quand elle ne peut pas s'appeler elle-même.

Exemple.

```

9  /** Renvoie x^n. */
10 double exp_naif(double x, unsigned n) {
11     if (n > 0) {
12         return x * exp_naif(x, n-1);
13     } else { // n == 0
14         return 1;
15     }
16 }
```

exemples.c

La fonction ci-contre calcule x^n (avec $n \geq 0$). Pour cela :

$$x^n = \begin{cases} x \cdot x^{n-1} & \text{si } n > 0 \\ 1 & \text{si } n = 0 \end{cases}$$

On peut représenter visuellement le déroulement de cette fonction sur des exemples :

(a) `exp_naif(1.5, 4)`

(b) `exp_naif(2.0, 5)`

FIGURE 4.1 – Visualisation d'appels à `exp_naif`

1 Exemples

1.0 Affichage d'un triangle

Écrivons une fonction qui affiche un triangle de n lignes d'étoiles, comme ceci :

```

***
**
*
```

(a) Triangle de $n=3$ étoiles

```

*****
****
***
**
*
```

(b) Triangle de $n = 5$

FIGURE 4.2 – Des triangles

Essayons d'écrire une fonction récursive pour ce problème. Pour cela, cherchons une « structure récursive » dans le problème.

On remarque que le triangle $n-1$ est inclus dans le triangle n :

```
*****
****
***
**
*
```

FIGURE 4.3 – Inclusion des triangles

On en déduit la solution récursive suivante :

Fonction Triangle

Entrées : $n \geq 0$

```
1 si n > 0 alors
2   Afficher une ligne de n étoiles
3   Triangle(n-1)
```

Remarque. Pour bien comprendre cette fonction, il faut la faire tourner à la main.

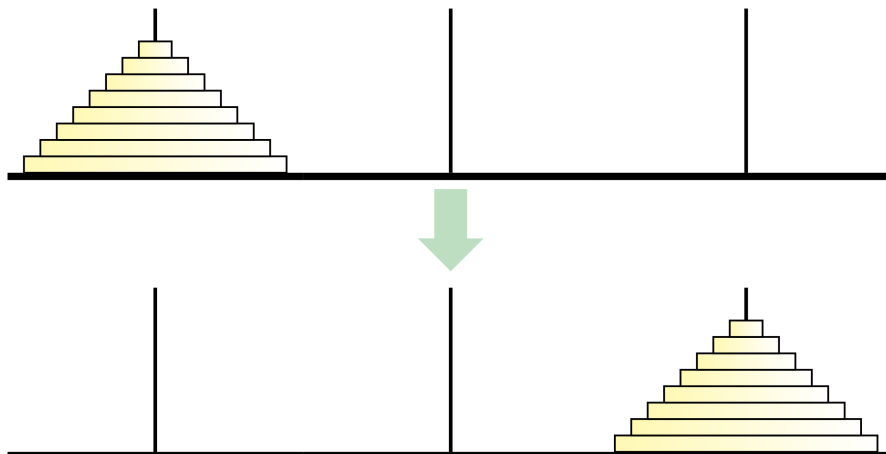
Exercice. Écrire cette fonction en C, et la tester. *Il n'y a aucune syntaxe particulière à utiliser pour la récursivité en C.*

Exercice. Faire de même mais avec un triangle "pointe en haut".

1.1 Tours de Hanoï

Les tours de Hanoï sont un casse-tête très célèbre :

- On dispose de 3 tiges (emplacements), sur lesquelles se trouvent des disques, empilés les uns sur les autres.
- Il y a un total de n disques. Ils sont caractérisés par leur diamètre, qui sont deux à deux distincts. Un disque ne peut jamais reposer sur un disque plus petit.
- On peut déplacer le disque qui est au sommet d'une pile en le plaçant au sommet d'une autre (en respectant la règle des diamètres!).
- Initialement, tous les disques sont sur une même tige. Le but est de tous les déplacer sur une autre des 3 tiges.

FIGURE 4.4 – Situation initiale et finale des tours de Hanoï avec $n=8$ disques (src : *Algorithms*, J. Erikson)

Exemple. Voici une succession de déplacements qui résout les tours de Hanoï à 3 disques :

FIGURE 4.5 – Résolution de Hanoï à 3 disques

On veut écrire une solution récursive au problème : on veut trouver une méthode pour déplacer n disques d'une tige vers une autre. Pour cela, **on suppose que l'on a des camarades très intelligent-es qui savent déjà le résoudre pour des instances plus petites**. L'objectif est de réussir à utiliser leur aide pour résoudre notre instance.

En bidouillant un peu, on réalise qu'une étape intermédiaire inévitable dans toute solution est de placer le plus gros disque (celui du fond de la pile de départ) sur la tige d'arrivée. Il faut donc réussir à enlever les autres disques d'au-dessus de lui, et comme il est plus gros que les autres il faut que la tige d'arrivée soit vide quand on le place. Autrement dit, il faut déplacer les $n-1$ premiers disques sur la tige qui n'est ni départ ni arrivée.

...¹

Mais en fait « déplacer $n-1$ premiers disques d'une tige à une autre », c'est exactement une autre instance du problème ! Nos camarades peuvent donc le résoudre ! Il nous suffit ensuite de déplacer le gros disque sur la tige d'arrivée, puis de re-déplacer les $n-1$ premiers.

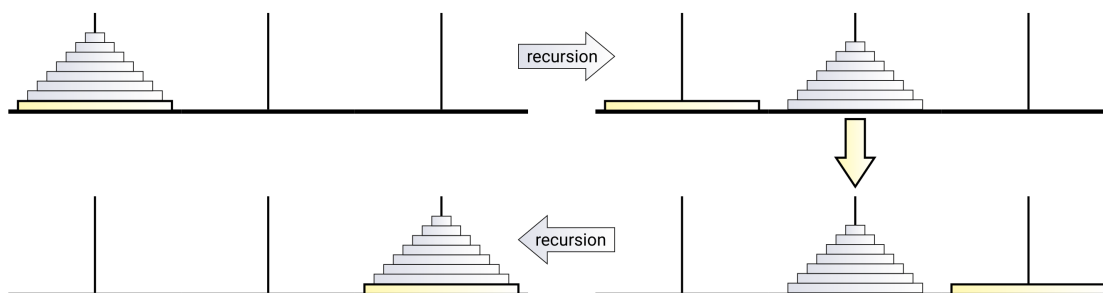


FIGURE 4.6 – Le fonctionnement récursif de l'algorithme (src : *Algorithms*, J. Erikson)

1. Insérer roulements de tambours dramatiques.

On a arrêté de réfléchir. On ne cherche SURTOUT PAS à « déplier » les appels récursifs : nos camarades sont très intelligent-es et on leur fait confiance !!!!!!!!!!!

Il reste une subtilité : quand $n = 0$, l'étape « déplacer $n - 1$ disques » n'a pas de sens. Dans ce cas, on ne *peut pas* appliquer notre méthode récursive et il faut résoudre cette instance par nous-même. Heureusement, déplacer $n = 0$ disques n'est pas trop compliqué... On obtient donc le pseudo-code suivant :

Fonction Hanoï

Entrées : i : la tige de départ; j : la tige d'arrivée; n : le nombre de disques à déplacer

```

1 si  $n > 0$  alors
2    $k \leftarrow$  tige autre que  $i$  ou  $j$ 
3   HANOÏ( $i$ ,  $k$ ,  $n-1$ )
4   Déplacer 1 disque de  $i$  à  $j$ 
5   HANOÏ( $k$ ,  $j$ ,  $n-1$ )
```

Exemple. En fait, la résolution précédente de Hanoï à 3 disques était déjà une application de cet algorithme !

Exercice. Appliquer cet algorithme récursif pour $n = 4$. *Conseil :* prévoyez beaucoup de place, et notez les appels récursifs au fur et à mesure; votre mémoire ne suffira pas.

Remarque.

- J'insiste : pour trouver cette solution, nous n'avons **pas** cherché à comprendre comment les camarades intelligent-es procèdent !! Les camarades / appels récursifs sont des « boîtes noires » qui résolvent les autres instances, sans que l'on ait à comprendre comment.
- De manière cachée, nous avons commencé par **généraliser** : on n'a pas directement résolu le problème de « déplacer n tiges du départ à l'arrivée », on a résolu le problème de « déplacer les *plus petits* disques d'une tige (où ils doivent tous être) vers une autre ».
- En particulier, ce qui fait marcher cet algorithme est que l'on essaye toujours de déplacer les disques les plus petits : même si on ne le pose pas sur des tiges vides (la tige intermédiaire est rarement vide), on les pose sur des disques plus gros : tout va bien.
- Il est formateur d'essayer d'analyser la complexité de cet algorithme : cf cours complexité.

1.2 Exponentiation rapide

La fonction `exp_naïf` précédente est une implémentation de la fonction $(x, n) \mapsto x^n$. Si on compte sa complexité en nombre de multiplications $M(n)$, on obtient l'équation $M(n) = 1 + M(n-1)$ et $M(0) = 0$, donc $M(n) = n$.

