



IN101

Algorithmique et programmation

François Pessaux

2023-2024 v1.4.0



Table des matières

1 Introduction, rappel et motivations	7
1.1 Introduction	7
1.2 Motivations	8
1.3 Rappels d'architecture matérielle	9
1.3.1 Traitement de l'information	9
1.3.2 La mémoire	9
1.3.3 Le microprocesseur	10
1.4 Quelques rappels de C	11
1.4.1 Compilation	11
1.4.2 Rappels de syntaxe	12
1.5 Algorithme	15
2 Spécification et domaines	19
2.1 Spécification	19
2.1.1 Une spécification simple	19
2.1.2 Une spécification intermédiaire	19
2.1.3 Une spécification pas triviale	20
2.1.4 Conseils	20
2.1.5 Un premier exemple	21
2.2 Après l'écriture	23
2.3 Conclusion	24
2.4 Domaines de entrées et des sorties	25
2.4.1 Domaine mathématique	26
2.4.2 Domaine sémantique	26
2.4.3 Domaine informatique	26
2.4.4 Conclusion	26
3 Passer à l'échelle	29
3.1 Vers des programmes plus compliqués	29
3.1.1 Structurer le code	29
3.1.2 Rédiger petit-à-petit	30
3.2 Quelques défauts courants de conception	32
3.2.1 Affichage <i>versus</i> retour	32
3.2.2 Duplication, imbrication, fouillis dans le code	32
3.3 Tester son programme	33
3.3.1 Tests fonctionnels	34
3.3.2 Tests fonctionnels <i>versus</i> structurels	37
3.3.3 Tests structurels	37

4 Complexité	39
4.1 Comparer des programmes	39
4.2 Complexité	40
4.3 Premier exemple	41
4.4 Notations de complexité	42
4.4.1 Classes de complexité	42
4.5 Complexité d'algorithmes pour Fibonacci	43
4.5.1 Calcul direct	43
4.5.2 Version récursive naïve	44
4.5.3 Version itérative simple	45
4.5.4 Version itérative plus rusée	45
4.5.5 Version itérative optimale	46
4.5.6 Comparaison des temps de calcul	46
4.6 Conclusion	47
5 Différents paradigmes de programmation	49
5.1 Diviser pour régner	49
5.2 Programmation dynamique	50
5.3 Programmation gloutonne	54
5.4 Conclusion	56
6 Initiation sommaire à la preuve d'algorithmes	57
6.1 Introduction	57
6.2 Spécification	57
6.3 Rappels de logique	57
6.4 Preuve d'algorithmes récursifs	58
6.4.1 Correction partielle, correction totale	58
6.4.2 Terminaison de fonction récursive	58
6.4.3 Valeur absolue	58
6.4.4 Division euclidienne (1)	59
6.4.5 Multiplication (1)	60
6.4.6 Puissance entière (1)	61
6.4.7 Une fonction qui ne termine pas	62
6.4.8 Une fonction qui ne termine (toujours) pas	62
6.4.9 Terminaison de fonction récursive un peu plus en détail	63
6.5 Preuve d'algorithmes avec des boucles	65
6.5.1 Correction (partielle)	65
6.5.2 Terminaison de boucle	65
6.5.3 Identité	66
6.5.4 Fausse l'identité	67
6.5.5 Division euclidienne (2)	68
6.5.6 Division euclidienne (3)	69
6.5.7 Puissance entière (2)	70
6.5.8 Multiplication (2)	70
6.5.9 Minimum dans un tableau	71
6.5.10 Appartenance à un tableau	73
6.6 Conclusion	74

Préambule

Ce document est le support accompagnant le cours IN101 « Algorithmique et programmation » dispensé à l'ENSTA Paris dans le cadre de la première année du cursus ingénieur. Il s'adresse à un public d'élèves ingénieurs généralistes ayant déjà rencontré l'informatique dans leur cursus antérieur, sans pour autant avoir approfondi le domaine. Ce document se veut donc un accompagnement pour la conception méthodique d'algorithmes répondant à des spécifications en vue d'une implantation finale dans un langage de programmation. Le langage choisi est C mais pourrait être tout autre. Ce document n'est pas un cours d'algorithmique proposant l'étude de structures de données ou d'algorithmes « standards bien connus ». Ce document n'est pas non plus un livre habituel : il propose de suivre fidèlement le déroulement des séances de cours et pourra donc présenter quelques détours et retours vers des propos rapidement esquissés dans des sections précédentes.

Si vous détectez des erreurs, des typos, je vous serai très reconnaissant de m'en faire part afin de pouvoir rendre ce document de meilleure facture pour les générations suivantes. Bonne lecture...

Séance 1

Introduction, rappel et motivations

1.1 Introduction

Il est une idée très répandue qui veut que l'informatique ne soit qu'un outil. Les mathématiciens font de la science exacte, les physiciens et les chimistes des sciences expérimentales, et les informaticiens ... de la bidouille. Il est curieux de noter que les bases de l'informatique moderne remontent bien avant l'avènement du matériel ayant permis de donner vie à cette informatique. Remontons « quelques » années en arrière, à l'époque de Georges Boole (1815-1864). Entre autres travaux, il se consacre à l'étude d'une algèbre de fainéant : 2 symboles. Difficile de faire moins sans tomber dans la trivialité ! Et pourtant, ces travaux posent une des pierres angulaires de ce qui deviendra des décennies plus tard l'informatique. Mais à cette époque-là, point d'ordinateurs.

Quelques années plus tard, dans les années 30, émerge une branche de la logique mathématique : la théorie de la calculabilité. On trouve parmi les personnes s'intéressant à ce domaine de grands noms comme Hilbert, Turing, Church, Kleene, Gödel, et bien d'autres. La calculabilité cherche d'une part à identifier la classe des fonctions qui peuvent être calculées à l'aide d'un algorithme et d'autre part à appliquer ces concepts à des questions fondamentales des mathématiques. Le point important est d'arriver à caractériser ce qu'une machine peut faire par opposition à ce qu'un être muni de raisonnement peut faire. Le lien entre informatique et mathématiques est en fait très fort. Il n'est point de programme sans projection du problème à résoudre dans un domaine mathématique, un domaine où des propriétés peuvent être vérifiées, prouvées, assurant que les calculs effectués sont logiquement (donc mathématiquement) fondés.

Des années après ces travaux théoriques, lorsque les progrès de l'électronique ont fait sortir les ordinateurs des centres de recherche pour les faire entrer en masse chez les particuliers, l'informatique pratique a connu un essor phénoménal, les programmes proliférant grâce à autant de développeurs professionnels qu'amateurs. Tout le monde y allait de son plus ou moins petit programme, qui pour un jeu, qui pour un utilitaire, dans un joyeux fatras de lignes de code alignées parfois dans un heureux hasard.

Bien présomptueux seraient ceux qui prétendraient « savoir programmer » sous prétexte qu'ils ont vu la syntaxe de C ou de tel autre langage. Programmer n'est pas connaître de la syntaxe. Derrière la syntaxe se cache la sémantique. Et c'est elle qui donne du sens au monde informatique. De surcroît, une fois le langage maîtrisé, il n'en reste pas moins que la conception d'algorithmes répondant à des problèmes doit être menée de manière rigoureuse et réfléchie. Rien n'est de meilleure preuve que le nombre perpétuellement renouvelé de développeurs, *a priori* formés dans des cursus d'informatique, et qui donnent lieu à des bugs en quantité constante, des tests et justifications fumeuses montrant que leur code fonctionne, voire des corrections d'erreurs apportant leurs nouvelles erreurs dans un rapport supérieur à 1 par rapport à celles effectivement corrigées.

Pour toutes ces raisons, il me semble important que le présent cours ne soit pas seulement un cours d'algorithmique, mais aussi un cours de programmation qui vous permettra de vous initier ensuite aux questions de structures de données et algorithmes « standards » que l'on trouve dans la littérature.

L'informatique est une science, la programmation est un art.

1.2 Motivations

Une question légitime peut venir en tête d'un élève ingénieur généraliste : « pourquoi dois-je avoir des cours d'informatique ? ». La réponse comporte de multiples facettes.

D'une part, l'informatique en tant qu'outil (pas forcément en tant que science) s'est imposée progressivement dans de nombreux métiers techniques et scientifiques. Suivent quelques exemples, loin de représenter une liste exhaustive. Les ingénieurs travaillant dans la simulation numérique développent des modèles et algorithmes pour résoudre leurs problèmes. Dans le domaine des systèmes embarqués, le logiciel côtoie l'électronique et la mécanique. En intelligence artificielle et exploitation des données, le développement fait bien entendu partie des activités indispensables. En cybersécurité la compréhension des mécanismes d'exécution des programmes est nécessaire pour comprendre certaines attaques et le développement peut permettre d'automatiser certaines tâches d'investigation ou de tests d'intrusion.

En second lieu, en tant qu'ingénieur généraliste, un élève ENSTA se doit de disposer de connaissances en informatique comme il en dispose en mathématiques, en mécanique ou en économie. Ces connaissances contribuent au large spectre que couvre le savoir d'un ingénieur ENSTA.

Troisième élément de réponse, il se peut fort bien que votre domaine ne soit pas du tout l'informatique, mais que vous soyez confrontés à des besoins ponctuels d'automatisation de certains traitements. Par exemple, vous avez obtenu des données d'une expérience, il faut les mettre en forme pour pouvoir les interpréter. Votre connaissance de la programmation vous permettra peut-être d'écrire le petit programme effectuant cette mise en forme, avant de retourner à votre domaine métier favori. De cette façon, vous n'aurez pas eu besoin de demander à un collègue informaticien, déjà surchargé bien sûr, de faire ce morceau de code pour vous et aurez certainement gagné du temps.

Pour terminer, au cours de leur carrière, un certain nombre d'entre vous embrasseront peut-être des responsabilités de chef de projet. Or de nos jours une immense majorité des projets d'envergure comportent des composants logiciels. À ce titre, en temps que pilote du projet, il est important que le responsable sache comprendre les problématiques des équipes qu'il dirige, les réponses qu'elles leur font, afin de ne pas passer pour un gestionnaire techniquement incompétent. Il n'est pas question d'être expert dans tous les domaines du projet mais de disposer des connaissances suffisantes pour éviter de sombrer dans une incommunicabilité qui pourrait mettre en péril l'avancée efficace du projet.

Tentons de comprendre en quoi ce cours se veut différent de ce que la majorité d'entre vous avez déjà vu. Pour un grand nombre d'entre vous, l'informatique vue dans vos précédentes années d'études était l'apprentissage du langage PYTHON pour écrire des programmes de taille modeste, de difficulté modeste, très souvent issus de problèmes mathématiques. La résolution de ces problèmes pouvait être fort compliquée en terme d'analyse mathématique mais était souvent simple en terme d'implantation informatique, en terme de structure algorithmique. L'utilisation courante de la bibliothèque NUMPY contribue à cette simplicité apparente algorithmique car elle met instantanément à disposition des fonctions effectuant des traitements non triviaux (par exemple la multiplication de matrices efficace, ou la transformée de Fourier).

Le premier cours de l'année, « Langage C », vous a appris un nouveau langage en vous présentant sa syntaxe, la sémantique de ses constructions, certains aspects techniques du modèle d'exécution (en particulier les pointeurs). Vous avez appliqué ce langage à la création de programmes modestes dont la difficulté algorithmique restait très modérée. En particulier, la distance entre les questions formulées en exercice et la traduction de leur solution en C était relativement faible.

Les buts de ce cours « Algorithmique et programmation » sont au nombre de quatre :

- apprendre à concevoir des algorithmes,
- apprendre à se poser des questions sur la spécification d'un programme (ce qui est demandé ou ce que l'on veut faire),
- acquérir une démarche systématique et scientifique :
 1. d'analyse,

2. de formalisation,
 3. de conception,
 4. d'implantation,
- planter *in fine* dans un langage de programmation.

Ce buts visent à vous permettre d'aborder des problèmes plus compliqués que ceux que vous avez majoritairement abordés jusqu'alors, à y répondre grâce à des algorithmes *corrects, maintenables et efficaces*.

1.3 Rappels d'architecture matérielle

Cette section a pour but de présenter l'architecture d'un ordinateur, support d'exécution des programmes. Elle se veut volontairement simpliste, avec une présentation extrêmement caricaturale qui ne reflète pas la complexité réelle des architectures actuelles. Son but est de rappeler que les programmes tournent dans un environnement matériel soumis à différentes contraintes en terme de ressources (mémoire, temps, etc.).

1.3.1 Traitement de l'information

La raison d'être des ordinateurs est le traitement de l'information. Une *information* est un message écrit à l'aide de *symboles*. Ce système de symboles est alors appelé *code*. Dans le monde numérique, le code utilisé est le *code binaire* composé de deux symboles (souvent notés 0/1 ou vrai/faux). Ces deux symboles correspondent à deux états de potentiel électrique dans les circuits électroniques composant les éléments d'un ordinateur.

De manière simplifiée (mais pas forcément fausse) un ordinateur est composé de trois éléments : la *mémoire* qui permet de stocker l'information, le *microprocesseur* qui permet de traiter l'information, les périphériques qui permettent d'échanger de l'information entre l'ordinateur et l'environnement extérieur. Un programme permet donc d'exprimer le « comment » du traitement de l'information.

1.3.2 La mémoire

En mathématiques les fonctions ne font pas apparaître de notion de mémoire. Par essence, une fonction est quelque chose qui s'évalue « instantanément » et toujours en la même valeur pour un même argument. Ainsi, $x \mapsto \sqrt{x}$ appliquée à 9 a toujours valu 3 et vaudra toujours 3.

Sitôt qu'une notion de temps apparaît, celle de mémoire accourt. Si l'on souhaite représenter une fonction qui incrémente (i.e. augmente de 1 à chaque fois) et retourne la valeur d'un compteur, il devient nécessaire de se « souvenir » de la valeur « au coup précédent » de ce compteur. Pour s'en « souvenir » il est nécessaire de mémoriser cette information dans une mémoire. De plus, à chaque fois que l'on appellera cette fonction avec le même compteur, un résultat différent sera retourné. La même chose se produit si l'on souhaite décrire une fonction qui représente l'état d'un interrupteur sur lequel on appuie : elle retournera «On» si l'interrupteur était avant à «Off» et «Off» s'il était à «On».

Physiquement existent différents modes de fonctionnement des mémoires. Nous en présentons succinctement trois mais il existe de nombreuses autres variantes.

- La RAM (Random Access Memory). Ce type de mémoire est accessible aussi bien en lecture qu'en écriture, ce qui en fait le candidat idéal pour la mémorisation d'informations qui évoluent dans le temps. Lorsque cette mémoire n'est plus alimentée électriquement, son contenu disparaît : elle est *volatile*.
- La ROM (Read Only Memory). Cette mémoire n'est accessible qu'en lecture. Son contenu est figé lors de la création physique du composant et ne peut plus être modifié. Son contenu ne disparaît donc pas lorsque la mémoire n'est pas alimentée : elle est *persistante*. On la retrouve par exemple dans les cartouches de jeux vidéos de consoles.
- L'EPROM (Erasable Programmable Read Only Memory). Il s'agit d'une mémoire accessible en lecture et persistante. Néanmoins, elle peut être réécrite dans des cas exceptionnels via l'utilisation d'un dispositif particulier (exposition à un rayonnement, modification de la tension d'alimentation, etc.).