Cela peut sembler bien, mais c'est en fait assez mauvais². Il existe une méthode plus maline, basée sur la remarque suivante : notons $p = \lfloor \frac{n}{2} \rfloor$ et utilisons :

$$x^n = \begin{cases} x \cdot x^p \cdot x^p & \text{si } n = 2p + 1 \text{ et } n > 0, \text{ c'est à dire si } n \text{ est impair et } > 0 \\ x^p \cdot x^p & \text{si } n = 2p \text{ et } n > 0, \text{ c'est à dire si } n \text{ est pair et } > 0 \\ 1 & \text{sinon, c'est à dire si } n = 0 \end{cases}$$

2. Vous en parlerez plus en MPI, mais faisons un brin de hors-programme : la complexité « pertinente » s'exprime en fonction de la taille des entrées. n s'écrit sur $\log_2 n$ bits, donc une complexité en $\Theta(n)$ est exponentielle en la taille de l'entrée. Et l'exponentiel, c'est beaucoup.

On obtient le code suivant :

```

22  /** Renvoie x^n. */
23  double exp_rap(double x, unsigned n) {
24      if (n > 0) {
25          int x_p = exp_rap(x, n/2); // x puissance p avec p = n/2
26          if (n % 2 == 0) { return x_p * x_p; }
27          else { return x * x_p * x_p; }
28      }
29      else {
30          return 1;
31      }
32  }

```



Ou en OCaml :

```

3  (** Renvoie x^n (avec n >= 0) *)
4  let rec exp_rap x n =
5      if n > 0 then
6          let x_p = exp_rap x (n/2) in (* x puissance p avec p = n/2 *)
7          if n mod 2 = 0 then x_p *. x_p else x *. x_p *. x_p
8      else
9          1.

```



La complexité $M(n)$ de cet algorithme-ci vérifie $M(n) \leq 2 + M(n/2)$ et $M(0) = 0$. On en déduit³ $M(n) = O(\log_2 n)$, ce qui est *bien* mieux que la complexité précédente !

Remarque.

- On être plus précis sur la complexité de cet algorithme. Notons N_0 le nombre de 0 dans l'écriture binaire de n , et N_1 celui de 1 (on a en particulier $N_0 + N_1 = \log_2 n$).

Avec ces notations, on peut⁴ montrer que $M(n) = N_0 + 2N_1$ qui est bien un $O(\log_2 n)$.

- On peut prouver qu'il faut $\Omega(\log_2 n)$ opérations élémentaires. En effet, on peut prouver qu'il *faut* lire chacun des bits de n .

Pour cela, raisonnons par l'absurde et supposons qu'il existe un algorithme totalement correct qui fonctionne sans lire tous les bits de l'entrée n . Soit $n \in \mathbb{N}$ tels qu'un bit de n ne soit pas lu. Notons m l'entier obtenu en changeant la valeur de ce bit non lu. Alors l'algorithme renvoie la même réponse sur x^m et x^n ... mais ces deux valeurs sont distinctes⁵ : l'algorithme n'est donc pas correct, absurde.

Il faut donc lire chacun des bits de n , et donc effectuer au moins $\Omega(\log_2 n)$ opérations élémentaires. Sachant cela, effectuer $O(\log_2 n)$ multiplications est très satisfaisant.

⚠ Attention, une erreur commune avec cet algorithme est d'effectuer deux appels récursifs à chaque fois (ce que l'on évite dans les codes ci-dessus en stockant le résultat de l'appel dans une variable). Si on fait cela, la complexité redevient⁶ $O(n)$.

1.3 À propos des boucles

1.3.0 Transformation boucle <-> récursion

La plupart des boucles peuvent être réécrites par une fonction récursive. L'idée est de faire une fonction qui « fait une itération », puis qui s'appelle elle-même pour faire la prochaine itération. Ainsi, les deux codes ci-dessous sont équivalents :

3. Pour cela, remarquer que $M(1) = 2$ simplifie la manipulation des log.

4. Il faut étudier l'écriture binaire de n durant la suite d'appels récursive, et utiliser le fait que le dernier bit indique la parité

5. Sauf pour $x = 0$ ou $|x| = 1$, d'accord.

6. Utiliser la méthode de calcul de complexité par arbre.

```

34
35 /** Somme des entiers jusqu'à n
   ↳ inclus. */
36 int somme_entiers(int n) {
37     int i = 0;
38     int somme = 0;
39     while (i <= n) {
40         somme = somme + i;
41         i = i + 1;
42     }
43     return i;
44 }

```

```

12
13 (** Somme des entiers jusqu'à n
   ↳ inclus *)
14 let somme_entiers n =
15     let rec boucle somme i =
16         if i <= n then
17             boucle (somme+i) (i+1)
18         else
19             somme
20     in
21     boucle 0 0

```

Ici, la boucle C a été transformée en la fonction boucle. Dans la boucle C, on a $\text{somme}' = \text{somme} + i$ et $i' = i + 1$. La version récursive fonctionne en calculant ces valeurs prime (donc en « simulant une itération »), puis en s'appelant elle-même pour simuler les itérations suivantes.

C'est une transformation classique, que l'on fera souvent lorsque l'on travaille en OCaml.

Remarque. La transformation inverse (récursion \rightarrow boucle) est possible, mais plus technique lorsqu'il y a plusieurs appels récursifs (il faut simuler la pile d'appels à la main).

1.3.1 Récursivité terminale (hors-programme)

Si une fonction récursive s'appelle *une seule fois*, et que cet appel est *la toute dernière chose* effectuée, alors il est très simple de transformer la fonction récursive en boucle. C'est une optimisation faite par le compilateur OCaml⁷ !

Cela permet de gagner :

- un peu de temps, car passer d'une itération d'une boucle à la suivante est plus rapide que d'ouvrir un appel récursif
- beaucoup d'espace, car il n'y a pas besoin d'utiliser de l'espace pour chacun des appels successifs.

⚠ Écrire une fonction récursive terminale demande parfois de modifier le code de manière peu lisible. Il faut d'abord faire une version lisible et correcte et ensuite seulement une version optimisée.

2 Comment concevoir une fonction récursive

Proposition 3 (Concevoir une solution récursive).

Pour concevoir une solution récursive à un problème sur une instance I :

- On suppose que l'on a une boîte noire magique (la récursion) qui peut résoudre toutes les instances du problème sauf I .
- On cherche un lien entre I et une ou plusieurs autres instances du problème. Souvent, cela implique de « reconnaître le problème dans le problème » (comme pour les triangles) ou de « exprimer $f(n)$ à l'aide de $f(n-1)$ ».
- On cherche les cas où le lien trouvé ne peut pas être utilisé : ce sont les cas de base, et on doit en trouver une solution sans récurrence.
- On vérifie que les instances « diminuent » d'un appel sur l'autre, afin de garantir que la suite d'appel finit par atteindre un cas de base et termine.

7. Et que gcc peut faire si on le lui demande, et que cela n'entre pas en conflit avec d'autres optimisations.

3 Analyse de fonctions récursives

Les méthodes pour analyser la terminaison, la correction et la complexité d'une fonction récursive sont dans les cours en question.

4 Compléments

4.0 Élimination des appels redondants

Lorsque l'on écrit des fonctions récursives, il arrive que certaines instances plus petites soient résolues plusieurs fois. Voici par exemple une fonction (très peu recommandée) pour calculer $\binom{n}{k}$:

```

24 (** Renvoie k parmi n *)
25 let rec binom n k =
26   if k = 0 || k = n then
27     1
28   else
29     ( binom (n-1) k ) + ( binom (n-1) (k-1) )

```

 exemples.ml

Si on applique cette méthode et que l'on essaye par exemple de calculer $\binom{5}{2}$, certains appels sont calculés plusieurs fois :

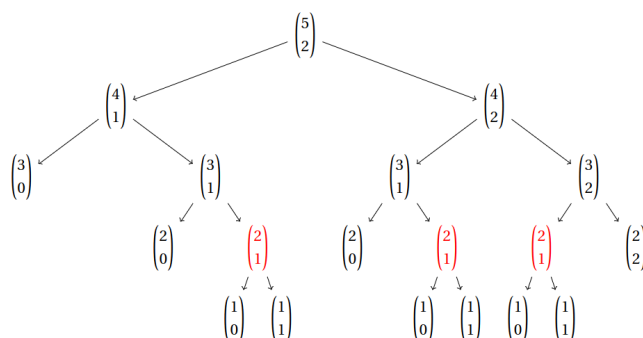


FIGURE 4.7 – Arbre d'appels de `binom 5 2`. Une redondance d'appels est mise en exergue en rouge. (src : J. Larochette)

Ces appels redondants ralentissent le fonctionnement, puisque l'on recalcule plusieurs fois la même chose. Il est plus efficace de mémoriser ces valeurs pour ne pas avoir à les recalculer ensuite : c'est l'objet de la programmation dynamique, que nous verrons au second semestre.

4.1 Avantages et inconvénients de la récursion

Avantages. La récursivité :

- peut produire un code plus lisible et plus simple à comprendre.
- produit un code souvent plus simple à prouver (car les hypothèses de récurrence / invariants sont plus simples).
- permet de résoudre des problèmes très difficiles à résoudre sans !!
- est très adaptée au travail sur les structures de données naturellement récursives (dont les arbres).
- est très adaptée aux problèmes naturellement récursifs.

Inconvénients. La récursivité :

- est plus éloignée du fonctionnement de la machine, et est ralentie par le temps nécessaire à l'ouverture d'un appel de fonction.
- utilise de l'espace mémoire en empilant les appels.
- produit *parfois* un code moins clair qu'un code itératif⁸ plus direct.

En résumé, c'est un outil extrêmement puissant. Comme tout outil, il n'est pas absolu et a ses limites d'utilisation. Comme tout bon outil, il rend simple des tâches difficiles. Un grand pouvoir implique de grandes responsabilités !

4.2 Querelle sémantique

Différents termes existent pour parler de la récursivité :

- « récursivité », « fonction récursive » sont les termes usuels en français.
- « récurrence » est également utilisé (c'est de toute façon la même notion qu'en mathématiques).
- « récursion » est une anglicisme du terme anglais « recursion », souvent utilisé.
- « induction », « fonction inductive » sont des anglicismes venant du terme anglais « induction », qui signifie « récurrence ».

À titre personnel, j'utiliserai et mélangerai toutes ces versions. Je connais des auteurs/autrices qui restreignent certains termes⁹ : adaptez-vous à ce que vous dit la personne avec qui vous interagissez.

4.3 Quand « déplier les appels » ?

J'ai insisté sur ce cours sur l'importance de ne pas « déplier les appels » lorsque l'on conçoit un algorithme récursif. Il y a cependant certains cas où déplier les appels peut-être utile :

- Pour tracer un arbre d'appel, il faut bien déplier un peu.
- Pour déboguer, il peut-être bon de déplier dans sa tête ou à la main les premiers ou les derniers appels. Cela permet de se rendre compte d'erreurs de non-respect de pré-condition, ou du besoin d'une pré-condition.

8. Un code itératif est un code non-récursif.

9. J'ai hésité à le faire aussi, mais je juge qu'à notre modeste niveau, le gain gagné par ces subtilités ne vaut pas l'effort de distinction.

Chapitre 5

STRUCTURES DE DONNÉES (S1)

Notions	Commentaires
Définition d'une structure de données abstraite comme un type muni d'opérations.	On parle de constructeur pour l'initialisation d'une structure, d'accessor pour récupérer une valeur et de transformateur pour modifier l'état de la structure. On montre l'intérêt d'une structure de données abstraite en terme de modularité. On distingue la notion de structure de données abstraite de son implémentation. Plusieurs implémentations concrètes sont interchangeables. La notion de classe et la programmation orientée objet sont hors programme.
Distinction entre structure de données mutable et immuable.	Illustrée en langage OCaml.

Extrait de la section 3.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Structure de liste. Implémentation par un tableau, par des maillons chaînés.	On insiste sur le coût des opérations selon le choix de l'implémentation. Pour l'implémentation par un tableau, on se fixe une taille maximale. On peut évoquer le problème du redimensionnement d'un tableau.
Structure de pile. Structure de file. Implémentation par un tableau, par des maillons chaînés.	

Extrait de la section 3.2 du programme officiel de MP2I : « Structures de données séquentielles ».

SOMMAIRE

0. Tableaux dynamiques	82
0. Tableaux C vs listes OCaml	82
1. Fonctionnement	83
<i>Ajout dans un tableau dynamique (p. 84). Création, suppression (p. 85).</i>	
2. Analyse de complexité	85
3. Complément mi hors-programme : RETRAIT	85
1. Interfaces et types abstraits.....	87
0. Aspects pratiques	87
<i>Librairies (p. 87). Interface (p. 88). Étapes de la compilation (p. 90).</i>	
1. Aspects théoriques	91
2. Structures de données séquentielles	93
0. Types de base	93
1. Listes linéaire	93
<i>Implémentation par tableaux de longueur fixe (p. 93). Implémentation par tableaux dynamiques (p. 94). Implémentation par listes simplement chaînées (p. 94). Variantes des listes chaînées (p. 97).</i>	
2. Piles	97
<i>Implémentations par tableaux (p. 98). Implémentation par listes simplement chaînées (p. 99).</i>	

3. Files	100
<i>Implémentations par tableaux circulaires (p. 100). (Hors-programme) Amélioration avec des tableaux dynamiques (p. 103). Implémentations par listes simplement chaînées (p. 103). Implémentation par listes doublement chaînées (p. 104). Par double pile (p. 104).</i>	
4. Variantes des files	105
3. Aperçu des structures de données S2	105

0 Tableaux dynamiques

0.0 Tableaux C vs listes OCaml

Les tableaux sont un outil très utile de la programmation impérative¹. Ils permettent de stocker beaucoup d'éléments tout en ayant un accès à chacun de ces éléments en temps constant ; mais ont un inconvénient : ils ne sont pas redimensionnables. Si une fonction utilise un tableau à 200 cases, et qu'elle découvre durant l'exécution que sur certaines entrées il en fallait en fait plus, on est bloqués². Similairement, si on se rend compte qu'il en fallait beaucoup moins, on a gaspillé de la mémoire (et à force de gaspiller de la mémoire, on n'en a plus...)

Exemple.

- On veut faire une fonction qui prend en entrée un entier `n` et renvoie le tableau de ses diviseurs. Combien de cases réserver ? Pas clair. Il est suffisant d'en réserver `n`, bien sûr, mais on sent bien que c'est beaucoup trop.
- On veut faire une fonction prend en entrée un entier `n` et calcule le tableau des nombres premiers inférieurs ou égaux à `n`. On a exactement le même problème qu'avec le point précédent...

Et pourtant, en OCaml, ces exemples ne posent pas de difficulté particulière avec des listes. Voici par exemple pour les diviseurs de `n` :

```

3  (** Construit la liste des diviseurs positifs et >=k de n .
4     * On suppose n >= 0 . *)
5  let rec construit_lst_diviseurs n k =
6      if k > n then
7          []
8      else if n mod k = 0 then
9          k :: (construit_lst_diviseurs n (k+1))
10     else
11         construit_lst_diviseurs n (k+1)
12
13
14  (** Renvoie la liste des diviseurs positifs de n .
15     * On suppose n >= 0 . *)
16  let lst_diviseurs n =
17      construit_lst_diviseurs n 1

```



Les fonctions `lst_diviseurs` et `lst_preiers` renvoient exactement les éléments attendus, et ne surallouent aucune case : les listes créées ont exactement la même taille.

Mais à l'inverse, l'inconvénient des listes est que l'on ne peut pas accéder à un élément par son indice... et quand bien même on recoderait une fonction qui calcule le *i*ème élément d'une liste, elle serait en temps $\Theta(i)$ et non en temps constant.

1. Rappel : la **programmation impérative** est la méthode de programmation consistant à écrire un programme en décrivant une succession de modifications de la mémoire. C'est ce que l'on fait en C, par exemple.

2. Ici j'utilise 200 comme exemple, et on pourrait m'objecter que `malloc` permet de créer des "tableaux" à `n` cases. C'est vrai, mais si la fonction crée un tableau à `n` cases et se rend compte qu'il en fallait n^2 sur certaines entrées spécifiques, le problème reste le même.

Les **tableaux dynamiques** sont une structure de données qui offre le meilleur des deux mondes : ils permettent un accès à chaque élément en temps constant, et sont redimensionnables ce qui permet de les agrandir quand ils ne sont plus assez grand !

Remarque. Si vous connaissez les listes Python, ce sont des tableaux dynamiques³ !

0.1 Fonctionnement

L'objectif de nos tableaux dynamiques est de :

- Pouvoir accéder aux éléments par leur indice et les modifier en temps constant.
- Pouvoir ajouter un élément *à la fin* en temps constant⁴
- (Et aussi pouvoir créer et supprimer un tableau dynamique)

L'idée est de faire un tableau volontairement trop grand et de n'en utiliser que les premières cases. Comme ça, pour ajouter un élément à la fin, il suffit d'utiliser la prochaine case libre :

FIGURE 5.1 – Ajout dans un tableau dynamique, cas facile

Définition 1 (Tableaux dynamiques, structure de base).

Un tableau dynamique est composé de 3 données :

- le tableau en lui-même. Dans ce cours, on le nommera `arr` .
- son nombre réel de cases, c'est à dire le nombre de cases du tableau qui existent dans la mémoire. Dans ce cours, on le nommera `len_max` .
- son nombre de cases utilisées. Dans ce cours, on le nommera `len` .

Les `len` éléments stockés dans le tableau dynamique sont stockés, dans l'ordre, dans les `len` premières cases de `arr` .

Remarque. En C, `arr` ne sera pas un tableau mais une zone mémoire créée avec `malloc`. Dans cette partie, j'utilise le terme « tableau » pour simplifier. De plus, mis à part la création/destruction qui est différente, un tableau et une zone allouée s'utilisent de la même façon...

Cette façon de faire répond à presque toutes nos contraintes : on peut accéder à un élément par son indice en temps constant, modifier l'élément qui se trouve à un indice donné en temps constant ; on peut aussi ajouter un élément en temps constant lorsque `len < len_max` . Reste à définir comment ajouter un élément lorsqu'il n'y a plus de cases vides.

3. En un peu plus compliqué. Les listes Python sont une petite pépite de bonnes idées mises bout à bout pour que ce soit très efficace.

4. Ou au début, c'est symétrique.

0.1.0 Ajout dans un tableau dynamique

Une première idée serait de créer un nouveau tableau ayant une case de plus, de recopier l'ancien dedans, et d'ajouter l'élément à la fin. Hélas, cela n'est pas très efficace en temps :

FIGURE 5.2 – Ne faites pas l'AJOUT comme ça !!