On la retrouve par exemple pour mémoriser les programmes de démarrage des ordinateurs, qui doivent être disponibles, malgré la coupure d'alimentation, dès la mise sous tension. Ces programmes très particuliers sont très rarement mis à jour, ainsi le besoin d'écriture reste effectivement exceptionnel.

Au sein d'un ordinateur se trouvent différents types de mémoires, chacun dédié à une utilisation particulière répondant à un compromis entre vitesse d'accès et capacité de stockage.

- Les mémoire de masse (disques mécaniques). Fournissant d'énormes volumes de stockage (en téraoctets) ce sont des mémoires relativement lentes avec ~ 10 ms de temps d'accès. Devenues peu coûteuses, elles constituent les disques durs de nos machines.
- Les mémoire de masse (disques SSD). Ces mémoires, basées sur des technologies ressemblant à des EPROMs se sont grandement développées ces dernières années au point de fournir désormais plusieurs centaines de gigaoctets de stockage pour un temps d'accès $\sim 0.1 - 0.3$ ms. Elles constituent les « disques durs » des ordinateurs récents de plus ou moins haut de gamme.
- La mémoire vive (RAM). Externe au microprocesseur, avec un temps d'accès $\sim 40 - 50$ ns, c'est la mémoire « de travail » de la machine, celle dans laquelle sont traitées les informations manipulées par les programmes.
- La mémoire cache. Il s'agit de mémoire très rapide, accessible en $\sim 5 - 10$ ns qui se trouve noyée dans le microprocesseur. En petite quantité (jusqu'à quelques mégaoctets), elle est utilisée pour mémoriser des informations auxquelles le microprocesseur accède très souvent au cours de l'exécution d'un programme. Ce peut être aussi bien des instructions que des données. Du fait de sa petite taille, le cache doit régulièrement être déchargé d'informations moins utiles pour les remplacer par celle devenues plus souvent utilisées en fonction d'où se trouve l'exécution du programme.
- Les registres. Ce sont les mémoires les plus rapides de l'ordinateur, logées dans le cœur du microprocesseur, avec un temps d'accès de l'ordre de la nanoseconde. Ce sont sur ces mémoires qu'opèrent une grande majorité des instructions du microprocesseur. En très petit nombre (de quelques uns, à quelques dizaines), les registres sont des ressources précieuses que le compilateur tentera, lors de la création de l'exécutable, d'attribuer le plus efficacement aux « variables » du programme.

Une question légitime est « pourquoi n'a-t-on pas que des registres, ce serait plus rapide ? ». La réponse est en partie une question de coût. Augmenter le nombre de registres revient à augmenter la surface du microprocesseur. Plus on augmente cette surface, plus on augmente le risque de présence de défauts physiques dans le silicium de ce dernier. Ainsi, plus de rebuts implique une élévation du coût. Poussé au paroxysme, il deviendrait difficile, avec les technologies actuelles, d'obtenir des processeurs fonctionnels. L'autre élément de réponse à cette question provient de la chaleur dissipée qui est proportionnelle à la surface du circuit. Augmenter le nombre de registres revient à augmenter la surface du circuit (plus de transistors), donc augmenter la chaleur qu'il dissipe.

1.3.3 Le microprocesseur

C'est l'unité « qui travaille », celle qui transforme effectivement l'information, la mémoire ne faisant que la stocker. De manière très simplifiée, on peut idéaliser la représentation d'un microprocesseur en deux unités.

- L'unité *arithmétique et logique* (UAL / ALU) qui exécute les instructions. Par exemple, c'est elle qui va additionner le contenu de deux registres et stocker le résultat dans un de ces registres, dans un autre, voire en mémoire.
- L'unité de *contrôle* qui se charge du décodage et du séquencement des instructions. Par exemple, c'est elle qui va déterminer qu'un mot binaire représente l'ordre d'additionner deux registres et lesquels.

Les instructions que le microprocesseur exécute sont élémentaires et loin des constructions mises à disposition par les langages de haut niveau. Ainsi, au niveau du microprocesseur la notion de fonction n'existe pas : c'est le compilateur du langage qui va avoir traduit cette construction en une suite d'instructions ayant cette sémantique. À titre d'exemple, les instructions peuvent effectuer des additions, des décalages, des copies entre registres, des copies entre les registres et la mémoire, des sauts, etc. Il existe également des instructions plus exotiques dont on comprendrait l'intérêt dans un cours de compilation ou de système d'exploitation.

Le désavantage d'écrire manuellement des programmes avec instructions est double. D'une part chaque microprocesseur a son propre jeu d'instructions. Ainsi, un programme écrit pour un microprocesseur n'est pas transportable sur un autre modèle sans réécriture complète. D'autre part, vu le caractère élémentaire des instructions, on imagine bien qu'il va falloir écrire un très grand nombre d'instructions, ce qui augmente le temps de développement et le risque d'erreurs. Pour terminer, l'absence de constructions de haut niveau, telles que les fonctions pour ne citer qu'elles, rend la programmation très ardue, l'encodage devant être réalisé manuellement.

1.4 Quelques rappels de C

Le but de ce chapitre n'est pas de se substituer au cours de langage C mais de rappeler quelques éléments essentiels qui serviront durant le cours.

Le choix de ce langage est motivé par trois éléments. D'une part c'est un langage que vous avez déjà étudié, dont vous connaissez désormais la syntaxe. D'autre part, planter ce cours dans ce langage vous permettra de le pratiquer plus longuement et renforcera votre aisance pour les prochains cours. Pour terminer, C étant un langage de haut niveau ... suffisamment bas niveau, il ne fournit pas de constructions « magiques » masquant la réalité et la complexité algorithmique derrière des mécanismes mystérieux et des fonctions toutes faites. Ainsi, n'importe quel programme écrit en C pourra aisément être traduit dans un autre langage alors que l'inverse peut nécessiter un travail conséquent voire colossal.

Une fois que l'on est conscient du coût de certaines structures de données, de certaines fonctions d'une bibliothèque, que l'on est capable de développer la démarche permettant de concevoir un algorithme avec des constructions élémentaires, on peut sereinement utiliser d'autres langages plus adaptés et plus faciles pour certaines tâches que l'on aura donc identifiées en pleine conscience.

Les constructions nécessaires pour ce cours seront relativement restreintes. Seront nécessaires les expressions de base (arithmétiques, conditionnelles), les boucles (**for**, **while**), les fonctions, les tableaux avec allocation statique et dynamique et la lecture/écriture de fichiers. On fera également usage de temps à autre du passage d'arguments par adresse dans le cas où il sera nécessaire de retourner plusieurs résultats.

1.4.1 Compilation

La création d'un exécutable passe par la compilation du code source C. Le compilateur utilisé dans ce cours est **gcc** et la compilation d'un fichier source **foo.c** est obtenue par la ligne de commande suivante :

```
gcc -Wall -Werror -Wfatal-errors foo.c
```

dans laquelle trois options figurent de manière importante :

- **-Wall** : permet d'activer la vérification de tous les types d'avertissement. Un avertissement est émis lorsque le compilateur détecte une situation qui ne l'empêche pas de compiler (en faisant parfois des hypothèses arbitraires pour y arriver) mais qu'il considère comme anormale. Ce peut être des omissions, des traitements par défaut, des incohérences de types, etc. Quoi qu'il en soit, de tels avertissements sont dans 99% des cas (sinon plus) des erreurs latentes. Le programme obtenu après la compilation pourra fonctionner dans un grand nombre de cas et échouer dans des cas limites, pourra ne fonctionner que sur certaines machines, pourra ne fonctionner que dans un certain environnement logiciel, etc. Il convient donc d'éliminer toutes les sources d'avertissement.
- **-Werror** : permet d'élèver les avertissements au rang d'erreurs. Cela permet de se prémunir contre la tentation de laisser des avertissements non résolus puisque la compilation échouera en présence d'avertissements.
- **-Wfatal-errors** : arrête la compilation à la première erreur détectée. Par défaut, lorsqu'une erreur est détectée, **gcc** tente de continuer la compilation quand même. Malheureusement, pour y parvenir il doit faire des suppositions pour se relever de l'erreur détectée et ces suppositions peuvent entraîner d'autres erreurs en cascade. Il en résulte une interminable suite d'erreurs affichées en résultat de compilation, qui n'ont pas forcément de raison d'être une fois la première corrigée et qui inflige généralement un

sérieux coup au moral tant on a l'impression que tout est faux. Régions les erreurs une par une, sérialisons les problèmes...

Par défaut, l'exécutable généré se nomme `a.out` et peut être invoqué via le terminal par :

`./a.out`

Il est également possible de donner un nom spécifique à l'exécutable au moment de la compilation en spécifiant l'option `-o` suivi du nom désiré.

`gcc -Wall -Werror -Wfatal-errors -o foo.x foo.c`

générera un exécutable nommé `foo.x` qu'il sera possible d'invoquer dans le terminal par : `./foo.x`. Il convient d'être prudent et d'éviter absolument l'inattention fatale :

`gcc -o hello.c hello.c`

qui provoque la destruction du fichier source puisqu'il se trouve écrasé par l'exécutable résultat de la compilation.

1.4.2 Rappels de syntaxe

Déclaration/définitions

En C les variables sont typées à leur déclaration. Ainsi, déclarer une variable nécessite l'énoncé de son type suivi de son nom suivi d'une éventuelle initialisation.

```
unsigned int i, j = 27 ;
short t[5] ;
```

Le code ci-dessus déclare deux variables `i` et `j` de type entier, `i` n'étant pas initialisée, `j` étant initialisée à 27. Il déclare ensuite `t` comme un tableau non initialisé (donc contenant des valeurs *a priori* quelconques) d'entiers courts (sur 16 bits).

Les fonctions sont définies en spécifiant leur type de retour suivi du nom de la fonction suivie, entre parenthèses, des types et noms des arguments suivis entre accolades du corps (instructions) de la fonction.

```
char* make_string (unsigned int len, char init) { ... }
```

Le code ci-dessus définit une fonction `make_string` retournant un pointeur sur caractères, ayant deux arguments `len` et `init`, respectivement de types entier non signé et caractère.

La fonction `main`

Tout programme doit comporter *une et une seule* fonction nommée `main` qui sera le point d'entrée du programme (la première fonction appelée). La fonction `main` peut avoir deux définitions possibles :

- `int main () { ... }`

Dans ce cas elle ne reçoit pas d'arguments en provenance de la ligne de commande.

- `int main (int argc, char *argv[])` { ... }

À l'inverse, les arguments fournis sur la ligne de commande sont disponibles sous forme de *chaînes de caractères* dans le tableau `argv`. La longueur de ce tableau est donnée par `argc`. La première entrée de ce tableau contient toujours le nom + chemin de la commande lancée. Les « vrais » arguments commencent à partir de la seconde entrée. Il y a donc `argc - 1` « vrais » arguments.

Si l'on invoque un programme au terminal par `/bin/ls -l /tmp/` alors `argv[0]` désignera la chaîne `"/bin/ls"`, `argv[1]` la chaîne `"-l"` et `argv[2]` la chaîne `"/tmp/"`.

Les arguments de la ligne de commande étant des chaînes de caractères, lorsque ceux-ci *représentent* des scalaires il convient de les convertir dans le type adéquat en utilisant les fonctions dédiées de la bibliothèque standard, nécessitant l'inclusion de `<stdlib.h>` :

- `atoi` string → int

- `atol` : string → long int
- `atoll` : string → long long int
- `atof` : string → float

La fonction `main` doit toujours retourner un entier qui représente le code de retour du programme. Par convention, 0 signifie que le programme considère qu'il n'a pas rencontré d'erreur. Toute autre valeur signifie l'inverse. Par convention certaines valeurs correspondent à des cas d'erreur bien identifiés. La lecture d'un cours dédié au langage C permettra d'approfondir la question.

Le respect de la convention du code de retour (au moins 0 et $\neq 0$) est très importante afin de pouvoir utiliser les programmes dans des scripts. Avant d'exécuter chaque commande du script, on peut ainsi vérifier que l'exécution de la commande précédente a été un succès. Cela permettra, par exemple, de ne pas exécuter un `rm -Rf` si le `cd` précédent a échoué, évitant ainsi de détruire le répertoire courant...

Lorsqu'un programme est interrompu par le système d'exploitation à cause d'une violation d'accès ("segmentation fault" par exemple), c'est le système qui provoque l'émission d'un code de retour différent de 0 (puisque le programme ayant été tué, il n'a pas pu exécuter d'instruction `return`).

Types de base

Les types de base (scalaires) sont `char`, `int`, `float` et `double`. Certains peuvent être affectés par des modificateurs de taille ou de présence de signe : `signed`, `unsigned`, `short`, `long`, `long long`.

Les booléens sont représentés par des entiers avec la convention que $0 \equiv \text{faux}$ et $\neq 0 \equiv \text{vrai}$. Néanmoins, l'utilisation de `#include <stdbool.h>` permet l'ajout d'un nom de type `bool` et des valeurs `true` et `false`. Ces booléens restent représentés *in fine* par des entiers, cependant l'utilisation de ces définitions facilite grandement la relecture et la maintenabilité des programmes.

Constructions de contrôle

Le langage dispose de trois constructions de boucles et de deux formes de conditionnelles, le `if` pouvant omettre la partie `else` en cas de traitement vide.

- `while (...) { ... }`
- `do { ... } while (...) ;`
- `for (...) { ... }`
- `if (...) { ... }`
- `if (...) { ... } else { ... }`
- `switch (...) { case ... : ... break ; case ... : ... break ; default : ... break ; }`

Il faut noter que le cas `default` de la construction `switch` est optionnel.

Entrées/sorties

L'affichage est réalisé grâce à la fonction `printf` de la bibliothèque standard de C et nécessite l'inclusion de `<stdio.h>`. Cette fonction prend en premier argument une chaîne (appelée *format*) dans laquelle les séquences débutant par un `%` seront remplacées par l'écriture textuelle des valeurs des arguments suivants, en accord avec l'interprétation qu'impose le `%`. À titre de rappel, la liste suivante fournit une liste non exhaustive de séquences `%` légales.

- `%d` → un `int`
- `%ld` → un `long int` en décimal
- `%u` → un `unsigned int` en décimal
- `%x` → un `int` en hexadécimal
- `%f` → un `float`
- `%lf` → un `double`
- `%e` → un `double` en notation scientifique
- `%.7lf` → un `double` avec 7 chiffres après la virgule

- %07d → un **int** en décimal sur 7 chiffres (remplissage frontal avec des 0)
- % 7d → un **int** en décimal sur 7 chiffres (remplissage frontal avec des espaces)
- %c → un **char** (comme caractère ASCII, pas comme entier)

Ainsi, printf ("%d et %f\n", 4, 4.0 + 5) ; affichera la chaîne « 4 et 9.000000 » suivie d'un retour à la ligne.

L'opération duale, la récupération d'une saisie sur l'entrée standard est effectuée par la fonction **scanf** qui nécessite également l'inclusion de <stdio.h>. Cette fonction prend en premier argument un format suivi des *adresses* où stocker le résultat de l'interprétation des entrées saisies en accord avec le format. Cette fonction utilise donc le passage d'arguments par adresse dès lorsqu'il s'agit de saisir de scalaires (entiers, flottants).

Afin de ne pas rentrer dans le fonctionnement compliqué de **scanf**, on vous conseille (demande) de ne pas mettre d'autre texte que les séquences % dans le format. Dans le cas contraire, le résultat obtenu pourrait être très différent que ce qui est intuitivement attendu.

Si le texte saisi depuis l'entrée standard ne peut pas être interprété selon le format attendu (par exemple, utilisation d'un %d et entrée de l'utilisateur « ab21 »), alors le résultat de **scanf** est imprévisible.

Ainsi le programme **input.c** ci-dessous produira les résultats qui suivent en fonction des entrées correctes ou incorrectes obtenues sur l'entrée standard.

```

input.c

#include <stdio.h>

int main ()
{
    int i, j ;
    scanf ("%d %d", &i, &j) ;
    printf ("i: %d, j: %d\n", i, j);
    return (0) ;
}

$ gcc -Wall -Werror -Wfatal-errors input.c -o input
$ ./input
45 67
i: 45, j: 67
$ ./input
5
FHG
i: 5, j: 0
$ ./input
DFG 6
i: 0, j: 0

```

Allocation statique/dynamique

Il est possible d'allouer de la mémoire de deux façons. La première est l'allocation *statique* qui est utile lorsque la taille de la zone mémoire est connue au moment de la compilation et de taille relativement modeste. Cette seconde condition est nécessaire pour éviter soit :

- d'augmenter la taille de l'exécutable puisque (sous condition d'être initialisée) si cette zone mémoire représente une variable globale, elle se retrouvera textuellement dans ce dernier,
- de provoquer un débordement de pile si cette zone mémoire représente une variable locale.

Enfin, dans le cas d'une variable locale, il ne faut pas devoir retourner la zone allouée comme résultat de la fonction (par exemple si c'est un tableau) car elle devient inaccessible à la fin de la fonction.

```

#define SIZE (20)

```

```

char name[SIZE] ;

void f () {
    int t[5];
    ...
}

```

Lorsque la taille de la zone mémoire à allouer n'est pas connue au moment de la compilation, si elle est volumineuse ou doit être retournée par la fonction, il faut procéder à une allocation *dynamique*. L'allocation est effectuée par la fonction **malloc** qui prend en argument le *nombre d'octets* à allouer et retourne l'adresse (pointeur) de la zone effectivement attribuée. Une allocation doit être ultérieurement suivie d'une libération en utilisant la fonction **free**. Ces deux fonctions nécessitent l'inclusion de <stdlib.h>.

```

double* make_array (int size) {
    double *t ;
    t = malloc (size * sizeof (double)) ;
    if (t == NULL) { ... /* Gérer l'erreur. */ }
    ...
    return (t) ; /* Libérer avec free ultérieurement. */
}

```

En cas d'échec de l'allocation, le pointeur invalide **NULL** est retourné. Il convient donc de tester le pointeur retourné par **malloc** avant d'accéder à la zone qu'il désigne.

1.5 Algorithme

Avant de donner une définition possible de ce qu'est un *algorithme*, commençons par voir ce que *ce n'est pas*.

L'analogie courante est de dire que c'est comme une recette de cuisine. Je n'aime personnellement pas cette analogie. En effet une recette est suivie par l'être humain qui tente de mitonner un bon petit plat. Or, en l'occurrence ce n'est pas à vous de suivre les ordres mais à la machine. Votre travail en temps que concepteur de programme est d'élaborer cette suite d'ordres, pas de la suivre. Cette différence de point de vue me semble fondamentale car jamais il n'est précisé que cette analogie doit être considérée dans le cas où l'on est un chef cuisinier en train de concevoir une recette. D'autre part, cette analogie est fort approximative et ne rend pas compte de la rigueur dont doit faire preuve l'énoncé d'un algorithme. N'emploit-on d'ailleurs pas le terme « recette de cuisine » pour désigner une méthode un peu empirique, un bricolage en lequel on espère plutôt que l'on ne croit ?

La seconde confusion commune est de mettre au même rang un algorithme et un langage de programmation (ou de manière moins erronée un programme écrit dans un langage). Un langage de programmation n'est qu'un ensemble de mots, de constructions qui permettent d'implanter un algorithme en vue d'obtenir un programme qu'il sera possible d'exécuter.

Considérons plutôt un algorithme comme une méthode de passage d'un ensemble de *données* D à un ensemble de *résultats* R par l'intermédiaire de *calculs*. C'est donc une *fonction*, qui peut être partielle dans le cas où des erreurs sont possibles. De plus, c'est une fonction *calculable* puisque son « exécution » par une machine doit retourner un résultat en temps fini.

Ainsi, la fonction « qui renvoie vrai s'il fera beau demain » n'est pas calculable. La fonction non calculable sans doute la plus connue est celle qui dit si un programme (quelconque) termine ou non avec une entrée donnée. Ce problème est appelé *problème de l'arrêt*.

Lorsque l'on cherche à concevoir un algorithme répondant à un problème, la première question à se poser est :

« Qu'est-ce que j'ai, qu'est-ce que je veux ? »

Le « Comment le faire ? » ne peut venir qu’après. Cette question initiale doit permettre de déterminer deux informations essentielles.

- Les données à traiter, que l’on nomme *entrées*. Ce sont les hypothèses du problème, elles sont imposées. Les changer revient à modifier le problème.
- Les résultats à obtenir, que l’on nomme *sorties*. Ils seront synthétisés par l’algorithme à partir des entrées.

Nous reviendrons plus en détail en section 2.4 sur l’importance de déterminer précisément entrées et sorties. Le lien entre les entrées et les sorties est de deux natures. D’une part celui énoncé par le futur utilisateur : on parle de *spécification*. D’autre part existe le lien effectivement décrit par le calcul, donc par l’algorithme (qui aboutira à un programme). Un algorithme répondra donc *correctement* à un problème posé si ce qu’il calcule satisfait bien la spécification. Dit autrement, si l’algorithme calcule bien ce qui est attendu, ce qui est exprimé par le besoin.

Nous reviendrons plus en détail sur la notion de correction dans la section 6.1 dans laquelle il sera question de preuves d’algorithmes.

Illustrons, pour le moment intuitivement, quelques éléments évidents mais importants dans la conception d’un algorithme puis d’un programme. Considérons le problème « Je veux additionner 2 entiers ». Un premier algorithme (et son implantation en C) pourrait être le suivant.

```
foo.c

#include <stdio.h>

int addition (int x, int y) {
    return x + y ;
}

int main () {
    printf ("%d\n%d\n", addition (0, 0), addition (4, 6)) ;
    return 0 ;
}
```

L’exécution de ce programme conduit au résultat suivant qui nous permet de conclure que « ça marche », l’algorithme répond bien au problème posé.

```
$ gcc -Wall -Werror -Wfatal-error foo.c
$ ./a.out
0
10
```

Considérons maintenant un autre algorithme pour ce même problème et le résultat de son exécution.

```
foo.c

#include <stdio.h>

int addition (int x, int y) {
    int res = x ;
    for (int i = 0; i < y; i++)
        res = res + 1 ;
    return res ;
}

int main () {
    printf ("%d\n%d\n", addition (0, 0), addition (4, 6)) ;
    return 0 ;
}
```

```
$ gcc -Wall -Werror -Wfatal-error foo.c
$ ./a.out
0
10
```

L'affichage nous montre que ce programme fonctionne également correctement. On remarque néanmoins que cette seconde version est plus longue, en terme de code source, que la première. La longueur d'un code source n'est pas forcément liée à sa rapidité d'exécution, mais un rapide examen montre que dans notre cas, ce second programme sera plus long à exécuter que le premier. En effet, au lieu d'une simple addition, on constate que pour additionner x et y il va falloir faire y tours de boucle, invoquant chacun une simple addition.

Nous voyons donc poindre ici le fait qu'il n'existe pas un seul algorithme pour répondre à un problème et que certains sont plus *efficaces* que d'autres. Comme nous l'approfondirons en section 4, l'efficacité pourra être mesurée en terme de temps d'exécution mais également en terme d'espace mémoire requis. Nous allons naturellement chercher à concevoir des algorithmes efficaces. L'intérêt évident est une attente réduite pour obtenir les résultats, mais il existe également un intérêt moins évident : l'empreinte écologique du programme obtenu. En effet, plus un programme tournera longtemps pour fournir son résultat, plus il nécessitera d'énergie. Plus il nécessitera de ressources (mémoire, espace disque), plus il nécessitera d'énergie. En effet, les processeurs et systèmes d'exploitation modernes sont capables d'adapter la consommation énergétique en fonction des besoins de ressources et de calcul des programmes tournant sur une machine.

Pour terminer ce chapitre, considérons une dernière version de notre programme.

```
foo.c

#include <stdio.h>

int addition (int x, int y) {
    return (x * y) ;
}

int main () {
    printf ("%d\n%d\n", addition (0, 0), addition (4, 6)) ;
    return 0 ;
}
```

```
$ gcc -Wall -Werror -Wfatal-error foo.c
$ ./a.out
0
24
```

Le test avec $(0, 0)$ n'indique aucun dysfonctionnement. Néanmoins, celui avec $(4, 6)$ révèle clairement une erreur. Cela démontre que pour tester un programme, il ne faut pas se contenter d'un seul test. Plusieurs doivent être effectués, choisis de manière avisée qui plus est. En effet, tester $(0, 0)$ n'est pas très discriminant puisque $0 + 0$, $0 - 0$ et 0×0 sont tous égaux à 0. Donc ce test ne permet pas de détecter un remplacement erroné de l'addition par une autre opération arithmétique.

Séance 2

Spécification et domaines

2.1 Spécification

Comme brièvement introduit en section 1.5, on appelle *spécification* l'expression du besoin de la part de l'utilisateur, du client. Il existe plusieurs formes de spécifications. Cela peut consister en une simple formule mathématique qui est alors quasiment directement traduisible en une expression arithmétique du langage de programmation cible. Ce peut également être une formule mathématique plus complexe nécessitant une « mise en forme informatique ». Pour terminer, ce peut être un problème décrit de manière imprécise, dans un langage naturel et informel.

En fonction de la nature et de la précision de la spécification, un travail plus ou moins important de *modélisation* et de *raffinement* sera nécessaire.

2.1.1 Une spécification simple

Soit l'expression de besoin : « Écrire un programme permettant de convertir une température en degrés Fahrenheit vers des degrés Celcius ».

Une rapide recherche dans un livre de physique fournit la formule de conversion : $x^{\circ}\text{F} \longrightarrow \frac{5}{9}(x - 32)^{\circ}\text{C}$. Il est alors possible de la traduire simplement en C en ayant toutefois identifié que les domaines d'entrée et de sortie sont des flottants (puisque les températures sont des réels et non des entiers).

```
float ftoC (float t) {
    return (5.0 / 9.0 * (t - 32.0));
}
```

2.1.2 Une spécification intermédiaire

Considérons la spécification « Écrire un programme permettant de calculer $f(n)$ sachant que : »

$$\begin{cases} f(0) = 1 \\ f(n) = \prod_{k=1}^n k \end{cases}$$

La formule donnée contient deux cas. Pour autant, est-il nécessaire que l'algorithme en comporte également deux ? Est-il possible de faire en sorte que le cas $f(0)$ soit géré par le cas général ?

D'autre part, le produit introduit une notion de répétition. Il faut donc planter un mécanisme *d'itération*. Le fait-on au travers d'une boucle ou bien au travers d'une récursion ?

Les choix faits pour répondre à ces deux questions peuvent mener à des algorithmes très différents. Ci-dessous, dans le programme de gauche une boucle est utilisée qui permet de ne pas rendre explicite le cas $f(0)$. En effet, l'initialisation à 1 de la variable `res` permet de faire en sorte que la boucle ne tourne pas,

le résultat attendu est obtenu pour 0. Dans le programme de droite, une récursion remplace la boucle et le cas $f(0)$ devient explicite comme cas d'arrêt de la récursion.

```
int f (int x) {                                int f (int x) {  
    int res = 1 ;                            if (x <= 0) return 1 ;  
    while (x > 1) {                        return (x * f (x - 1)) ;  
        res = res * x ;  
        x-- ;  
    }  
    return res ;  
}
```

2.1.3 Une spécification pas triviale

Lorsque le besoin est énoncé de façon très générale, la spécification peut devenir compliquée et l'idée de l'algorithme n'est généralement pas immédiate. Par exemple :

- « Écrire un programme permettant de calculer les déformations d'une structure. »
- « Écrire un programme permettant de naviguer sur le WEB. »
- « Écrire un programme permettant de simuler les cours de la bourse. »

Dans de tels cas un effort de modélisation est indispensable : il faut représenter le problème de manière plus formelle. Ce problème étant naturellement compliqué, il va être nécessaire de le décomposer itérativement en sous-problèmes plus simples jusqu'à temps d'obtenir des (sous) problèmes que l'on sait résoudre facilement, dont les algorithmes se traduisent aisément dans un langage de programmation.

De nos jours, de nombreux logiciels (navigateurs WEB, systèmes d'exploitation, outils multimédias, etc.) dépassent les millions voire les dizaines de millions de lignes de code. Leur conception est forcément passée par d'innombrables étapes de décomposition, de *raffinement*.

2.1.4 Conseils

Lorsque l'on est confronté à la conception d'algorithmes il est indispensable de *ne pas commencer par programmer*. En effet, l'absence de réflexion préalable, de modélisation, de décomposition provoque très souvent l'oubli de cas particuliers, une mauvaise compréhension du problème posé, le développement d'algorithmes inefficaces, non maintenables et inutilement compliqués, voire un échec total.

Ce type de méthode de développement aboutit souvent à des programmes remplis de cas particuliers issus de corrections successives de problèmes identifiés au fur et à mesure, corrections pouvant être incompatibles entre elles et amenant encore plus de corrections et de cas particuliers.

De même, il ne faut pas penser son algorithme en terme d'un langage de programmation particulier et des fonctions de sa bibliothèque standard. Un algorithme doit être agnostique envers le langage qui servira à son implantation. Ne pas se soumettre à cette règle peut amener à deux problèmes.

D'une part l'algorithme obtenu à partir de cette conception peut se trouver difficile à planter dans un autre langage qui ne dispose pas des facilités utilisées. Il est alors difficilement *portable*.

D'autre part, l'algorithme obtenu peut être inefficace car conçu à partir des fonctions que l'on *connaît* du langage utilisé et que l'on s'est *senti obligé* d'utiliser. Par exemple, lire le contenu d'un fichier de données ligne par ligne, sous prétexte que l'on ne connaît que cette fonction de lecture n'est pas forcément une solution adaptée et peut complexifier tout le reste du traitement alors qu'une lecture mot par mot ou caractère par caractère aurait pu grandement simplifier le problème.

Au contraire, il est nécessaire d'étudier comment on résoudrait soi-même le problème, « à la main », en faisant des exemples, des essais, des dessins, des schémas ou tout autre artefact permettant d'exercer un esprit posé d'*analyse* et de *synthèse*. Les meilleurs et indispensables outils sont donc le papier et le crayon (la gomme éventuellement, tout dépend de si l'on supporte les ratures ^②).

Cette phase de réflexion, qui peut sembler une perte de temps au débutant, s'avère toujours un énorme gain de temps lors de l'implantation. Il va sans dire (ce qui va sans dire va toujours mieux en le disant), que ce conseil s'applique également en examen...