Avec cette méthode, à partir du moment où le tableau est rempli, chaque ajout se fait en temps $O(\text{len_max})$. Ce n'est clairement pas efficace. On pourrait essayer de l'améliorer en créant un nouveau tableau non de longueur $\text{len_max}+1$ mais de longueur $\text{len_max}+10$, cependant cela ferait qu'1 opération sur 10 serait en temps linéaire (les 9 autres en temps constant), et donc chaque paquet de 10 opérations seraient en temps $O(\text{len_max} + 9.O(1) = O(\text{len_max}))$. Bref, si l'on prévoit de faire plusieurs ajouts, cela ne fonctionne pas très bien. Idem pour d'autres valeurs fixées de 10 : 1000, 10000, etc ont tous le même problème⁵.

La solution est de ne pas agrandir en rajoutant un nombre fixe de cases, mais un nombre variable : en fait, on va simplement *doubler* la longueur maximale !

FIGURE 5.3 – AJOUT successifs, avec doublements du tableau

5. Des petit-es malin-es objecteront que avec 10^9 on est à peu près bons, car il est rare que l'on fasse 10^9 ajouts. C'est vrai, mais : 1) cela peut arriver, et 2) vous voulez vraiment que chaque tableau dynamique demande au moins 1Go de mémoire ?

La fonction AJOUT qui ajoute un élément à un tableau dynamique est donc la suivante :

Fonction Ajout

Entrées :

- un tableau dynamique composé de `arr`, `len` et `len_max`. On les considère passés par référence.
- `x` un élément à ajouter à la fin du tableau

Sorties : rien, mais a comme effet secondaire de modifier le tableau pour y ajouter `x`

```

1 si len < len_max alors
2   arr[len] ← x
3   len ← len + 1
4 sinon
5   new_arr ← tableau de longueur 2*len_max
6   // Recopier arr dans new_arr
7   pour chaque i ∈ [0; len[ faire
8     new_arr[i] ← arr[i]
9   // Remplacer arr par new_arr
10  arr ← new_arr
11  len_max ← 2*len_max
12  // Ajouter x
13  arr[len] ← x
14  len ← len + 1

```

Remarque. Notez que les lignes 2-3 et 10-11 sont les mêmes : on peut réécrire ce code sous la forme « si `arr` est entièrement rempli, le remplacer par `new_arr` ». Ensuite, dans tous les cas, ajouter `x` ».

0.1.1 Création, suppression

Pour supprimer un tableau dynamique, il suffit de supprimer `arr`, `len` et `len_max`.

Pour créer un tableau, il suffit de créer ces trois données, avec quand même un point précis : même si l'on crée un tableau dynamique initialement vide, il faut tout de même initialiser `len_max` à au moins 1 (et donc créer `arr` comme ayant au moins 1 case). En effet, si on prend 0, comme AJOUT double la longueur maximale... deux fois zéro ça fait zéro.

0.2 Analyse de complexité

Analyser la complexité de cette version de AJOUT n'est pas aisé : la plupart des appels à la fonction coutent $O(1)$ (quand on effectue uniquement les lignes 2-3), mais certains $O(\text{len})$ (les lignes 5-11). On va en fait analyser le coût d'une *succession* d'appels à AJOUT. Faire cette analyse est l'objet de la **complexité amortie** : cf cours sur la complexité, section complexité amortie.

On y prouve que :

Proposition 2 (Complexité amortie de AJOUT).

Une succession de n ajouts coûte dans le pire des cas, au total, $O(n)$. Autrement dit, le coût amorti de AJOUT est $O(1)$.

(Précisons que cette complexité amortie est bien celle d'un pire des cas d'une succession quelconque de AJOUTE.)

Démonstration. Cf cours complexité amortie (3 preuves différentes y sont données). □

0.3 Complément mi hors-programme : RETRAIT

Notre opération AJOUT est l'équivalent du `append` de Python. Si vous connaissez Python, vous savez qu'une autre opération est souvent utilisée : `pop`, qui supprime et renvoie le dernier élément. Dans ce

cours, on la nommera RETRAIT.

Il est en soi très facile de l'implémenter : enlever l'élément d'indice $\text{len}-1$ (et faire une erreur s'il n'y a aucun élément). Mais on voudrait faire mieux : pouvoir de temps en temps réduire `len_max`, afin de récupérer la mémoire qui ne sert plus.

La première idée, qui ne marche pas, est de faire le symétrique de AJOUT : lorsque la longueur est inférieure à la moitié de la longueur maximale, créer un tableau deux fois plus petit, etc etc. En soi, l'idée est très bonne : mais elle s'agence mal avec AJOUT. En effet, juste après un AJOUT qui « double » le tableau, on obtient un tableau presque à moitié vide. Ainsi, faire un RETRAIT juste après cet AJOUT mène à une contraction du tableau, refaire un AJOUT ensuite une extension, etc : on a une suite d'opération au sein de la quelle AJOUT et RETRAIT ne seraient *pas* en temps constant amorti. Bref, ça va pas.

FIGURE 5.4 – Pourquoi il ne faut pas contracter le tableau à la moitié

Il faut en fait « éloigner » le pire cas de RETRAIT (la contraction, qui crée un tableau plus petit) du pire cas de AJOUT. Pour ce faire, une solution simple est de **diviser par 2 la longueur maximale du tableau lorsqu'il est aux 3/4 vide**, et non lorsqu'il est à moitié vide :

FIGURE 5.5 – Fonctionnement de la suppression

Fonction Retrait**Entrées :**

- un tableau dynamique composé de `arr`, `len` et `len_max`. On les considère passés par référence.

Sorties :

- si `len = 0`, indique une erreur.
- sinon, le dernier élément de `arr` (c'est à dire le plus à droite). A également pour effet secondaire modifier le tableau pour y enlever cet élément.

```

1 si len = 0 alors renvoyer une erreur
   // Retirer le dernier élément x
2 sortie ← arr[len]
3 len ← len-1

   // Si besoin, contracter arr
4 si len ≤ len_max/4 et len_max ≥ 4 alors
5   new_arr ← tableau de longueur len_max/2
6   pour chaque i ∈ [0; len] faire
7     new_arr[i] ← arr[i]
8   arr ← new_arr
9   len_max ← len_max/2

   // Ne pas oublier de renvoyer la sortie
10 renvoyer sortie
```

Proposition 3 (Complexité amortie de RETRAIT).

En codant RETRAIT et AJOUT comme présentés ici, on obtient pour ces deux opérations une complexité amortie $O(1)$.
(Précisons que cette complexité amortie est bien celle d'un pire des cas d'une succession quelconque de RETRAIT et AJOUT.)

Démonstration. Cf TD. □

1 Interfaces et types abstraits

1.0 Aspects pratiques

Remarque. Cette sous-partie du cours contient de nombreuses approximations. Ces concepts seront définis de manière plus précise et plus approfondie en cours de génie logiciel (en école).

1.0.0 Librairies

Lorsque l'on programme, il est usuel de découper son code en plusieurs fichiers : l'idée est que chaque fichier implémente « un paquet » cohérent de fonctions.

Définition 4 (Librairie et code client).

Un fichier constitué de plusieurs fonctions ayant pour but d'être utilisées dans d'autres fichiers est appelé une **librairie**.

Un code qui utilise une librairie est appelé un **code client**.

On appelle un tel paquet une **librairie**.

Exemple.

- `stdio` est une librairie C. Son nom signifie **standard in-out** : c'est la librairie standard (c'est à dire la librairie « officielle ») qui contient les fonctions de lecture (par exemple `scanf`) et d'écriture/affichage (par exemple `printf`) (`out`).
- `stdlib` (librairie standard généraliste qui contient notamment `malloc` et `free`), `stdbool` (librairie standard qui contient le type `bool`), `stdint` (librairie standard qui contient des types d'entiers sur 8/16/32/48/64 bits) ou encore `assert` (contient la fonction `assert`) sont des librairies C.

Remarque.

- Le terme « librairie » est un anglicisme qui s'est imposé. Vous croiserez peut-être à la place la traduction **bibliothèque logicielle**.
- En OCaml, on parle plutôt de **module**. En réalité, un module est un légèrement différent d'une librairie (c'en est une version évoluée), mais je ne ferai pas la différence à notre niveau.

Exemple.

- `List` est le module des listes en OCaml.
- `Printf` est le module qui contient la fonction `printf` (et ses parentes) en OCaml.
- (et nous en verrons d'autres cette année et l'an prochain)

Pourquoi programmer en plusieurs fichiers ?

- Ne pas tout recoder à chaque fois : une fois que `stdlib` est codée, pas besoin de la recoder ! Il suffit de l'inclure pour pouvoir s'en servir. Et ça tombe bien, recoder `malloc` n'est vraiment pas un exercice facile...
- Accélérer la compilation : notre code n'appelle que les librairies dont il a besoin, et rien de plus. Ainsi, il y a moins de choses à compiler, et on compile donc plus vite.
- Garder chacun des fichiers relativement courts : il est plus simple de trouver un bogue si on sait qu'il est dans tel fichier de 2000 lignes plutôt que dans tel fichier de 50000 lignes. Idem si l'on veut aller relire ou modifier une fonction particulière.
- Segmenter et organiser le travail : on programme une chose à la fois.
On peut également donner chacun des fichiers à une équipe de programmeurs/programmeuses différentes : chaque équipe se spécialise alors dans son fichier, avance efficacement dessus (pendant que les autres avancent efficacement sur le leur), et le tout est terminé plus rapidement.