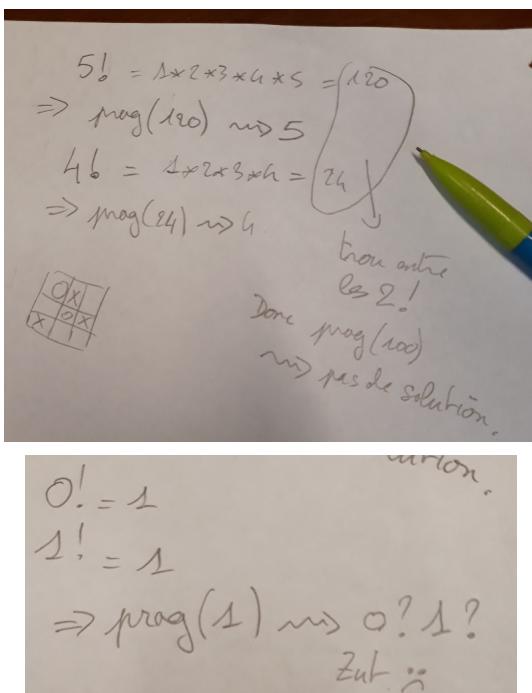
2.1.5 Un premier exemple

Considérons la spécification « Écrire un programme qui prend un entier et dit de quoi sa valeur absolue est la factorielle ».

Devant une expression aussi peu claire, il convient de reformuler le problème un peu plus clairement : « Soit $x \in \mathbb{Z}$, trouver y tel que $y! = |x|$ ». Il apparaît alors qu'il est demandé d'écrire la fonction factorielle inverse.

Pour répondre à la sempiternelle première question (« Qu'est-ce que j'ai, qu'est-ce que je veux ? »), nous commençons par déterminer les domaines d'entrée et de sortie :

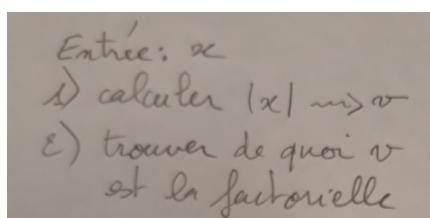
- mes entrées : un nombre entier positif, nul ou négatif,
- mes sorties : un nombre entier positif.



Prenons un peu de temps à faire quelques essais sur papier. À force d'avoir écrit la factorielle au moins 1000 fois, nous savons que $5!$ est égal à 120. Donc si nous appelons notre programme avec 120, il devra nous retourner 5. De même, $4!$ est égal à 24, donc si nous appelons notre programme avec 24 il devra nous retourner 4. Il apparaît alors qu'entre 120 et 24 il n'y a pas de nombre qui soit une factorielle : il y a un trou. Donc si l'on appelle notre programme avec 100, le programme ne trouvera pas de solution. Il faut donc revenir sur le domaine des sorties que l'on avait initialement défini : c'est un nombre entier positif *ou une erreur*.

Continuons la réflexion à coup d'essais et de petits dessins. Par convention, $0!$ est égal à 1. Mais, par construction, $1!$ est aussi égal à 1. Donc si nous appelons notre programme avec 1, que doit-il répondre ? 0 ou 1 ? Il n'y a pas de meilleure réponse possible. Donc il doit retourner une *autre* erreur dans les cas où il est appelé avec 0, 1 ... ou -1 puisque le problème posé prend la valeur absolue de l'entrée.

Maintenant que nous avons déterminé les domaines d'entrée et de sortie, nous pouvons nous atteler à la question du « comment » : l'algorithme en tant que tel. Pour ce faire, nous allons *itérativement* décomposer le problème en *sous-problèmes*.



Nous avons identifié 2 sous-problèmes : celui de calculer la valeur absolue de l'entrée x , que l'on nommera v et celui de trouver de quoi v est la factorielle. Nous allons alors traiter chacun d'entre eux séparément, séquentiellement et *localement*.

Entrée: x
 1) calculer $|x| \rightarrow v$
 2) trouver de quoi v est la factorielle

2 cas pour x :
 $x < 0 \rightarrow v = -x$
 $x \geq 0 \rightarrow v = x$

On remarque que nous avons fait un choix : nous aurions pu à la place considérer les cas $x \leq 0$ et $x > 0$.

Le sous-problème étant suffisamment simple, nous pouvons revenir à celui laissé en suspens qui devient le problème courant : calculer la factorielle inverse. Par l'analyse du domaine des entrées, nous avons identifié deux cas à traiter : $v = 0$ ou 1 et le reste.

2 cas pour x :
 $x < 0 \rightarrow v = -x$
 $x \geq 0 \rightarrow v = x$

2 cas pour v :
 $v = 0 \text{ ou } 1 \rightarrow$ erreur, on ne sait pas dire si résultat est 0 ou 1
 sinon

- calculer $1!, 2!, 3!, \dots$
 - vérifier à chaque étape si = nombre initial
 - si oui: trouvé, sinon continuer
 - arrêt si $i! >$ nombre initial

Nous commençons par le calcul de la valeur absolue. Nous identifions deux cas et déterminons pour chacun d'entre eux quel est le résultat attendu pour le sous-problème courant. Nous avons d'une part le cas $x < 0$ qui induit $v = -x$ et d'autre part $x \geq 0$ qui donne $v = x$.

Nous décomposons donc ces deux cas. Quand v est égal à 0 ou 1, comme on ne sait pas quelle valeur retourner, on générera une erreur. Ce cas étant réglé, il nous reste à traiter celui des autres valeurs de v . Il devient donc notre sous-problème courant. Notons que nous raisonnons toujours sur chaque sous-problème *localement*.

2 cas pour x :
 $x < 0 \rightarrow v = -x$
 $x \geq 0 \rightarrow v = x$

2 cas pour v :
 $v = 0 \text{ ou } 1 \rightarrow$ erreur, on ne sait pas dire si résultat est 0 ou 1
 sinon

- calculer $1!, 2!, 3!, \dots$
 - vérifier à chaque étape si = nombre initial
 - si oui: trouvé, sinon continuer
 - arrêt si $i! >$ nombre initial

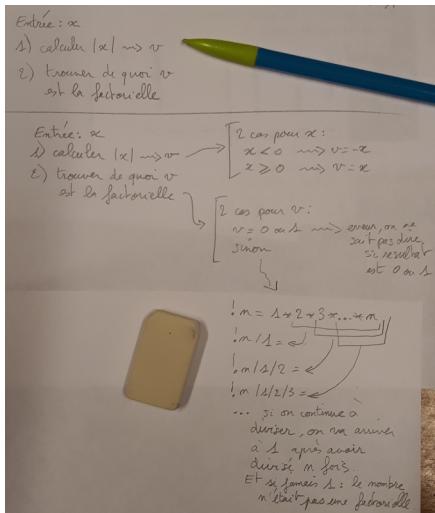
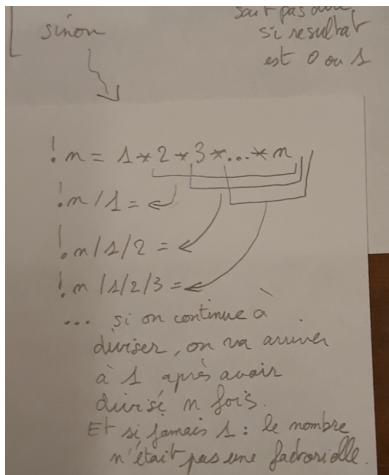
Un algorithme relativement simple consiste à calculer $1!$, à vérifier si c'est égal à v , si oui, on a trouvé le résultat recherché, si non, on recommence en calculant $2!$ et ainsi de suite. Si à une étape la factorielle calculée est supérieure strictement à v , alors plus la peine de continuer, on ne trouvera pas de résultat valide, on s'arrête en levant une erreur.

On voit ici apparaître une structure itérative qui va être décrite par une boucle. On remarque également que l'on peut « optimiser » cet algorithme en ne calculant ni $0!$ ni $1!$ (on ne gagne pas grand chose, mais c'est inutile de le faire, donc autant ne pas faire de choses inutiles).

Cet algorithme fonctionne *a priori*. Néanmoins, on a une furieuse impression de recommencer toujours les mêmes calculs : en calculant $4!$ on recalcule $3!$ pour le multiplier par 4, or on l'a déjà calculé à l'itération précédente. Et ce motif de gaspillage d'énergie se retrouve à toutes les itérations. Cette remarque devrait nous permettre de trouver une solution plus efficace.

En effet, $n! = 1 \times 2 \times 3 \times \dots \times n$. Donc si l'on divise successivement v par 1, puis 2, puis 3, etc. il va nous rester $2 \times 3 \times \dots \times n$, puis $3 \times \dots \times n$ etc. jusqu'à obtenir 1 si v était bien une factorielle. Donc, à chaque étape il suffit de vérifier si le résultat de la division est égal à 1. Si oui, alors le résultat est le diviseur courant. Si non, on continue et on s'arrête en erreur si le résultat de la division vaut 0.

Nous avons ici encore une simple boucle mais plus besoin de calculer les factorielles successives qui elles aussi auraient nécessité une boucle chacune. L'efficacité est donc *intrinsèquement* meilleure. Et quitte à gagner un peu, autant ne pas diviser par 1 et faire commencer notre boucle avec le diviseur 2.



Nous venons de trouver la solution à notre dernier sous-problème ce qui clôt les étapes de raffinement du problème global. À chaque étape, nous avons raisonné localement sans se soucier du problème global (puisque'il a été décomposé en sous-problèmes) ni des problèmes en suspens (puisque l'on a effectué un découpage adéquat au niveau de raffinement précédent pour obtenir des sous-problèmes « qui se combinent bien »). Les questions et les solutions sont donc venues petit-à-petit en simplifiant le problème initial. Nous obtenons au final une solution *globale* qui s'implante sans grande réflexion dans n'importe quel langage. Cette solution a été conçue sans *a priori* sur le langage d'implantation. Les aspects langage sont apparus à la toute fin du développement, une fois l'algorithme conçu.

```
#include <stdio.h>
#include <stdlib.h>

int factinv (int x) {
    int i, div, v ;
    /* Compute v as |x|. */
    if (x < 0) v = -x ;
    else v = x ;
    /* Check error. */
    if ((v == 0) || (v == 1)) return -1 ;
    div = v ;
    for (i = 2; i <= v; i++) {
        div = div / i ;
        if (div == 1) return i ;
    }
    return -1 ;
}
```

2.2 Après l'écriture

Notre algorithme est désormais conçu, notre programme est finalement écrit, il reste néanmoins à le faire fonctionner. Pour ce faire, nous allons le *tester* en fournissant des entrées et en vérifiant que les sorties sont celles attendues. Nous appliquons alors un *jeu de tests*.

```
$ gcc -Wall -Werror -Wfatal-error factinv.c
$ ./a.out 120
5
$ ./a.out -5040
7
$ ./a.out 0
-1
$ ./a.out 1
-1
$ ./a.out 3
2
$ ./a.out -25
4
```

Les quatre premiers tests sont satisfaisants mais les deux derniers révèlent une erreur. Pour analyser la

défaillance nous allons utiliser une méthode qui a fait ses preuves lorsqu'une simple relecture du code ne suffit pas à déterminer la cause. Nous allons *tracer* le comportement du programme en y insérant des affichages de variables pertinentes.

Si notre programme trouve un résultat sur ces deux tests, c'est qu'à une itération `div` vaut 1. Nous voyons dans le programme que `div` est obtenu à chaque itération par `div = div / i`. Affichons les valeurs de `div` et `i` avant la division et celle de `div` après la division.

```
int factinv (int x) {
    int i, div, v ;
    /* Compute v as |x|. */
    if (x < 0) v = -x ;
    else v = x ;
    /* Check error. */
    if ((v == 0) || (v == 1)) return -1 ;
    div = v ;
    for (i = 2; i <= v; i++) {
        printf ("Before div = %d\n", div);
        printf ("i = %d\n", i);
        div = div / i ;
        printf ("After div = %d\n", div);
        if (div == 1) return i ;
    }
    return -1 ;
}
```

```
$ ./a.out 3
Before div = 3
i = 2
After div = 1
2
```

On constate qu'après la division de 3 par 2 `div` vaut 1. Effectivement, le programme traitant des entiers, c'est la division euclidienne (entière) qui est utilisée. L'erreur commise dans notre modélisation est de ne pas avoir pris en compte le *type* des « nombres » manipulés. En effet, tout comme en mathématiques $\mathbb{Z} \neq \mathbb{R}$, en informatique `int` \neq `float`.

La cause de l'erreur étant déterminée il faut y remédier. Une première solution est d'utiliser des flottants à la place d'entiers dans le programme : on ne change pas l'algorithme, on change uniquement l'implantation. Cette solution est peu satisfaisante car la factorielle étant définie sur \mathbb{N} , faire intervenir \mathbb{R} n'est pas très élégant. De plus, l'utilisation des flottants est sujette aux erreurs d'arrondi et risquer d'ajouter des erreurs pour en corriger une n'est pas une solution encourageante.

Quand est-on certain que le calcul a réussi lors de la division ? La réponse à cette question est « quand la division tombe juste », donc quand le reste est nul. Si au cours d'une division le reste n'est pas nul, inutile de continuer à itérer : nous savons que le nombre initial n'était pas une factorielle. Une fois ce test fait, il devient sûr de regarder le quotient.

Il suffit alors de modifier légèrement notre algorithme pour insérer le calcul du reste de `div` divisé par `i` et la vérification de sa nullité avant de continuer.

```
...
for (i = 2; i <= v; i++) {
    int rem = div % i;
    if (rem != 0) return -1;
    div = div / i ;
    if (div == 1) return i ;
}
...
```

2.3 Conclusion

Concevoir un algorithme nécessite de la *rigueur* et l'explicitation de *détails*. Les types des données manipulées doivent être formellement définis. Toutes les étapes de calcul doivent être explicitées. Les limites

de la solution choisie doivent être clairement spécifiées. Une fois le périmètre de la solution déterminé, tous les cas possibles dans ce périmètre doivent être traités. Par exemple, dans un programme de mécanique on peut choisir de ne traiter que les solides indéformables, par contre il faudra gérer le cas où l'on rencontre un solide de masse nulle (indéformable n'implique pas de caractéristique de masse).

Lors de la phase d'implantation certains aspects techniques du langage cible peuvent apparaître. Par exemple en C, l'allocation dynamique explicite de mémoire et la libération peuvent être nécessaires alors que dans d'autres langages c'est un gestionnaire automatique qui s'en charge. De même l'utilisation d'arguments supplémentaires passés par adresse peuvent apparaître pour qu'une fonction « retourne » plusieurs résultats alors qu'un langage disposant de n-uplets natifs aurait évité ce détail technique.

Quoi qu'il en soit, l'implantation n'intervient qu'après que le problème initial ait été « suffisamment » décomposé pour devenir un ensemble de problèmes « suffisamment » simples à résoudre.

2.4 Domaines de entrées et des sorties

Revenons plus en détail sur l'importance de la détermination des domaines des entrées et des sorties. Considérons la spécification « Écrire un programme qui calcule la racine carrée d'un nombre donné en entrée ».

La racine carrée étant définie sur \mathbb{R}^+ , nous en déduisons que nous allons utiliser des `float`. Néanmoins, comme ces derniers sont forcément signés, nous ne pouvons pas restreindre le domaine informatique structurellement avec ce type. Il faudra donc gérer deux cas dans l'algorithme : < 0 et ≥ 0 . Une fois cette analyse effectuée, nous pouvons planter un algorithme à base de méthode itérative de Newton.

```
#define X0 (3.0) /* Random seed > 0. */

float squr (float a, int max_iter) {
    int i ;
    float xn = X0 ;

    if (a < 0) return -1 ; /* Assume -1 is the error value. */
    for (i = 0; i < max_iter; i++)
        xn = 0.5 * (xn + (a / xn)) ;

    return xn ;
}

int main (int argc, char *argv[]) {
    if (argc != 3) {
        printf ("Error. Wrong number of arguments.\n") ;
        return 1 ;
    }

    printf ("%f\n", squr (atof (argv[1]), atoi (argv[2]))) ;
    return 0 ;
}
```

Notons que nous avons pris soin de gérer le cas où le nombre d'arguments fournis dans le `main` est incorrect. En ce sens, nous gérons bien tous les cas dans le périmètre identifié de la solution.

La solution proposée s'appuie tacitement sur des hypothèses très fortes qui ont dirigé la forme de notre algorithme. En effet, dans la spécification initiale « Écrire un programme qui calcule la racine carrée d'un nombre donné en entrée », qui a dit que « nombre » était un réel ? De même, il n'est pas dit que le résultat doit être un réel.

Si l'on avait fait d'autres hypothèses, l'algorithme aurait été différent. Si l'on avait choisi que le résultat soit un entier alors il y aurait eu d'autres cas d'absence de racine ($\sqrt{1.44} = 1.2$ qui n'est pas représentable par un entier). De même si l'on avait décidé que le résultat devait être un complexe, il n'y aurait plus eu de cas d'erreur mais il aurait fallu implanter une arithmétique de complexes en plus.

Ainsi, il est impératif de définir *clairement* le domaine des entrées et des sorties puisque cela peut avoir un important impact sur la structure de l'algorithme. Nous allons distinguer trois types de domaines à déterminer.

2.4.1 Domaine mathématique

Le but est de déterminer la structure mathématique manipulée par l'algorithme. Ce peut être un entier, un entier positif ou nul, un réel, une suite finie ordonnée de lettres (mot, chaîne de caractères), une valeur de vérité (booléen, vrai / faux), un couple, un ensemble ordonné, etc.

Par exemple, une date sera représentée par trois entiers. Le premier représentant le jour sera compris entre 1 et 31 inclus. Le second, pour le mois sera compris entre 1 et 12 inclus. Quant au troisième pour l'année, il sera non contraint (on veut alors représenter des années avant JC). Un âge sera un entier naturel (strictement ?) positif. Une température en degrés Celcius sera un réel (par contre, un réel positif – ou nul ? – si c'est une température en Kelvin).

2.4.2 Domaine sémantique

Le domaine sémantique vient affiner le domaine mathématique en précisant la nature des objets manipulés et permet de définir les opérations autorisées sur les données.

Une valeur peut représenter une distance. Mais cette distance peut être de nature astronomique ou subatomique. Une valeur peut aussi représenter une vitesse. Cette vitesse peut être angulaire ou linéaire.

En fonction des domaines sémantiques des données, elles peuvent être combinées ou non. Par exemple, ajouter une vitesse et une distance n'a aucun sens. Par contre diviser une distance par une vitesse est licite et retourne une durée. Ajouter une vitesse linéaire et une vitesse angulaire n'a pas non plus de sens, quand bien même ce sont deux vitesses.

Le domaine sémantique peut également avoir un impact sur la manière d'interpréter des calculs. À titre d'exemple, nous avons vu en TD que l'on devait comparer l'égalité de flottants modulo un ϵ choisi (pour éviter les problèmes d'arrondi). Cet ϵ n'existe pas de manière absolue et dépend des ordres de grandeur. Ainsi, si l'on vérifie l'existence d'une collision à partir de la distance entre deux objets, si l'on prend $\epsilon = 10m$, dans le cas de distances astronomiques, deux étoiles à cette distance seront clairement entrées en collision. Par contre, à l'échelle de particules, ces dernières seront très loin d'une collision.

2.4.3 Domaine informatique

Ce domaine est le dernier raffinement, celui qui va définir par quel type informatique la donnée va être représentée (un flottant, un entier, une structure, un tableau, etc.).

Ce choix « technique » peut nous faire nous heurter à la sombre réalité qui est que les scalaires informatiques sont des approximations des nombres mathématiques. Cette réalité apporte différents problèmes dus à la finitude de ces nombres, aux erreurs d'arrondi etc. Les algorithmes peuvent devoir gérer explicitement ces problèmes. Par exemple, dans le programme de calcul de racine carrée de la section 2.4, les `float` étant toujours signés, il était impossible de restreindre structurellement le domaine d'entrée à \mathbb{R}^+ . L'algorithme a donc dû effectuer un test pour vérifier que son entrée était positive et émettre une erreur dans le cas contraire.

2.4.4 Conclusion

Afin de finement déterminer les domaines des entrées et des sorties, il faut donc déterminer les domaines mathématiques, puis sémantiques et enfin informatiques. Si les deux premiers domaines peuvent souvent être

déterminés dans un ordre indifférent, le domaine informatique ne peut être déduit qu'en dernier. C'est lui qui fournira la notion de type utilisée dans l'algorithme puis dans le programme (comme implantation de l'algorithme).

La détermination de ces domaines permet éventuellement de préciser la spécification et de révéler des incomplétudes dans cette dernière, souvent dues à des hypothèses tacites. Cela permet également de commencer à esquisser la structure de l'algorithme puisque cela va permettre d'identifier certains cas d'erreur.

Séance 3

Passer à l'échelle

3.1 Vers des programmes plus compliqués

Dans le chapitre précédent nous avons examiné comment modéliser un problème à résoudre, réfléchir aux domaines des entrées et des sorties, décomposer en sous-problèmes par raffinements successifs. Nous avons également vu l'importance d'un travail préalable sur papier, avec des dessins, des exemples. Et finalement, est arrivée la phase d'implantation de l'algorithme conçu, dans un langage de programmation afin d'obtenir un programme.

3.1.1 Structurer le code

Le travail d'implantation peut devenir conséquent lorsque le programme est compliqué. Il est alors primordial de structurer la rédaction du *code*. Tout comme nous avons raffiné l'algorithme par *étapes successives*, nous allons rédiger le programme par étapes successives. La quantité de code à écrire peut rapidement devenir importante. Il est alors nécessaire de découper le code en *unités de calcul séparées*.

(une algo)

$\text{foo}(a, b, c, d) =$
si a, b, c et $d < 0 \rightarrow$ erreur
sinon

$$\begin{cases} t_1 = \min(a, b) \\ t_2 = \max(c, d) \\ t_3 = \min(a, c) \\ t_4 = \max(b, d) \\ n = C_{t_1}^{t_2} + C_{t_3}^{t_4} + |a| \end{cases}$$

Si $n > 3$
 | \rightarrow "PLAUF"
sinon
 | \rightarrow "PLAF"

Considérons que la phase préliminaire de conception nous a amené à l'algorithme de l'image ci-contre. Une traduction sans tentative de réutilisation, de factorisation, d'organisation du code est typique d'un développement direct sans analyse assez poussée préalable. Il arrive parfois de trouver chez les débutants des programmes inutilement longs qui auraient pu être réduits de manière conséquente. Le gain de temps à la rédaction (et aux corrections d'erreurs de syntaxe que l'on commet forcément) compense largement le temps passé à structurer un programme.

Le programme suivant implante « correctement » l'algorithme. Du moins, son exécution retourne les résultats attendus. Néanmoins il est excessivement long et semble répéter plusieurs constructions identiques.

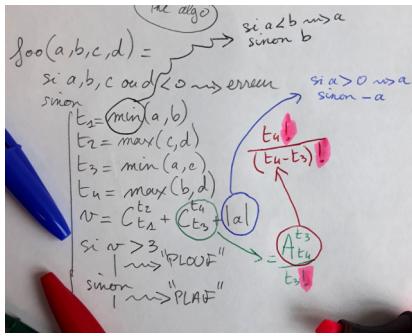
```
void foo (int a, int b, int c, int d) {  
    int i, t1, t2, t3, t4, fact_t1 ;  
    int fact_t1_t2, arr_t1_t2, abs_a ;  
    int comb_t1_t2, fact_t3, fact_t3_t4 ;  
    int arr_t3_t4, comb_t3_t4 ;  
    if ((a < 0) || (b < 0) || (c < 0) || (d < 0))  
        printf ("erreur") ;  
    if (a < b) t1 = a ;  
    else t1 = b ;  
    if (c > d) t2 = c ;  
    else t2 = d ;  
    if (a < c) t3 = a ;  
  
    fact_t1_t2 = 1 ;  
    for (i = 1; i <= t1 - t2; i++)  
        fact_t1_t2 = fact_t1_t2 * i ;  
    arr_t1_t2 = fact_t1_t2 / fact_t1_t2 ;  
    comb_t1_t2 = arr_t1_t2 / t2 ;  
    fact_t3 = 1 ;  
    for (i = 1; i <= t3; i++)  
        fact_t1 = fact_t3 * i ;  
    fact_t3_t4 = 1 ;  
    for (i = 1; i <= t3 - t4; i++)  
        fact_t3_t4 = fact_t3_t4 * i ;  
    arr_t3_t4 = fact_t3_t4 / fact_t3_t4 ;
```

```

else t3 = c ;
if (b > d) t4 = b ;
else t4 = d ;
fact_t1 = 1 ;
for (i = 1; i <= t1; i++)
    fact_t1 = fact_t1 * i ;
}

comb_t3_t4 = arr_t3_t4 / t4 ;
if (a < 0) abs_a = -a ;
else abs_a = a ;
if (comb_t1_t2 + comb_t3_t4 + abs_a > 3)
    printf ("PLOUF\n") ;
printf ("PLAF\n") ;
}

```



Une analyse de cet algorithme permet de se rendre compte que certains calculs sont décomposables en sous-calculs (de manière similaire à un problème décomposable en sous-problèmes). En particulier, C_n^k se calcule à partir de A_n^k . De la même manière, C_n^k et A_n^k se calculent à partir de $n!$. De plus, les calculs de minimum et maximum sont effectués plusieurs fois.

Il est alors indispensable de décomposer le programme en plusieurs *fonctions*. Le gain est important en terme de *lisibilité* et de *maintenabilité*. En effet, un programme plus concis se comprend plus aisément et y apporter des évolutions y est alors plus facile. De plus, lors de la rédaction, il y a moins d'opportunités d'introduire des erreurs.

Pour terminer, l'apparition de fonctions (comme pour $n!$, `min`, `max`, etc.) permet d'une part de nommer les calculs (lisibilité et maintenabilité accrues) et d'autre part de réutiliser du code sans le dupliquer. Ce code peut même être partagé entre différents programmes en le mettant dans une bibliothèque. Après ré-usinage, on obtient donc le programme suivant.

```

int fact (int n) {
    int res = 1 ;
    for (int i = 1; i <= n; i++)
        res = res * i ;
    return res ;
}

int argt (int n, int k) {
    return fact (n) / fact (n - k) ;
}

int comb (int n, int k) {
    return argt (n, k) / fact (k) ;
}

int min (int a, int b) {
    if (a < b) return a ;
    return b ;
}

int max (int a, int b) {
    if (a > b) return a ;
    return b ;
}

int abs (int x) {
    if (x < 0) return -x ;
    return x ;
}

void foo (a, b, c, d) {
    if ((a < 0) || (b < 0) || (c < 0) || (d < 0))
        printf ("erreur\n") ;
    int t1 = min (a, b) ;
    int t2 = max (c, d) ;
    int t3 = min (a, c) ;
    int t4 = max (b, d) ;
    if (comb (t1, t2) + comb (t3, t4) + abs (a) > 3)
        printf ("PLOUF") ;
    printf ("PLAF") ;
}

```

3.1.2 Rédiger petit-à-petit

Lorsque la quantité de code à rédiger est importante, il n'est pas judicieux d'écrire le programme au kilomètre et d'attendre d'avoir terminé pour compiler. En règle général on aboutit au résultat suivant :

```

$ gcc -c -Wall junk.c
junk.c:10:14: error: use of undeclared identifier 't2'
    if (c > d) t2 = c ;
junk.c:11:8: error: use of undeclared identifier 't2'
    else t2 = d ;
junk.c:12:18: error: expected expression
    if (a < c) t3 := a
junk.c:20:25: error: use of undeclared identifier 't2'
    for (i = 1; i <= t1 - t2; i++)
junk.c:27:3: error: use of undeclared identifier 'fact_t3_t4'; did you mean
      'fact_t1_t2'?

```

```
junk.c:4:31: note: 'fact_t1_t2' declared here
    int i, t1, t3, t4, fact_t1, fact_t1_t2, arr_t1_t2, abs_a ;
junk.c:29:5: error: use of undeclared identifier 'fact_t3_t4'; did you mean
    'fact_t1_t2'?
...
... 459 autres à venir...
```

qui sape le moral, et ce même lorsque l'on a l'habitude d'écrire du code depuis des années. En effet, l'oubli d'une déclaration, une typo dans un nom de variable, l'omission d'un point-virgule, toutes ces erreurs d'inattention arrivent chez tout le monde.

Faisons un détour par les mathématiques et examinons la manière dont on rédige une preuve. Soit à prouver $\forall f g x, x > 0 \wedge f(x) > 0 \wedge g(x) > 0 \Rightarrow f(2x) + g(2x) > 0$. Une démonstration est construite par étapes. On pose les hypothèses, on énonce un but que l'on prouve par des étapes elles-mêmes disposant d'hypothèses et de sous-but. Esquissons la démonstration de notre propriété.

1. Posons l'hypothèse $H_0 : x > 0$
2. Posons l'hypothèse $H_1 : f(x) > 0$
3. Posons l'hypothèse $H_2 : g(x) > 0$
4. Montrons que $f(2x) + g(2x) > 0$ par :
 - (a) une preuve que $f(2x) > 0$
 - (b) une preuve que $g(2x) > 0$
 - (c) CQFD par 4.a, 4.b et les propriétés de +

Nous savons que si nous arrivons à prouver $f(2x) > 0$ et $g(2x) > 0$ alors nous arrivons à prouver $f(2x) + g(2x) > 0$. Ces deux sous-but ne sont pas encore prouvés, mais ils permettent, s'ils sont effectivement prouvés, de garantir la cohérence globale du raisonnement. Il reste alors à appliquer la même méthode pour prouver chacun d'entre eux. La démonstration se poursuit donc en profondeur, de manière *incrémentale* avec une preuve de 4.a et une preuve de 4.b.

1. Posons l'hypothèse $H_0 : x > 0$
2. Posons l'hypothèse $H_1 : f(x) > 0$
3. Posons l'hypothèse $H_2 : g(x) > 0$
4. Montrons que $f(2x) + g(2x) > 0$ par :
 - (a) une preuve que $f(2x) > 0$
Montrons que $f(2x) > 0$ par :
 - i. une preuve que $2x > 0$
 - ii. CQFD par H_1 et 4.a.i
 - (b) une preuve que $g(2x) > 0$
Montrons que $g(2x) > 0$
 - i. ...
 - (c) CQFD par 4.a, 4.b et les propriétés de +

L'écriture d'un programme informatique peut procéder de la même façon, incrémentalement. Cela permet de vérifier la correction syntaxique du code au fur et à mesure, tant en cours d'écriture d'une longue fonction que d'un programme tout entier.