En résumé, c'est très confortable !

1.0.1 Interface

Dans votre ordinateur, vous *n'avez pas* le fichier `stdlib.c`, c'est à dire le code source de `stdlib`. Vous avez à la place uniquement une version compilée de lui-ci nommée `stdlib.o`, ainsi que son interface `stdlib.h` (sounds familiar ?).

Définition 5 (Interface).


L'**interface** (en anglais : **A**pplication **P**rogramming **I**nterface, ou **API**) d'une librairie est un fichier qui décrit et abstrait son contenu. On y trouve les signatures des variables/fonctions/types déclarés la librairie et la spécification de ceux-ci ; ainsi aussi la liste des dépendances de la librairie (c'est à dire la liste des autres librairies qu'elle utilise).

Définition 6 (Organisation des fichiers d'une librairie).

En C, les interfaces sont appelées des *headers*. Si le fichier source s'appelle `truc.c`, l'interface s'appelle `truc.h` et la version compilée du code source `truc.o`.

En OCaml, `truc.ml` a pour interface `truc.mli` et pour version compilée `truc.o`.

Exemple. Voici un extrait de `dynArray.h`, une interface pour une librairie de tableaux dynamiques que nous coderons en TP :

 **dynArray.h**

```

6  #include <stdlib.h>
7  #include <stdbool.h>
8  #include <stdio.h>
9
10
11 /* Structure d'un tableau dynamique */
12 struct dynArray_s {
13     int* arr;           // pointeur vers la zone contenant les cases
14     unsigned len;       // nombre de cases utilisées de arr
15     unsigned len_max;   // nombre de cases de arr
16 };
17 typedef struct dynArray_s dynArray;
18
19
20 /* Constructeurs */
21
22 /** Crée un tableau dynamique de longueur n, dont les cases sont initialisées
23     ↪ à x */
24 dynArray dyn_create(unsigned len, int x);
25
26 /** Libère le contenu de d */
27 void dyn_free(dynArray* d);
28
29 /** Crée un tableau dynamique qui contient les len valeurs pointées par ptr
30     ↪ */
31 dynArray dynarray_of_array(unsigned len, int const* ptr);

```

Dans cet exemple, on peut voir que la librairie `dynArray` :

- utilise les librairies `stdlib`, `stdbool` et `stdio`.
- définit un type nommé `struct dynArray_s` (aussi nommé simplement `dynArray`).
- et définit les fonctions `dyn_create`, `dyn_free` et `dynarray_of_array` dont les prototypes sont indiqués. Il y a également une documentation de ces fonctions.

Pourquoi faire une interface ?

- Lorsque l'on veut utiliser une librairie, savoir *comment* la librairie fonctionne ne nous intéresse guère. Ce que l'on veut savoir, c'est quelles sont les fonctions que l'on peut appeler et ce qu'elles font : exactement ce que l'interface nous dit.

Par exemple, vous n'êtes jamais allés voir le code source de `printf` ... et heureusement.⁶

- Une interface permet de cacher certaines fonctions, en les omettant de l'interface. C'est très utile quand le code source utilise des fonctions « auxiliaires ».

Par exemple, le code source ma fonction `dyn_create` utilise une fonction `max`. Cependant, coder `max` n'est pas le but de cette librairie, et je ne veux pas alourdir l'interface en y mettant cette fonction : je ne la mets donc pas.

C'est d'autant plus utile lorsque la fonction auxiliaire est un peu incompréhensible, car elle réalise une tâche extrêmement spécifique que l'on ne comprend qu'avec le code source du tout sous les yeux.

6. Si vous êtes curieux-se, `printf` appelle `vprintf` qui appelle `vfprintf` qui fait tout le travail. Le code source de celle-ci est disponible ici : <https://github.com/lattera/glibc/blob/master/stdio-common/vfprintf.c> (la fonction commence à la ligne 1236 et fait environ 400 lignes, globalement très hors-programme).

- Une interface décrit uniquement la signature des fonctions : en particulier, on peut changer complètement la façon précise de les implémenter sans que cela n'impacte les codes clients. En d'autres termes, on peut faire une mise à jour de la librairie sans qu'il n'y ait à adapter les codes clients : c'est plutôt très bien.

Remarque. Certaines interfaces indiquent la complexité des fonctions, d'autres noms : c'est un débat. Certain-es informaticien-n-es demandent à ce que les complexité apparaissent car c'est important pour évaluer la performance du code client ; tandis d'autres demandent à ne pas les indiquer car les mettre revient généralement à forcer une implémentation spécifique et donc à s'interdire de changer le fonctionnement « sous le capot » plus tard.⁷

1.0.2 Étapes de la compilation

Cette sous-sous-partie est volontairement très vague/flou. Vous en comprendre plus les termes avec de l'expérience.

Définition 7 (Étapes de compilations).

Un compilateur effectue les étapes suivantes quand il compile :

- Pré-compilation : une première passe qui a pour but de préparer les fichiers. C'est notamment dans cette phase qu'en C les `sizeof(type)` sont remplacés par leur valeur, et (hors-programme) que les `#define` sont résolus.
- Analyse lexicale, syntaxique et sémantique du code : il s'agit de « lire » le code et de comprendre son « organisation interne ». Cf cours de MPI pour l'analyse lexicale et syntaxique (l'analyse sémantique est hors-programme).
- Génération de code intermédiaire puis de code objet : chacun des fichiers du code est, individuellement, transformé en une version compilée de ses fonctions. Cette version compilée est le fichier `.o`. Cependant, les appels de fonctions ne sont pas encore fonctionnels dans ces `.o` : lorsqu'une fonction A appelle une fonction B, l'appel ne fonctionne pas encore : il y a à la place quelque chose comme « TODO : ici insérer un appel à B ».
- Édition de liens : les appels de fonctions sont rendus fonctionnels, et le fichier est rendu exécutable.

C'est notamment l'étape où le compilateur doit « relier » différents codes sources entre eux pour la première fois.

FIGURE 5.6 – Résumé du rôle des différents fichiers

7. À titre personnel, je suis plutôt de celles et ceux qui veulent voir les complexités ; quitte à ce que ce soit avec la mention « cette complexité pourra changer à l'avenir ». Mon avis personnel n'a cependant que très peu de valeur.

Remarque. (Vous n'avez pas à comprendre le détail de chaque étape pour comprendre ce qui suit :)

- les librairies sont généralement stockées dans votre ordinateur avec uniquement le fichier objet `.o` et l'interface `.h` (ou `.mli`). Ainsi, quand on veut utiliser une librairie, le compilateur n'a pas besoin de refaire toutes les premières étapes sur la librairie ! Il n'aura à faire que l'édition de liens, et la compilation est donc grandement accélérée.
- (Hors-programme) Vous avez peut-être déjà entendu parler des fichiers *shared object* `.so` sous Linux, ou des `.dll` sous Windows. Ce sont une sorte particulière de fichiers objets, c'est à dire une version compilée d'une librairie. C'est en réalité sous cette forme que sont distribuées les librairies, plutôt que comme des simples `.o`.

1.1 Aspects théoriques

Définition 8 (Structure de données).

Une **structure de données** est une façon d'organiser des données et de les manipuler efficacement.

Exemple. En décrivant les tableaux dynamiques en section 1, nous avons décrit une structure de donnée.

Définition 9 (Type abstrait, signature).

Un **type abstrait**, aussi appelé **structure de donnée abstraite**, est une considération abstraite d'une structure de donnée. Elle est décrite par sa **signature**, c'est à dire par la donnée de :

- l'identifiant (le nom) du type abstrait.
- les autres types abstraits dont il dépend.
- les identifiants de constantes du type.
- la signature des opérations que l'on peut effectuer sur le type.

Exemple. Voici deux signatures :

(a) Interface du type abstrait Bool

(b) Interface du type abstrait Tableau Dynamique

FIGURE 5.7 – Deux exemples d'interface

Définition 10 (Signature vs documentation (syntaxe vs sémantique)).

La signature explique quelles sont les « choses » que l'on peut écrire à l'aide du type (quelles fonctions on peut appliquer et dans quels cas). Mais elle n'explique pas ce que *font* ces « choses » ! En termes scientifiques, on dit qu'une signature décrit uniquement la **syntaxe**, c'est à dire les règles d'écritures.

Pour savoir ce que font ces « choses », il faut donner de la **sémantique**, c'est à dire **documenter** les constantes et les opérations du type.

Exemple. Voici des exemples de documentation pour les 3 opérations des booléens :

- **non** : renvoie la négation d'un booléen, c'est à dire que $\text{Non}(\text{vrai}) = \text{faux}$ et réciproquement.
- **ou** : renvoie vrai si et seulement si au moins l'un de ses deux arguments vaut vrai .
- **et** : renvoie vrai si et seulement si ses deux arguments valent vrai .

Remarque.