De même, une fois que le code écrit compile, il est possible de vérifier la correction d'exécution du code au fur et à mesure. On peut vérifier les fonctions déjà écrites voire celles partiellement écrites. Lorsqu'une fonction est manquante pour en tester une autre, on peut la remplacer par un « bouchon » (*stub* en anglais) qui est une fonction factice. Ce bouchon peut alors retourner une valeur constante ou une valeur aléatoire afin de simuler une implantation effective. Il « bouche le trou ».

3.2 Quelques défauts courants de conception

3.2.1 Affichage *versus* retour

Une erreur très répandue est la confusion entre affichage et retour du résultat d'une fonction. Considérons la spécification « Écrire une fonction qui calcule la factorielle de son argument » et le programme suivant censé y répondre.

```
int fact (int n) {
    int res = 1 ;
    for (int i = 1 ; i <= n; i++)
        res = res * i ;
    printf ("%d\n", res);
}
```

Le prototype de cette fonction indique qu'elle retourne un entier. Néanmoins, elle ne retourne rien et se contente d'afficher son résultat. Si nous tentons d'utiliser cette fonction en l'invoquant dans un `main`

```
int main () {
    printf ("%d\n", fact (5) * 2) ;
    return 0 ;
}
```

le compilateur génère un avertissement indiquant que la fonction se termine sans retourner de valeur alors qu'elle n'est pas définie comme `void`. Notons qu'en l'absence de l'option de compilation `-Wfatal-errors`, un exécutable est généré en dépit de l'avertissement.

```
$ gcc -Wall foo.c
foo.c:8:1: warning: control reaches end of non-void function [-Wreturn-type]
1 warning generated.
$ ./a.out
120
0
```

À l'exécution, l'affichage dans la fonction imprime bien le résultat, néanmoins la prétendue valeur de retour utilisée dans le `main` donne un résultat erroné. Ainsi cette fonction ne peut-être utilisée au sein de calculs plus compliqués puisqu'elle ne retourne pas de résultat.

Quand bien même elle afficherait *et* retournerait son résultat, elle ne serait pas très pratique à utiliser au sein de calculs compliqués. Imaginons que cette factorielle soit utilisée dans le programme de la section 3.1.1 qui calcule des combinaisons et des arrangements, nous verrions s'afficher de nombreux messages inutiles qui viendraient polluer le terminal, noyant les messages pertinents dans un flot de résultats de calculs intermédiaires.

L'affichage d'un résultat n'est pertinent que pour l'utilisateur qui attend un *résultat final*. Il peut également servir dans des scripts qui exploitent les messages envoyés sur la sortie standard, mais ces messages doivent avoir une valeur informative réelle, une valeur de résultat d'exécution.

Schématiquement, sauf spécification contraire, un *programme affiche* son résultat et une *fonction retourne* son résultat. Dans les langages munis d'un interprète en ligne de commande (comme OCAML ou PYTHON), il est possible d'appeler directement une fonction pour voir s'afficher la valeur qu'elle retourne. Ce n'est pas possible en C et il faudra donc écrire un `main` qui appellera la fonction et affichera la valeur renvoyée.

3.2.2 Duplication, imbrication, fouillis dans le code

Soient la spécification « Écrire une fonction qui affiche l'intersection de deux intervalles » et le programme y répondant suivant :

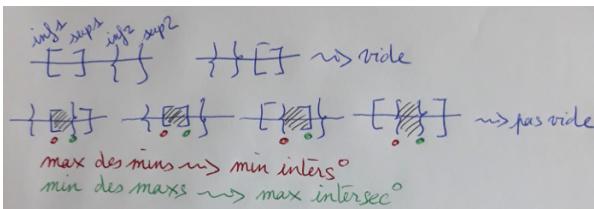
```

void inter_interv (int inf1, int sup1, int inf2, int sup2) {
    if (inf1 < inf2) {
        if (sup1 < inf2) { printf ("[]\n") ; return ; }
    }
    if (inf2 < inf1) {
        if (sup2 < inf1) { printf ("[]\n") ; return ; }
    }
    if (inf1 <= inf2) {
        if (sup1 >= inf2) {
            if (sup1 >= sup2) { printf ("[%d; %d]\n", inf2, sup2) ; return ; }
            else { printf ("[%d; %d]\n", inf2, sup1) ; return ; }
        }
    }
    if (inf2 <= inf1) {
        if (sup2 >= inf1) {
            if (sup2 >= sup1) { printf ("[%d; %d]\n", inf1, sup1) ; return ; }
            else { printf ("[%d; %d]\n", inf1, sup2) ; return ; }
        }
    }
}

```

Ce programme a réellement été écrit par un ou une élève des années précédentes, ce n'est donc pas une caricature. On y remarque des répétitions de structures similaires. L'imbrication des conditionnelles est importante. Cela a pour effet de rendre le programme difficilement compréhensible et difficilement maintenable. De plus, il est difficile de se convaincre que ces imbrications de tests couvrent bien tous les cas et qu'il n'y a pas de recouvrements (conditions testant plusieurs fois la même combinaison).

Une telle structure de code révèle une réflexion et une décomposition en sous-problèmes non satisfaisantes. Une vue plus synthétique du problème global est nécessaire.



Il en découle la nécessité de deux fonctions auxiliaires `min` et `max` pour simplifier fortement le code.

```

int min (int a, int b) { return ((a < b) ? a : b) ; }
int max (int a, int b) { return ((a > b) ? a : b) ; }

void inter_interv (int inf1, int sup1, int inf2, int sup2) {
    if ((sup1 < inf2) || (sup2 < inf1)) printf ("[]\n");
    else {
        int inf = max (inf1, inf2) ;
        int sup = min (sup1, sup2) ;
        printf ("[%d; %d]\n", inf, sup) ;
    }
}

```

Si l'on examine comment calculer les bornes inférieure et supérieure de l'intervalle intersection, on découvre six cas possibles. Les deux premiers mènent à l'intervalle vide. Les quatre suivants montrent que la borne inférieure de l'intersection est le *max* des bornes inférieures et la borne supérieure est le *min* des bornes supérieures.

3.3 Tester son programme

Il arrive « parfois » que le programme que l'on a écrit ne fonctionne pas. Ce n'est pas dramatique ni une preuve d'incompétence. Même les développeurs expérimentés introduisent des erreurs en programmant.

Ce qui est important est de détecter la présence d'erreurs et de les corriger, avant livraison au client. J'ai entendu un jour « un bon programmeur ne fait pas d'erreurs ». Dans ce cas, je pense qu'il n'existe aucun « bon » programmeur. Je prétendrai plutôt que « un bon programmeur ne fait pas *trop* d'erreurs et les corrige *efficacement* (vite et bien) ».

Un programme peut être erroné soit parce que l'algorithme est intrinsèquement incorrect, soit parce que l'implantation est incorrecte malgré un algorithme correct, soit pour les deux raisons. Pour s'en apercevoir il faut exécuter son programme avec des jeux de données pertinents. Il n'est pas question dans ce cours d'étudier le domaine du test logiciel (un cours existe en 2A voie Info). Il s'agit plutôt de donner quelques conseils informels pour guider l'intuition, pour tenter de trouver des jeux de test intéressants.

Il est possible d'effectuer des tests à deux niveaux de granularité. D'une part on peut tester le programme dans son ensemble : on parle alors de tests *haut-niveau*. Il est également possible de tester les fonctions individuellement : il s'agit alors de tests dits *bas-niveau*. De manière intermédiaire on peut tester la collaboration de plusieurs fonctions (on parle alors de tests *d'intégration*). Dans tous les cas la question qui se pose est de déterminer quelles valeurs des entrées utiliser. Une fois ces entrées déterminées il faudra déterminer les sorties attendues.

Une erreur importante est de déterminer ces valeurs et *les résultats attendus* à partir de la lecture du code source. Considérons la fonction suivante censée retourner le *min* de ses arguments :

```
int min (int x, int y) {
    if (x < y) return y ;
    return (x) ;
}
```

Dans son code nous pouvons lire qu'elle prend deux entrées. Il y figure une conditionnelle donc deux cas sont possibles. Si l'on choisit de tester avec $x = 5$ et $y = 6$, en accord avec le code, vu que $5 < 6$, la fonction doit retourner 6.

Bien entendu, si nous effectuons ce test et attendons 6 comme résultat nous serons satisfaits et convaincus du bon fonctionnement de la fonction puisque 6 sera bien retourné. Néanmoins, qu'avons-nous réellement vérifié ? Nous avons juste vérifié que le compilateur n'était pas buggé, qu'il avait bien traduit ce qui était écrit en un exécutable correct. Rien n'a été vérifié par rapport à la *spécification* de la fonction, à ce qui était initialement attendu de son comportement. Force est de constater qu'elle ne retourne pas le *min* de ses deux arguments.

Le point important est donc de toujours déterminer les *résultats attendus* à partir de la *spécification*. On doit donc déterminer des jeux d'entrées et vérifier que les sorties obtenues sont conformes à l'attendu et non au code. Nous allons distinguer deux catégories de tests.

D'une part les tests *fonctionnels*, effectués sans regarder le code source, doivent vérifier le bon comportement des cas nominaux (devant aboutir à un succès) et la gestion correcte des cas d'erreur. Il faut donc déterminer des entrées permettant d'aboutir à ces deux configurations. Il est particulièrement important de tester également des entrées aux limites des cas nominaux et d'erreurs. Bien souvent, des bugs se glissent à ces endroits (comme un $<$ à la place d'un \leq , un $+1$ ou -1 oublié dans l'utilisation d'un tableau ou de sa taille, etc.).

D'autre part, les tests *structurels*, effectués en regardant le code source visent à détecter les erreurs dues à une mauvaise structure du programme. Ils doivent (idéalement) permettre d'activer toutes branches du code (les deux branches pour d'une conditionnelle, 0 et n itérations pour une boucle, etc.). Ils doivent également s'assurer que des valeurs en dehors des domaines de définition d'opérations utilisées dans le code sont correctement gérées (par exemple s'il fait appel à une fonction de racine carrée).

3.3.1 Tests fonctionnels

Donnons quelques exemples de tests fonctionnels pour un programme dont la spécification est « trouver le plus petit des entiers dans un fichier texte dont le nom est passé en argument de la ligne de commande ».

Le code du programme à tester est le suivant, mais nous n'allons pas l'exploiter pour les jeux de test puisque nous parlons ici de tests *fonctionnels*.

```
minfile_broken.c

#include <stdio.h>

int main (int argc, char *argv[]) {
    int i, the_min ;
    FILE *file = fopen (argv[1], "rb") ;

    while (!feof (file)) {
        fscanf (file, "%d", &i) ;
        if (i < the_min) the_min = i ;
        fscanf (file, "%d", &i) ;
    }
    fclose (file) ;
    printf ("Min is %d\n", the_min) ;
    return 0 ;
}
```

Puisque le programme prend un argument en ligne de commande, il est naturel de vérifier son comportement dans le cas où l'on oublie de lui en fournir.

```
$ gcc -Wall minfile_broken.c
$ ./a.out
Segmentation fault: 11
```

Le résultat se solde par une interruption de l'exécution par le système d'exploitation, le programme essayant d'accéder à une zone mémoire interdite. On assiste ici à un manque de robustesse.

Nous pouvons également lui fournir en argument un nom de fichier inexistant ou non accessible.

```
$ ./a.out /tmp/doesntexist
Segmentation fault: 11
```

Le résultat est le même, pas forcément pour la même raison (le programme plante à un autre endroit) mais avec le même effet.

Après avoir tenté l'absence de fichier, essayons maintenant de lui fournir un fichier vide comme entrée.

```
$ ll /tmp/empty
-rw-r--r-- 1 didou wheel 0 12 mar 13:20 /tmp/empty
$ ./a.out /tmp/empty
Min is 0
```

Au lieu d'un message d'erreur informant l'utilisateur, le résultat obtenu est 0 qui n'est pas particulièrement correct. Après les cas un peu pathologiques (qui sont des cas d'erreur), nous pouvons envisager des cas nominaux (donc avec des fichiers bien formés).

```
$ more /tmp/data1.txt
3
5
67
34
56
8
-90
-45
45
$ ./a.out /tmp/data1.txt
Min is -90
```

Nous obtenons cette fois le résultat attendu, -90 étant bien le plus petit des entiers de ce fichier. Comme un seul test nominal n'est pas suffisant, testons avec un autre fichier.

```
$ more /tmp/data2.txt
3
-500
67
34
56
8
-90
-45
45
$ ./a.out /tmp/data2.txt
Min is -90
```

Le résultat est à nouveau incorrect, la réponse attendue étant -500. Alors certes, pour écrire ce test j'ai un peu « triché » car j'ai exploité la connaissance d'un bug que j'avais introduit dans le code (double lecture qui fait que l'on saute un entier sur deux). Néanmoins, la création de plusieurs jeux de test aurait vraisemblablement permis de détecter cette erreur même sans avoir « triché ». D'où l'intérêt de ne pas se contenter de un ou deux tests pour conclure au bon fonctionnement d'un programme.

Avec quatre tests sur cinq incorrects, il convient de corriger ce programme pour le rendre à la fois *correct* et *robuste*. Il faut donc vérifier la présence effective du bon nombre d'arguments sur la ligne de commande, vérifier l'ouverture du fichier, effectuer une lecture avant de tester l'atteinte de fin de fichier et enfin supprimer la lecture en double.

```
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i, the_min ;
    if (argc != 2) { /* Check the presence of the command line argument. */
        printf ("Error. Wrong number of arguments.\n") ;
        return 1 ;
    }
    FILE *file = fopen (argv[1], "rb") ;
    if (file == NULL) { /* Check if the file is opened. */
        printf ("Error. File not readable.\n") ;
        return 1 ;
    }
    if (fscanf (file, "%d", &i) == EOF) { /* Does it contain data ? */
        printf ("Error. Empty file.\n") ;
        fclose (file) ; /* Don't forget to close the opened file. */
        return 1 ;
    }
    while (!feof (file)) { /* Read until the end of file. */
        if (i < the_min) the_min = i ;
        fscanf (file, "%d", &i) ; /* Try to get the next data. */
    }
    fclose (file) ;
    printf ("Min is %d\n", the_min) ;
    return 0 ;
}
```

3.3.2 Tests fonctionnels *versus* structurels

Effectuer des tests fonctionnels avec toutes les valeurs possibles de toutes les entrées est irréalisable. Rien qu'une fonction prenant un entier (`int`) en argument nécessiterait 2^{32} tests. Les fichiers utilisés ci-dessus ont été générés aléatoirement (presque si l'on considère le dernier cas où j'ai exploité ma connaissance du code pour guider le test). Pour trouver la dernière erreur sans tricher, il aurait peut-être fallu générer un grand nombre de jeux de test. De plus, en fonction de l'implantation, des fonctions choisies, certaines erreurs peuvent se trouver latentes et n'être révélées que parce que l'on sait que ces fonctions sont utilisées. Il est donc nécessaire de faire cohabiter les deux types de tests, fonctionnels et structurels.