- La documentation peut aussi prendre une forme très mathématique. C'est notamment utile lorsque l'on veut automatiser les preuves de programme. Voici par exemple un axiome vérifié par Longueur :
 $\forall n : \text{Entier}, \forall x : \text{Élément}, \text{longueur}(\text{creer}(n, x)) = n$.
 Dans la plupart des cas, sauf ambiguïté, on préfère donner une documentation en français plutôt qu'en mathématique pour des raisons évidentes de lisibilité.
- D'un point de vue pratique, une erreur de *syntaxe* est quelque chose qui est détecté à la compilation (par exemple une erreur de type), tandis qu'une erreur de *sémantique* crée un bogue dont on ne se rend compte qu'en testant le code.
- En langage naturel (en français si vous êtes francophone), une erreur de syntaxe correspond à une erreur d'orthographe/grammaire, tandis qu'une erreur de sémantique correspond à une phrase qui ne veut rien dire.

Définition 11 (Constructeur, accesseur, transformateur).

n classe les différentes opérations d'un type abstrait en :

- **constructeur** : ce sont les opérations qui créent ou détruisent une de données.
- **accesseur** : ce sont les opérations qui renvoient une information sur la structure de données (sans la modifier).
- **transformateur** : ce sont les opérations qui modifient une structure de donnée existante.

Exemple. Dans le type abstrait `Tableau dynamique`, `Créer` est un constructeur, `Longueur` un accesseur et `Ajout` un transformateur.

Remarque. Les signatures des transformateurs peuvent être écrites de deux façons : soit d'une façon impérative (ils ne renvoient pas de structure de donnée, mais modifient la structure de donnée passée en argument), soit de façon fonctionnelle (ils ne modifient pas leur argument mais renvoient le nouvel état de la structure de données).

Exemple.

- « `Ajout : Tableau Dynamique × Élément → Rien` » est une signature impérative (et la documentation doit préciser que `Ajout` modifie celui passé en argument).
- « `Ajout : Tableau Dynamique × Élément → Tableau Dynamique` » est une signature fonctionnelle (et la documentation doit préciser l'absence d'effets secondaires).

2 Structures de données séquentielles

Une structure de donnée est dite **séquentielle** lorsqu'elle range les éléments les uns après les autres, dans un certain ordre.

Exemple. Les tableaux C ou les listes OCaml sont des structures de données séquentielles.

L'objectif de cette section est de présenter différents types abstraits de structures de données séquentielles, et une ou plusieurs implémentations.

2.0 Types de base

On a déjà vu comment sont implémentés les entiers, flottants, booléens : dans ce cours, je les considérerai comme des types de base.

On a également déjà vu certains types construits à l'aide d'autres types : par exemple les pointeurs (on utilise des pointeurs vers un élément d'un autre type) ou les tableaux (des tableaux d'un autre type).

Remarque. Notez qu'il existe en fait de nombreux types de tableaux : les tableaux d'entiers, les tableaux de booléens, les tableaux de etc... On dit que le type tableau est **paramétré** par le type de son contenu.

2.1 Listes linéaire

Avant-propos : le terme « liste » est victime d'une forte polyésmie⁸. Il veut dire des choses différentes ici et là. Dans ce cours, par « liste » je désigne spécifiquement une liste linéaire :

Définition 12 (Liste linéaire).

Une **liste linéaire** est une suite finie, éventuellement vide, d'éléments repérés selon leur rang :

$\ell = [\ell_0; \dots; \ell_{n-1}]$.

Si E est l'ensemble des éléments, l'ensemble \mathbb{L} est défini récursivement par :

$$\mathbb{L} = \emptyset + E \times \mathbb{L}$$

Définition 13 (Signature de Liste linéaire).

La signature (fonctionnelle) du type abstrait Liste Linéaire est :

- Type : Liste Linéaire (abrégié List ci-dessous)
- Utilis : Entier, Élément
- Constantes : `[]` est une liste linéaire
- Opérations :
 - `longueur` : `List` → Entier
 - `acces_ieme` : `List` × Entier → Élément
 - `supprime_ieme` : `List` × Entier → List
 - `insere_ieme` : `List` × Entier × Élément → List

Les trois dernières fonctions, respectivement : renvoient l'élément d'indice pris en argument, insère un élément de sorte à ce qu'il soit à l'indice pris en argument, ou supprime l'élément d'indice pris en argument.

2.1.0 Implémentation par tableaux de longueur fixe

Pour cette implémentation, on fixe un majorant `len_max` de la longueur de toute liste linéaire.

8. Notamment à cause de Python, qui utilise le terme pour désigner ses tableaux dynamiques... et encore, le terme était déjà polyésmique avant Python.

On représente alors une liste linéaire par un tableau à `len_max` cases, dont on n'utilise que les premières pour stocker les éléments les uns après les autres. Il faut également stocker `len` la longueur réelle de la liste. Autrement dit, on fait comme des tableaux dynamiques, mais sans redimensionner :

FIGURE 5.8 – Liste linéaire dans un tableau de longueur fixe

Proposition 14 (Complexités des listes linéaires par tableaux).

ne telle implémentation permet d'obtenir les complexités suivantes :

- `[]` : se calcule en $O(1)$ s'il ne faut pas initialiser les cases du tableau, et en $O(\text{len_max})$ sinon.
- `longueur` : $O(1)$
- `acces_ieme` : $O(1)$
- `insere_ieme` : $O(\text{len} - i)$ avec i l'indice auquel on insère.
- `supprime_ieme` : $O(\text{len} - i)$ avec i l'indice auquel on supprime.

Démonstration. Les seules complexités non triviales sont les deux dernières. Je présente ici uniquement la première, l'autre étant similaire.

Pour insérer en position i , on « décale » d'une case tous les éléments d'indice $> i$: il y a $\text{len} - i$ éléments à décaler, chaque décalage se fait en temps constant (1 écriture dans une case du tableau + 1 calcul d'indice), donc tout ce décalage est en $O(\text{len} - i)$. Il ne reste qu'à écrire l'élément dans la case voulue, ce qui se fait en temps constant. \square

2.1.1 Implémentation par tableaux dynamiques

En utilisant les idées des tableaux dynamiques, on peut utiliser des tableaux "redimensionnables" pour améliorer l'implémentation précédente. Cela permet de manipuler des listes linéaires dont la valeur finit par dépasser le `len_max` initial.

Comme on l'a vu dans le chapitre sur les tableaux dynamique, en amorti les complexités restent les mêmes.

2.1.2 Implémentation par listes simplement chaînées

Définition 15 (Liste simplement chaînée).

Une **liste simplement chaînée** est une structure de données composée de **cellules** (aussi appelées **maillons**, ou parfois **noeuds**) reliées entre elles. Chaque cellule stocke deux informations :

- L'élément de la liste stockée dans cette cellule.
- Un pointeur vers la cellule de l'élément suivant.

FIGURE 5.9 – Organisation d'une liste simplement chaînée

Exemple.

FIGURE 5.10 – Une liste chaînée particulière

Remarque.

- Sur mes schémas, je ne représente pas les cellules comme étant « proprement les unes après les autres ». C'est pour vous rappeler qu'en pratique, les cellules sont allouées sur le tas mémoire, et que l'on a aucune garantie de position dans le tas mémoire.
- On sait que l'on a atteint la fin d'une liste lorsque le pointeur vers la prochaine cellule a une valeur particulière. En C, c'est généralement NULL (le pointeur qui ne pointe sur rien).
- On type souvent une liste simplement chaînée comme étant un pointeur vers une cellule. Ainsi, pour donner une liste simplement chaînée, il suffit de donner un pointeur vers la cellule de tête. Pour donner la liste vide, on donne le pointeur NULL (ou autre valeur particulière fixée).
- C'est ainsi que sont implémentées les listes en OCaml !! Cela correspond d'ailleurs à ce que l'on a vu en TP, où l'on a dit qu'une liste correspond à l'un de ces deux motifs :

```
1 | []      (* liste vide *)
2 | t :: q  (* valeur de tête suivi d'un accès à la queue *)
```



En particulier, en OCaml, passer une `'a list` en argument revient exactement à passer un pointeur, c'est à dire que cela se fait en temps constant.

Proposition 16 (Complexité des opérations sur une liste linéaire).

Cette implémentation permet d'obtenir les complexités suivantes pour une liste linéaire à len éléments :

- `[]` : $O(1)$
- `longueur` : $O(\text{len})$
- `acces_ieme` : $O(i)$
- `insere_ieme` : $O(i)$ avec i l'indice auquel on insère.
- `supprime_ieme` : $O(i)$ avec i l'indice auquel on supprime.

Démonstration. Respectivement :

- il suffit de renvoyer le pointeur NULL
- il faut parcourir toutes les cellules, en mémorisant le nombre de cellules. Chaque cellule se traite en temps constant (incrémenter le nombre de cellule vues puis aller à la suivante).
- idem, mais on ne les parcourt que jusqu'à atteindre la cellule d'indice i (où l'on renvoie alors la valeur stockée, en temps constant).
- Similaire au précédent, à ceci près que :
 - 0) On accède à la cellule d'indice i .
 - 1) On crée une nouvelle cellule, que l'on fait pointer vers la queue de la cellule d'indice i .
 - 2) on modifie le chainage de la cellule d'avant. Si l'on ne veut pas modifier la liste prise en argument, on doit recréer tout le début de la liste.

FIGURE 5.11 – Insertion dans une liste chaînée, sans modifier l'originale

- De même que le précédent.



Remarque. Un gros avantage des listes simplement chaînées est qu'elles réduisent la duplication en mémoire lorsque l'on « étend » de différentes façons des listes. Par exemple :

```
1 let lst = [4; 5; 6; 7; 8; 9]
2 let adora = 3 :: lst
3 let catra = 10 :: 20 :: lst
4 let glimmer = 55 :: lst
```



correspond en mémoire à :

FIGURE 5.12 – Illustration de la non-duplication des queues des listes OCaml

Cette propriété est notamment utile lorsque l'on fait un algorithme qui essaye différentes façons d'étendre une liste ; par exemple les algorithmes de retour sur trace (ou *backtracking* en anglais, que nous verrons au S2).

Remarque. (Petit complément au programme) De même, la fonction `List.append` qui concatène deux listes (c'est à dire les « colle ») est en temps et espace linéaire en la longueur de la première liste. En particulier, si on peut éviter d'y faire appel plusieurs fois, c'est (bien) mieux.

Pro-tip (au programme aussi) : `List.append l0 l1` se note aussi `l0 @ l1` .

Exemple.

```
1 let l0 = [1; 2]
2 let l1 = [2; 3; 4; 5]
3 let l2 = l0 @ l1
```



correspond en mémoire à :

FIGURE 5.13 – Illustration de la concaténation de listes OCaml

2.1.3 Variantes des listes chaînées

Voici quelques variantes qui existent (il en existe bien d'autres) :

- On peut mémoriser une « meta-donnée » en plus de la liste linéaire, typiquement un pointeur vers le début et la fin :

- La liste peut-être doublement chaînée, c'est à dire que chaque cellule retient la cellule d'avant et celle d'après.
- Le chaînage peut-être circulaire, c'est à dire que la dernière cellule pointe sur la première.
- Et bien d'autres que vous croiserez peut-être.⁹

2.2 Piles

Une pile est un cas particulier de liste linéaire. On abandonne l'idée d'insérer ou de supprimer n'importe où, et même d'accéder n'importe où. En échange, on demande une *garantie* sur l'ordre des éléments :

Définition 17 (Pile).

Une **pile** (anglais : **stack**) est une liste linéaire qui garantit que le dernier élément inséré sera le premier élément supprimé. En particulier, on ne peut pas insérer ou supprimer n'importe où. On dit qu'une pile vérifie le principe **LIFO** : Last In First Out (dernier entré, premier sorti).

9. Techniquement, on peut voir les arbres (S2) comme le cas où il peut y avoir plusieurs cellules suivantes... je préfère voir les listes chaînées comme un cas particulier des arbres.

Exemple. Une pile de linge à trier, ou une pile de copies à corriger, ou n'importe quel autre empilement de la vie réelle.

Exemple.

FIGURE 5.14 – Empilages et dépilages successifs sur une pile

Remarque. La pile mémoire est une pile.

Définition 18 (Signature des piles).

Voici la signature (impérative) des piles, un peu documentée :

- Type : Pile
- Utilise : Élément, Booléen
- Constantes : `pile_vide`
- Opérations :

<ul style="list-style-type: none"> – <code>empile</code> : Pile \times Élément \rightarrow rien – <code>depile</code> : Pile \rightarrow Élément – <code>sommet</code> : Pile \rightarrow Élément – <code>est_vide</code> : Pile \rightarrow Bool 	<p><i>Cette fonction ajoute un élément sur la pile (et la modifie donc).</i></p> <p><i>Cette fonction retire le dernier élément de la Pile (et la modifie donc) et le renvoie.</i></p> <p><i>Renvoie l'élément au « sommet » de la Pile, c'est à dire le prochain élément à sortir de la pile.</i></p>
--	--

Remarque.

- On aurait aussi pu donner une signature fonctionnelle.
- Cette propriété **LIFO** rend les piles indispensables dans la résolution de nombreux problèmes (à noter que la récursion peut fréquemment remplacer une pile).

2.2.0 Implémentations par tableaux

L'idée est de faire un tableau dynamique, où l'on empile et depile à la fin : `empiler` correspond alors à `AJOUT`, et `depiler` à `RETRAIT`.

FIGURE 5.15 – Succession d'opération sur des piles implémentées par tableaux dynamiques

Proposition 19 (Complexité des piles par tableaux dynamiques).

Cette implémentation permet d'obtenir les complexités suivantes pour une pile éléments :

- `pile_vide` : $O(1)$
- `empile` : $O(1)$ amorti
- `depile` : $O(1)$ amorti
- `sommet` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Cf la section sur les tableaux dynamiques. Pour `sommet`, il suffit de renvoyer l'élément d'indice `len-1`, et pour `est_vide` il suffit de tester si `len = 0`. □

2.2.1 Implémentation par listes simplement chaînées

L'idée est de faire une liste simplement chaînée, où l'on empile et dépile en tête : `empiler` correspond alors à une insertion en indice 0, et `depiler` à une suppression en indice 0 (plus le fait de renvoyer l'élément).

FIGURE 5.16 – Succession d'opération sur des piles implémentées par listes simplement chaînées

Remarque. C'est cette implémentation qui est utilisée par le module `Stack` de OCaml, où le type `'a stack` est défini comme étant une `'a list ref` (c'est à dire un pointeur vers une `'a list`).

Proposition 20 (Complexité des piles par listes simplement chaînées).

Cette implémentation permet d'obtenir les complexités suivantes pour une pile :

- `pile_vide` : $O(1)$
- `empile` : $O(1)$
- `depile` : $O(1)$
- `sommet` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Cf la sous-sous-section sur les listes simplement chaînées. Pour `empiler`, `depiler` et `sommet`, ces opérations travaillent sur la cellule de tête et donc se font bien en temps $O(1)$. □

2.3 Files

Comme les piles, les Files sont des listes linéaires particulières où l'on restreint insère, supprime et accés pour obtenir une garantie d'ordre :

Définition 21 (File).

Une **file** (anglais : **queue**) est une liste linéaire qui garantie que le premier élément inséré sera le premier élément supprimé. En particulier, on ne peut pas insérer ou supprimer n'importe où. On dit qu'une file vérifie le principe **FIFO** : **F**irst **I**n **F**irst **O**ut (premier entré, premier sorti).

Exemple. Une file d'attente dans une boulangerie^{10 11}.

Exemple.

FIGURE 5.17 – Enfilages et défilages successifs dans une file

Définition 22 (Signature des files).

Voici la signature (impérative) des files, un peu documentée :

- Type : File
- Utilise : Élément, Booléen
- Constantes : file_vide
- Opérations :

<ul style="list-style-type: none"> – enfile : File × Élément → rien – defile : File → Élément – prochain : File → Élément – est_vide : File → Bool 	<p><i>Cette fonction fait entrer un élément dans la file (et la modifie donc).</i></p> <p><i>Cette fonction fait sortir le premier élément de la File (et la modifie donc) et le renvoie.</i></p> <p><i>Renvoie le prochain élément à sortir de la file.</i></p>
--	--

Remarque.

- Là encore, on aurait aussi pu donner une signature fonctionnelle.
- Cette propriété **FIFO** rend les piles indispensables dans la résolution de nombreux problèmes.

2.3.0 Implémentations par tableaux circulaires

On borne le nombre d'éléments de la file par une certaine constante len_max.

Pour utiliser des tableaux, la première idée est de faire presque comme avec des piles par tableaux dynamiques : pour enfiler on ajoute à droite, et pour défiler on retire à gauche. Cette méthode fonctionne, mais atteint vite un problème :

¹⁰. Boulangerie de Lamport =D

¹¹. Vous comprendrez la blague de la note de bas-de-page précédente en MPI. Je vous assure qu'elle est drôle.

FIGURE 5.18 – Problème de l'implémentation naïve des files dans un tableau

Pour pouvoir utiliser l'espace disponible, l'astuce est de rendre le tableau **circulaire** :

Définition 23 (Implémentation des files par tableaux circulaires).

Pour représenter une file dans un tableau circulaire à len_max cases, on mémorise :

- le tableau `arr` et sa longueur `len_max`
- `entree`, un indice du tableau qui stocke l'entrée de la file (c'est là qu'auront lieu les enfilages).
- `sortie`, un indice du tableau qui stocke la sortie de la file (c'est là qu'auront lieu les défilages).

Les éléments de la file sont alors stockés à partir de l'indice `debut` et jusqu'à l'indice `fin`, en prenant en compte la circularité (çad en calculant le prochain indice modulo len_max) :

FIGURE 5.19 – Organisation d'un tel tableau circulaire

FIGURE 5.20 – Un autre exemple où l'on voit l'aspect circulaire

Remarque. Les indices `entrée` et `sortie` sont parfois des indices de la file *inclus*, c'est à dire qu'on y trouve bien des éléments de la file, et parfois *exclus* (c'est à dire que le premier/dernier élément se trouve juste avant/après). Il faut bien lire la description de l'énoncé pour savoir !

Exemple.

FIGURE 5.21 – Succession d'enfilage et de défilage dans une file par tableaux circulaires

Remarque. Distinguer la file vide d'autres files n'est pas évident, et la façon de le faire dépend de l'implémentation. Il faut bien lire la description de l'implémentation pour savoir !

(a) File vide ou à 1 élément ?

(b) File vide ou pleine ?

FIGURE 5.22 – File vide ou non ?

Proposition 24 (Complexité des files par tableaux circulaires).