3.3.3 Tests structurels

Donnons quelques exemples de tests structurels pour un programme dont la spécification est « calculer le nombre de chiffres d'un nombre écrit en base 10 » et le code est le suivant.

```
int ndigits (int n) {
    return (1 + (int) log10 (n)) ;
}
```

En lisant le code source (on parle aussi de tests en *boîte blanche* car on a accès au code source, alors que les tests fonctionnels sont dits en *boîte noire*) on remarque l'utilisation de la fonction `log10`. L'idée de cet algorithme est la suivante :

- si n a p chiffres alors $10^{p-1} \leq n < 10^p$
- or $\log_{10}(10^{p-1}) = (p-1) \log_{10}(10) = p - 1$
- et $\log_{10}(10^p) = p \log_{10}(10) = p$
- donc $p - 1 \leq \log_{10}(n) < p$
- donc $p = \text{PartieEntiere}(\log_{10} n) + 1$

Sachant que la fonction `log10` n'est définie que sur \mathbb{R}^{*+} , il est naturel de tester une entrée négative, par exemple -100 pour obtenir une erreur.

```
$ ./a.out -100
-2147483647
```

La correction alors proposée consiste à dire que si l'entrée est négative, il faut prendre son opposé. En effet, deux entiers opposés s'écrivent avec le même nombre de chiffres (le signe n'étant typographiquement et mathématiquement pas un chiffre). On obtient alors le programme suivant.

```
int ndigits (int n) {
    if (n < 0) n = -n ;
    return (1 + (int) log10 (n)) ;
}
```

L'examen attentif révèle que cette correction est encore partielle. En effet, 0 est exclu du domaine de `log10`. Ainsi, tester avec une valeur positive ou négative non nulle fonctionne désormais, mais pas avec 0.

```
$ ./a.out 0
-2147483647
```

Pour rendre ce programme correct, il convient donc de gérer à part le cas du 0 qui s'écrit avec ... 1 chiffre. On obtient alors le programme suivant désormais exempt de bugs.

```
int ndigits (int n) {
    if (n == 0) return 1 ;
    if (n < 0) n = -n ;
    return (1 + (int) log10 (n)) ;
}
```

```
$ ./a.out -100
3
$ ./a.out 0
1
$ ./a.out 90
2
```

Séance 4

Complexité

Ce chapitre n'a pas pour vocation d'étudier la complexité formellement et en profondeur. Il a plutôt pour but une sensibilisation aux questions d'efficacité afin de guider la conception d'algorithmes privilégiant cette efficacité.

4.1 Comparer des programmes

Comme nous l'avons exposé (section 1.5), il n'y a jamais un seul algorithme permettant de répondre à un problème donné. On est alors tenté de comparer plusieurs solutions pour déterminer laquelle est la « meilleure ». Dans une comparaison il faut décider du *critère* utilisé. Examinons les deux programmes ci-dessous dont la finalité est de retourner la somme de ses deux arguments.

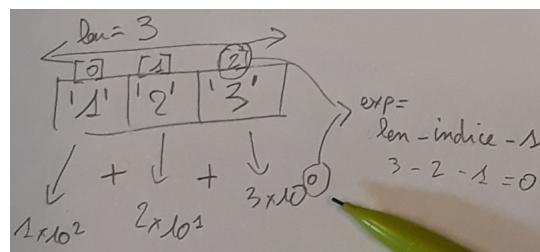
```
int addition (int x, int y) {  
    return (x + y) ;  
}
```

```
int addition (int x, int y) {  
    int res = x ;  
    for (int i = 0; i < y; i++)  
        res = res + 1 ;  
    return res ;  
}
```

On remarque que le programme de gauche est plus court que celui de droite. Pour autant, la longueur d'un code détermine-t-elle le nombre d'instructions à *effectivement exécuter*? La réponse est : pas forcément.

Ce qui est plus intéressant à comparer est ce nombre d'instructions car exécuter plus d'instructions nécessite plus de temps. Le programme de droite va devoir faire tourner sa boucle y fois, dans laquelle est effectuée une addition, alors que celui de gauche va invariablement exécuter une seule addition. Nous pouvons donc comparer l'efficacité *temporelle* de ces deux programmes et dire que celui de gauche est « meilleur » en *temps d'exécution*.

Si la comparaison précédente était triviale de par la différence de taille des programmes, ce n'est pas toujours le cas. Considérons le problème de convertir une chaîne de caractères en une valeur entière. Ainsi notre fonction appelée avec la chaîne "257" devra retourner l'entier 257.

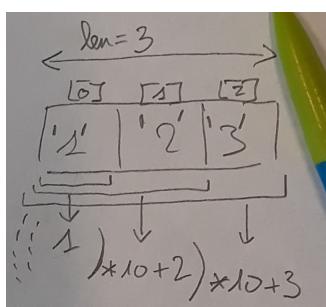


Si l'on fait un exemple avec la chaîne "123", on remarque que $123 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$. On en déduit qu'il suffit de parcourir la chaîne en additionnant chaque chiffre multiplié par la puissance de 10 convenable. Pour déterminer à quel exposant éléver 10 en fonction de l'indice i d'un caractère, on remarque qu'il décroît au fur et à mesure que l'on avance dans la chaîne pour terminer à 0 après avoir débuté à la longueur de la chaîne -1. On en déduit donc que l'exposant pour le caractère d'indice i est la longueur de la chaîne $-i - 1$. Il en découle un algorithme à base d'une simple boucle.

```
#include <math.h>

/* char = 8 bits integer whose value is the ASCII code of the character. */
unsigned int tonum (char *str) {
    unsigned int res = 0 ;
    int slen = strlen (str) ;
    for (int i = 0; i < slen; i++) ↗ ☺
        res = res + (str[i] - '0') * pow (10, (slen - i - 1)) ;
    return res ;
}
```

Dans ce programme figure un appel à la fonction `pow` pour effectuer l'élévation à la puissance. D'une part cette fonction travaille sur des flottants ce qui n'est pas très satisfaisant pour un problème qui traite de chaînes et d'entiers. D'autre part, cette fonction a un coût caché, que ce soit une naïve boucle ou une implantation plus subtile (nous aurions pu définir notre propre fonction `power` opérant sur des entiers, voire se créer une fonction `power10` ne calculant que des puissances de 10). De surcroît, lorsque nous calculons 10^5 , 10^4 , 10^3 etc. nous effectuons plusieurs fois les mêmes calculs (pour calculer 10^5 on calcule 10^4).



Une autre analyse du problème peut être menée en considérant que $123 = ((1) \times 10 + 2) \times 10 + 3$. On voit alors le poids de chaque chiffre se former au fur et à mesure que l'on progresse dans la chaîne de gauche à droite. Ainsi la fonction d'élévation à la puissance disparaît au profit de multiplications successives par 10. Il suffit alors de parcourir la chaîne du début à la fin, à chaque itération on multiplie le résultat de l'étape précédente par 10 et on y ajoute le chiffre courant. La valeur initiale avant de débuter le processus est alors 0. Nous obtenons encore un algorithme à base d'une simple boucle.

```
/* char = 8 bits integer whose value is the ASCII code of the character. */
unsigned int tonum (char *str) {
    unsigned int res = 0 ;
    int slen = strlen (str) ;
    for (int i = 0; i < slen; i++)
        res = res * 10 + (str[i] - '0') ;
    return res ;
}
```

Il est intéressant de remarquer que les deux algorithmes sont de tailles équivalentes. En première approche, tous deux se résument à une boucle qui parcourt la chaîne. La différence importante provient de la disparition des appels à la fonction `pow` dans le second, qui est donc intrinsèquement plus efficace.

On peut noter que nous pouvions même éviter de calculer la longueur de la chaîne dans le second algorithme (ce qui coûte une boucle tournant autant de fois que la chaîne comporte de caractères) en surveillant l'apparition du caractère '\0' qui clôture toute chaîne en C.

4.2 Complexité

La complexité d'un *algorithme* est la mesure de son *efficacité intrinsèque asymptotiquement* en fonction de la *taille* des données à traiter. Plusieurs mots sont importants dans cette définition.

Tout d'abord, on parle ici de complexité d'un algorithme, pas d'un programme. Un programme est en général constitué d'une multitude de fonctions implantant différents algorithmes. Estimer la complexité d'un logiciel de navigation Web, d'un jeu vidéo ou d'un traitement de texte est très compliqué de par la taille de ces applications et la variété des algorithmes mis en œuvre.

La notion de taille est primordiale : il faut être capable de mesurer les données d'entrée. Cette mesure peut être de différentes natures. Par exemple ce peut être le nombre de données à traiter (taille d'un tableau) ou la grandeur d'une valeur (si l'on écrit une fonction d'élévation à la puissance comme une simple boucle, la taille sera la valeur de l'exposant).

L'aspect asymptotique est également important car il indique que la complexité s'intéresse au comportement du programme lorsque la taille des données tend vers l'infini. La différence d'efficacité entre deux algorithmes pour traiter des petits problèmes, quand bien même ils n'auraient pas la même complexité peut rester non significative. Par exemple, la différence de temps d'exécution de nos deux algorithmes d'addition précédents serait indiscernable en pratique si l'on additionne 5 et 4 (coût de la boucle noyé dans le reste).

Pour terminer, l'efficacité mesurée est intrinsèque, donc indépendante de la vitesse de la machine qui exécutera l'algorithme. Que vous utilisiez un vieux Z80 ou un rutilant Core-i9, la complexité de l'algorithme restera la même. D'une certaine façon c'est assez moral car lorsque l'on considère un comportement asymptotique, que le processeur tourne 10, 20, 1000 fois plus vite, la durée d'exécution finira toujours par devenir significative.

Le complexité peut être exprimée dans le pire cas ou en moyenne. De même, elle peut être exprimée relativement au temps passé dans l'exécution : on parle alors de complexité *temporelle*. Mais aussi être exprimée en terme d'espace mémoire nécessaire à l'exécution : on parle alors de complexité *spatiale*.

On s'intéresse le plus souvent à la complexité temporelle car c'est celle qui est le plus facilement ressentie (on attend longtemps que l'ordinateur nous donne son résultat).

Pour calculer la complexité d'un algorithme sur une entrée de taille n , on compte le nombre d'opérations « de base » nécessaires et on regarde comment ce nombre évolue asymptotiquement. La notion d'opération « de base » peut être de différentes natures : on exprimera donc une complexité en nombre de *ces* opérations. On pense bien sûr aux instructions habituelles (addition, soustraction, etc.) mais ce peut être par exemple le nombre d'appels récursifs à une fonction comme nous le verrons en section 4.5.2.

4.3 Premier exemple

Considérons le programme suivant qui calcule la somme des n premiers entiers.

```
unsigned int sumint (unsigned int n) {
    unsigned int res = 0 ;
    for (unsigned int i = 1; i <= n; i++)
        res = res + i ;
    return res ;
}
```

La taille de l'entrée est la valeur de n . Plus n est grand, plus la boucle **while** va faire d'itérations. Ainsi, cette fonction va effectuer n additions + n incrémentations + n tests, ce qui donne un total de $3n$ instructions. La complexité est donc *linéaire* en la taille de l'entrée.

Notons que l'on ne s'intéresse en général qu'aux opérations calculant le résultat. Si nous avions eu un `printf` affichant le résultat, nous ne nous en serions pas préoccupé : il n'aurait pas contribué au résultat final. De surcroît son « poids » serait (relativement) indépendant de n (même si plus n est grand plus la somme obtenue sera grande, donc plus elle nécessitera de chiffres à l'affichage, mais on ne rentre pas dans ces niveaux de détails).

$$\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & n=5 \\
+ & + & + & + & + & + \\
n=5 & 4 & 3 & 2 & 1 & 0 \\
\hline
& \underbrace{n+n+n+n+n+n}_{n+1} & = & 2 * \overbrace{(1+2+\dots+5)}^{n+1} & = & n * (n+1)
\end{array}$$

Si l'on aborde le problème selon le raisonnement attribué à Gauss, on aboutit à une formule de calcul direct en fonction de n , bien connue :

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

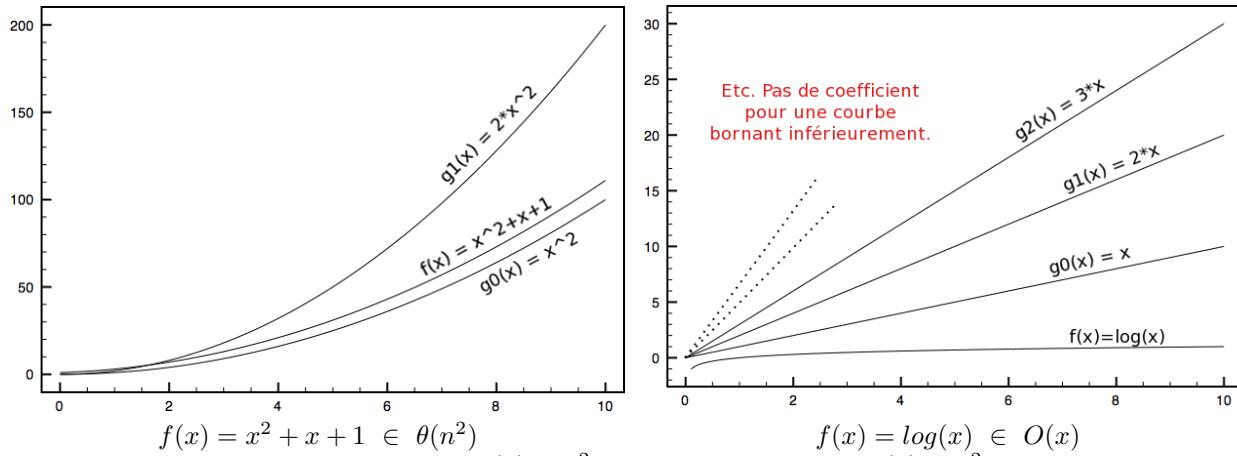
Ainsi il ne reste plus qu'à effectuer 1 addition + 1 multiplication + 1 division. Les tests de la boucle disparaissent ainsi que les incrémentations. Le calcul est désormais en temps constant : 3 opérations indépendamment de la valeur de n .

4.4 Notations de complexité

Pour évaluer la complexité d'un algorithme il faut encadrer le nombre d'opérations qu'il fait, donc déterminer une borne supérieure et une borne inférieure lorsque la taille de l'entrée tend vers $+\infty$. On positionne ensuite la complexité par rapport à une classe de fonctions mathématiques. On distingue deux notations différentes pour représenter les classes de fonctions :

- $O(g)$: l'ensemble des fonctions f telles que $0 \leq f(x) \leq k \times g(x)$
 - Avec $k \in \mathbb{R}_+^*$
 - Et $\exists x_0 \in \mathbb{N}^*, \forall x \geq x_0$ (\rightsquigarrow comportement asymptotique).
- $\theta(g)$: l'ensemble des fonctions f telles que $k_1 \times g(x) \leq f(x) \leq k_2 \times g(x)$
 - Avec $k_1, k_2 \in \mathbb{R}_+^*$
 - Et $\exists x_0 \in \mathbb{N}^*, \forall x \geq x_0$ (\rightsquigarrow comportement asymptotique).