Cette implémentation permet d'obtenir les complexités suivantes pour une file :

- `file_vide` : $O(1)$ ou $O(\text{len_max})$ s'il faut initialiser le contenu des cases
- `enfile` : $O(1)$
- `defile` : $O(1)$
- `prochain` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Respectivement :

- Il suffit de créer le tableau.
- Il suffit d'écrire dans la bonne case, qui est connue grâce à `entrée`, puis de mettre à jour `entree`.
- Idem mais pour la sortie.
- Il suffit de lire la bonne case grâce à l'indice `sortie`.
- La longueur d'une file soit se déduit de l'écart entre `entrée` et `sortie`, soit est stockée (en plus de stocker les deux indices) : dans tous les cas, c'est en temps constant.

□

Proposition 25 (Longueur d'une file par tableau circulaire).

Le nombre d'éléments d'une file implémentée par tableaux circulaires peut se calculer à l'aide de l'écart entre les indices.

La formule est généralement de la forme $(\text{sortie} - \text{entree}) \bmod \text{len_max}$, avec parfois un ± 1 à rajouter au tout selon l'implémentation précise utilisée.

Exemple. Dans les exemples ci-dessous, *sortie* est l'indice (inclus) du prochain élément à sortir, et *entrée* l'indice (inclus) de l'entrée de la file (c'est à dire du dernier élément inséré).

(a) Une première file à 4 éléments

(b) Une seconde file à 4 éléments (où la circularité se voit)

FIGURE 5.23 – Exemples de calculs de longueur de file dans des tableaux circulaires

2.3.1 (Hors-programme) Amélioration avec des tableaux dynamiques

Pour échapper au fait que les files dans des tableaux circulaires sont de longueur au plus len_max , on peut les implémenter dans des tableaux dynamiques. Il faut alors faire attention lors du doublement du tableau à correctement mettre à jour *debut* et *fin*, notamment lorsque $\text{fin} < \text{debut}$.

2.3.2 Implémentations par listes simplement chaînées

Une autre implémentation possible utilise des listes simplement chaînée dont on mémorise un pointeur vers le début et la fin. Ici, on utilisera une signature impérative : on modifie la liste simplement chaînée passée en argument¹²

Exemple.

FIGURE 5.24 – Enfilages et défilages successifs dans une file par listes simplement chaînée

12. Cela sera donc différent des exemples de listes OCaml précédents, donc.

Définition 26 (Implémentation des files par listes simplement chaînée).

Pour implémenter des files à l'aide de listes simplement chaînées, on utilise :

- une liste simplement chaînée. Le prochain élément à sortir est en tête, le dernier élément inséré dans la cellule de fin de liste.
- un pointeur sortie vers la tête de la liste. Il permet de défiler.
- un pointeur entree vers la dernière cellule de fin. Il permet d'enfiler.

Pour défiler, il suffit alors de lire l'élément du maillon de tête, et de modifier le pointeur sortie pour qu'il pointe sur la queue.

Pour enfiler, on crée un nouveau maillon de fin contenant l'élément à enfiler, puis on modifie le chainage de l'ancien maillon de fin (accessible via entree) pour le faire pointer sur le nouveau. On met ensuite à jour entree en conséquence.

Proposition 27 (Complexités des files par listes simplement chaînées).

ne telle implémentation permet d'obtenir les complexités suivantes :

- `file_vide` : $O(1)$
- `enfile` : $O(1)$
- `defile` : $O(1)$
- `prochain` : $O(1)$
- `est_vide` : $O(1)$

Démonstration. Toutes ces complexités devraient être évidentes à l'aide des descriptions des listes simplement chaînées et du fonctionnement de `enfile` et `defile`.

Précisons que pour `est_vide`, il suffit de tester si `sortie` pointe sur un maillon ou sur rien. \square

2.3.3 Implémentation par listes doublement chaînées

On peut faire l'implémentation précédente avec des listes doublement chaînées (et toujours un pointeur vers chacune de deux extrémités). Cela permet de parcourir la file dans les deux sens, et non simplement de la sortie vers l'entrée.

Remarque. C'est une version très connue, au point où plusieurs livres sur le sujet ne mentionnent pas la version simplement chaînée. Elle est pourtant très pratique ; parcourir la file dans le sens entrée->sortie ne sert pas si souvent que ça¹³.

2.3.4 Par double pile

On peut implémenter une file à l'aide de deux piles. L'idée est d'avoir une pile « d'entrée » où l'on empile les éléments enfilés, une pile de sortie où l'on dépile les éléments défilés, et lorsque cette dernière est vide on renverse la pile d'entrée sur la pile de sortie.

FIGURE 5.25 – File par double pile

13. Ne me faites pas dire ce que je n'ai pas dit : ça sert quand même. Mais à mon humble avis, pas assez souvent pour que cela justifie de ne pas présenter la version simplement chaînée.

Les opérations de la signature des files sont alors en temps constant (amorti). Plus de détails en TD.

Remarque. On peut aussi implémenter une pile à l'aide de deux files, mais c'est plus compliqué.

2.4 Variantes des files

Plusieurs variantes des files existent :

- Files à double entrée : on peut enfiler ou défiler sur les deux extrémités de la file. En anglais, on parle de **double ended queue**, souvent abrégé **dequeue**.
Pour les implémenter, on peut utiliser des tableaux circulaires ou des listes doublement chaînées.
- File de priorité : les éléments ne sortent pas de la file selon leur ordre d'entrée, mais selon une priorité qui leur est assignée. C'est typiquement ce que fait l'ordonnanceur de tâches de votre ordinateur (certains processus sont plus prioritaires à « faire avancer » que d'autres), un site internet (on priorise certaines requêtes sur d'autres¹⁴), ou un hôpital (on priorise les patient-es dont l'état est le plus critique). Nous verrons au S2 comment les implémenter.

3 Aperçu des structures de données S2

Au semestre 2, nous verrons d'autres de structure de données qui permettent de résoudre d'autres problèmes :

- Une nouvelle structure séquentielle, les dictionnaires.
- Les structures hiérarchiques (où les éléments ne sont pas à la suite mais au-dessus/dessous d'autres éléments) : les arbres (sous plusieurs formes), dont notamment les tas qui permettent de faire des files de priorité.
- Les structures relationnelles (où on stocke des relations entre éléments) : les graphes, qui permettent de faire à peu près l'entièreté de l'informatique^{15 16}

Nous verrons également et surtout plus de méthodes algorithmiques, c'est à dire de chapitres de « conception de solution pour résoudre un problème » (comme on a cherché des solutions pour Hanoï et autres problèmes dans le chapitre sur la récursion). Ces chapitres s'appuieront souvent sur des structures de données pour résoudre efficacement les problèmes.

14. De manière générale, l'organisation des réseaux informatiques ou téléphoniques sont basées sur des files d'attente et des files de priorité extrêmement optimisées; ainsi que sur des modélisations et études probabilistes poussées.

15. J'exagère un peu.

16. Mais pas tant que ça.

Chapitre 6

BONNES PRATIQUE DE PROGRAMMATION

Notions	Commentaires
Annotation d'un bloc d'instructions par une précondition, une postcondition, une propriété invariante.	Ces annotations se font à l'aide de commentaires.
Programmation défensive. Assertion. Sortie du programme ou exception levée en cas d'évaluation négative d'une assertion.	L'utilisation d'assertions est encouragée par exemple pour valider des entrées ou pour le contrôle de débordements. Plus généralement, les étudiants sont sensibilisés à réfléchir aux causes possibles (internes ou externes à leur programme) d'opérer sur des données invalides et à adopter un style de programmation défensif. [...]
Explicitation et justification des choix de conception ou programmation.	Les parties complexes de codes ou d'algorithmes font l'objet de commentaires qui l'éclairent en évitant la paraphrase.

Extrait de la section 1.2 du programme officiel de MP2I : « Discipline de programmation ».

Notions	Commentaires
Jeu de tests associé à un programme.	Il n'est pas attendu de connaissances sur la génération automatique de jeux de tests ; un étudiant est capable d'écrire un jeu de tests à la main, donnant à la fois des entrées et les sorties correspondantes attendues. On sensibilise, par des exemples, à la notion de partitionnement des domaines d'entrée et au test des limites.
[...] Chemins faisables. Couverture des sommets, des arcs ou des chemins (avec ou sans cycle) du graphe de flot de contrôle.	Les étudiants sont capables d'écrire un jeu de tests satisfaisant un critère de couverture des instructions (sommets) ou des branches (arcs) sur les chemins faisables.
Test exhaustif de la condition d'une boucle ou d'une conditionnelle.	Il s'agit, lorsque la condition booléenne comporte des conjonctions ou disjonctions, de ne pas se contenter de la traiter comme étant globalement vraie ou fausse mais de formuler des tests qui réalisent toutes les possibilités de la satisfaire. On se limite à des exemples simples pour lesquels les cas possibles se décèlent dès la lecture du programme.

Extrait de la section 1.3 du programme officiel de MP2I : « Validation, test ».

Notions	Commentaires
Notion de portée syntaxique [...]. [...]	On indique la répartition selon la nature des variables : globales, locales, paramètres.

Extrait de la section 5.2 du programme officiel de MP2I : « Gestion des ressources de la machine ».

SOMMAIRE

0. Écrire mieux.....	108
1. Tester mieux.....	108

0 Écrire mieux

1 Tester mieux