Par convention (malheureuse) de notation, on écrit $f = O(g)$ au lieu de $f \in O(g)$. Lorsque l'on dit qu'un algorithme « est en $O(f)$ » on dit que sa fonction de complexité appartient à la classe $O(f)$. Ainsi, dire qu'un parcours de tableau « est en $O(n)$ » signifie que son coût évolue linéairement en fonction de la taille n des données (sous-entendu la taille du tableau).



Dans le schéma de gauche ci-dessus, $f(x) = x^2 + x + 1$ est bornée par $g(x) = x^2$ et g est bornée par f : elles sont mutuellement bornées. En effet, il existe (au moins) deux coefficients entiers par lesquels multiplier g pour « encadrer » f . *A contrario*, à droite, $f(x) = \log(x)$ est bornée par $g(x) = x$ sans que l'inverse ne soit vrai.

4.4.1 Classes de complexité

Il existe un certain nombre de classes de complexité couramment utilisées, avec un nom dédié pour certaines d'entre elles. Leur classement peut être résumé de la manière suivante :

$$\log(n) \ll \sqrt{n} \ll n \ll n \log(n) \ll n^2 \ll n^3 \ll 2^n \ll \exp(n) \ll n! \ll n^n \ll 2^{2^n}$$

- *Logarithmique* : $f(x) = O(\log(x)) \rightarrow$ réalisable
- *Linéaire* : $f(x) = O(x) \rightarrow$ réalisable
- *Quadratique* : $f(x) = O(x^2) \rightarrow$ réalisable
- *Polynomiale* : $\exists k > 0, f(x) = O(x^k) \rightarrow$ souvent réalisable
- *Exponentielle* : $\exists b > 1, f(x) = O(b^x) \rightarrow$ en général irréalisable
- *Doublement exponentielle*, par exemple : $f(x) = O(2^{2^x})$.
- *Sous-exponentielle*, par exemple : $f(x) = O(2^{\sqrt{x}})$.

Ainsi, l'algorithme de tri par tas d'un tableau est de complexité $O(n \log n)$ (pour n le nombre d'éléments du tableau), celui du calcul de la matrice d'accessibilité d'un graphe (Roy-Warshall) est en $O(n^3)$ (pour n le nombre de sommets du graphe).

Il est intéressant de se donner quelques ordres de grandeur pour se rendre compte de la limite d'utilisabilité d'un algorithme en fonction de sa complexité. Les microprocesseurs modernes exécutent de l'ordre de 10^{11} instructions par seconde :

(2010)	AMD FX-8150 (8-core) @ 3.6 GHz	$\approx 1.0 \cdot 10^{11}$ instr/s
(2011)	Intel Core i7 2600K @ 3.6 GHz	$\approx 1.2 \cdot 10^{11}$ instr/s
(2016)	Intel Core i7 6950X @ 3 GHz	$\approx 3.2 \cdot 10^{11}$ instr/s
(2017)	AMD Ryzen 7 1800X @ 3.6 GHz	$\approx 3 \cdot 10^{11}$ instr/s

Si sur un ordinateur standard 2^{40} (10^{12}) instructions ne posent pas de problème, lorsque l'on monte dans les ordres de grandeur, les choses se compliquent rapidement. Les records actuels sont un peu au-delà de 2^{60} (10^{18}) opérations binaires. Les calculs nécessitant ce genre de nombre d'instructions sont réalisés par des organismes disposant de moyens colossaux (par exemple la NSA ou le projet Folding@home) souvent distribués (plusieurs ordinateurs travaillant en parallèle).

On considère actuellement en cryptographie qu'une méthode de chiffrement nécessitant 2^{80} (10^{24}) opérations binaires pour casser un code est sûre, cette puissance de calcul étant inatteignable. Cela confère aux clefs de 128 bits de longueur une résistance pendant encore quelques dizaines d'années (sauf révolution majeure).

4.5 Complexité d'algorithmes pour Fibonacci

La suite de Fibonacci, décrivant initialement la croissance d'une population idéale de lapins (il y a quelques hypothèses dans cette modélisation), est définie de la manière suivante :

$$\begin{cases} Fib_0 &= 0 \\ Fib_1 &= 1 \\ Fib_n &= Fib_{n-1} + Fib_{n-2} \text{ pour } n > 1 \end{cases}$$

Nous allons examiner la complexité de cinq algorithmes permettant de calculer le n ième terme de cette suite qui débute par $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233\dots$

4.5.1 Calcul direct

La suite de Fibonacci est une relation de récurrence linéaire d'ordre 2 à coefficients constants. En effet, nous pouvons réécrire $Fib_n = Fib_{n-1} + Fib_{n-2}$ en $Fib_{n+2} = Fib_{n+1} + Fib_n$ qui a la forme $U_{n+2} = a U_{n+1} + b U_n$.

Des résultats d'algèbre linéaire montrent que si l'équation caractéristique $x^2 - a x - b = 0$ associée à cette suite possède deux racines distinctes r_1 et r_2 , alors tout terme U_n est égal à $\lambda r_1^n + \mu r_2^n$. Il reste alors à déterminer λ et μ à partir de U_0 et U_1 .

Dans le cas de Fibonacci nous avons $a = b = 1$ donc l'équation caractéristique est $x^2 - x - 1 = 0$. Les racines de ce polynôme sont $\varphi = \frac{1}{2}(1 + \sqrt{5})$ et $\bar{\varphi} = \frac{1}{2}(1 - \sqrt{5})$.

Donc $Fib_n = \lambda \varphi^n + \mu \bar{\varphi}^n$. Pour trouver les valeurs de λ et de μ il nous faut donc résoudre le système d'équations :

$$\begin{cases} Fib_0 = 0 \\ Fib_1 = 1 \end{cases} \quad \begin{cases} \lambda\varphi^0 + \mu\bar{\varphi}^0 = 0 \\ \lambda\varphi^1 + \mu\bar{\varphi}^1 = 1 \end{cases} \quad \begin{cases} \lambda + \mu = 0 \\ \lambda\varphi + \mu\bar{\varphi} = 1 \end{cases}$$

Par la première équation, on a $\mu = -\lambda$ que l'on substitue dans la seconde pour obtenir $\lambda\varphi - \lambda\bar{\varphi} = 1$. D'où

$$\begin{aligned} \lambda \frac{1+\sqrt{5}}{2} - \lambda \frac{1-\sqrt{5}}{2} &= 1 \\ \lambda \left(\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2} \right) &= 1 \\ \lambda \frac{1+\sqrt{5}-1+\sqrt{5}}{2} &= 1 \\ \lambda\sqrt{5} = 1 &\Rightarrow \lambda = \frac{1}{\sqrt{5}} \text{ et } \mu = -\frac{1}{\sqrt{5}} \end{aligned}$$

Ainsi, $Fib_n = \frac{1}{\sqrt{5}}\varphi^n - \frac{1}{\sqrt{5}}\bar{\varphi}^n = \frac{1}{\sqrt{5}}(\varphi^n - \bar{\varphi}^n)$. Si l'on calcule la valeur de $\bar{\varphi}$ on obtient environ -0.62. Donc pour $n > 1$ tendant vers ∞ (asymptotiquement), Fib_n est l'entier le plus proche de $\frac{\varphi^n}{\sqrt{5}}$.

Nous avons donc une méthode de calcul direct qui se traduit en une simple expression arithmétique. On note que cet algorithme nécessite l'utilisation de flottants et de diverses fonctions mathématiques (puissance, racine carrée, partie entière) dont le coût est caché et pas simple à déterminer. De plus, l'utilisation de flottants en double précision est indispensable afin que les erreurs d'arrondi ne provoquent pas de résultat faux trop rapidement. Toutefois, ces erreurs finiront toujours par se produire pour un n suffisamment grand (ex. `fib(71)` dans l'implantation ci-dessous sur une machine 64 bits).

```
#include <math.h>

unsigned long int fib (unsigned int n) {
    /* L'utilisation des float donne parfois un résultat erroné par arrondis.
       Ex : fib (71) -> 308061521170130 au lieu de 308061521170129. */
    long double phi = 1. / 2. * (1 + sqrt (5)) ;
    return (round (powl (phi, n) / sqrt (5))) ;
}
```

4.5.2 Version récursive naïve

Si l'on implante directement la définition de la suite de Fibonacci on obtient une fonction récursive simple avec deux appels récursifs :

```
unsigned int fib (unsigned int n) {
    if (n < 2) return (n) ;
    else return (fib (n - 1) + fib (n - 2)) ;
}
```

Calculons la complexité de cet algorithme en nombre d'appels à la fonction `fib`. On remarque que le calcul de `fib (n)` se réduit *in fine* à une somme de Fib_0 et de Fib_1 (cas d'arrêt de la récursion). En effet,

$$\begin{aligned} Fib_4 &= Fib_3 + Fib_2 \\ &= (Fib_2 + Fib_1) + (Fib_1 + Fib_0) \\ &= (Fib_1 + Fib_0) + Fib_1 + Fib_1 + Fib_0 \end{aligned}$$

Déterminons R_n le nombre d'appels récursifs pour calculer Fib_n . De par la structure de l'algorithme nous avons : $R_n = 1 + R_{n-1} + R_{n-2}$ et $R_0 = R_1 = 1$.
Posons $R'_n = 1 + R_n$. (*)

Nous avons alors $R'_n = 1 + \color{blue}{1} + \color{red}{R_{n-1}} + \color{red}{R_{n-2}} = \color{red}{R'_{n-1}} + \color{red}{R'_{n-2}}$ avec $R'_0 = R'_1 = 2$.

C'est une suite récurrente qui ressemble à Fib mais avec des conditions initiales différentes.

Souvenons-nous que Fib se décompose en sommes de Fib_0 et Fib_1 . Ici, R' va se décomposer en sommes de R'_0 et de R'_1 . Or on remarque que $R'_0 = 2 \times Fib_1$ et $R'_1 = 2 \times Fib_2$. Donc $R'_n = 2 \times Fib_{n+1}$.

Par (*), on a $R'_n = 2 \times Fib_{n+1} = 1 + R_n$.

D'où $R_n = 2 \times Fib_{n+1} - 1$.

Ainsi le nombre d'appels récursifs pour calculer Fib_n est $2 \times Fib_{n+1} - 1$. Comme nous avons vu dans la précédente section que $Fib_n = \frac{\varphi^n}{\sqrt{5}}$ nous en déduisons que $\theta(\text{fib}(n)) = \theta(\varphi^n)$, donc que la complexité en nombre d'appels est *exponentielle*. Il en découle que le calcul va rapidement devenir d'une durée insoutenable lorsque n va grandir.

4.5.3 Version itérative simple

Dans l'algorithme précédent, nous remarquons que nous calculons de multiples fois les valeurs de Fib_i . Le but désormais est d'éviter ce gaspillage de temps en utilisant un tableau pour les mémoriser dès qu'on les a calculé une fois. Cette technique dite de *mémoïsation* relève de la *programmation dynamique* que nous étudierons plus en détail en section 5.2.

Pour calculer Fib_n nous avons donc besoin d'un tableau de $n + 1$ cases (de 0 à n) dont les cases 0 et 1 sont initialisées avec 0 et 1. Ensuite, on calcule les valeurs de Fib_i et on les range dans le tableau. Pour calculer la valeur de Fib_i on utilisera les valeurs de Fib_{i-1} et Fib_{i-2} déjà mémorisées dans le tableau.

```
unsigned int fib (unsigned int n) {
    unsigned int i ;
    /* Allocation dynamique. */
    unsigned int *fibs = malloc (sizeof (unsigned int) * (n + 1)) ;
    fibs [0] = 0 ;
    fibs [1] = 1 ;
    for (i = 2; i < n + 1; i++) {
        fibs [i] = fibs [i - 1] + fibs [i - 2] ;
    }
    return (fibs [n]) ;
}
```

L'algorithme revêt alors la forme d'une simple boucle jusqu'à n : sa complexité *temporelle* est alors *linéaire* (en $\theta(n)$) puisque l'accès à une case de tableau se fait en temps constant. Néanmoins, la complexité *spatiale* est également en $\theta(n)$.

4.5.4 Version itérative plus rusée

Dans l'algorithme précédent on peut remarquer que l'on n'utilise toujours que les deux précédentes cases du tableau : $fibs[i - 1]$ et $fibs[i - 2]$. Il est donc inutile d'avoir un tableau, deux variables suffisent. On obtient alors un nouvel algorithme qui conserve la structure de boucle et met à jour fib1 et fib0 qui représentent Fib_{n-1} et Fib_{n-2} à chaque entrée de boucle. À chaque sortie de boucle fib1 représente alors Fib_n .

$$\begin{aligned}\text{fib1}' &\leftarrow \text{fib0} + \text{fib1} \\ &= Fib_{n-2} + Fib_{n-1} \\ \text{fib0}' &\leftarrow \text{fib1}' - \text{fib0} \\ &= Fib_{n-2} + Fib_{n-1} - Fib_{n-2} = Fib_{n-1}\end{aligned}$$

```
unsigned int fib (unsigned int n) {
    unsigned int i ;
    unsigned int fib0 = 0 ;
```

```

unsigned int fib1 = 1 ;
if (n < 2) return (n) ;
for (i = 2; i < n + 1; i++) {
    fib1 = fib0 + fib1 ;
    fib0 = fib1 - fib0 ;
}
return (fib1) ;
}

```

La boucle n'ayant pas changé, la complexité temporelle reste linéaire mais la complexité spatiale est devenue constante, d'où un gain notable par rapport à l'algorithme précédent.

4.5.5 Version itérative optimale

Si pour $n \geq 2$, l'on écrit Fib_n et Fib_{n-1} sous la forme :

$$\begin{aligned} Fib_n &= 1 \times Fib_{n-1} + 1 \times Fib_{n-2} \\ Fib_{n-1} &= 1 \times Fib_{n-1} + 0 \times Fib_{n-2} \end{aligned}$$

on peut utiliser une notation matricielle et on a donc :

$$\begin{aligned} \begin{pmatrix} Fib_n \\ Fib_{n-1} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \times \begin{pmatrix} Fib_{n-1} \\ Fib_{n-2} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \times \begin{pmatrix} Fib_1 \\ Fib_0 \end{pmatrix} \end{aligned}$$

Le problème se réduit donc à une exponentiation de matrice pour laquelle un algorithme efficace existe, avec une complexité temporelle en $\theta(\log(n))$ (exponentiation rapide aussi appelée *square-and-multiply* en anglais). Quant à la complexité spatiale, elle est constante. Nous obtenons donc un algorithme très efficace avec cette dernière modélisation du problème, n'utilisant pas les flottants donc évitant des problèmes d'arrondi.

4.5.6 Comparaison des temps de calcul

Chaque algorithme a été implanté en C et les programmes obtenus ont été exécutés sur une machine d'il y a une dizaine d'années, disposant de 4 Go de RAM. Les machines récentes permettent de calculer avec des valeur de n plus grandes pour chaque algorithme, néanmoins pour des n suffisamment grands, on retrouve la même évolution finale (comportement asymptotique).

	n	40	2^{25}	2^{28}	2^{31}
fib01	$\theta(\varphi^n)$	31s		Calcul irréalisable	
fib02	$\theta(n)$	< 1s	18s	Segmentation fault	
fib03	$\theta(n)$	< 1s	4s	25s	195s
fib04	$\theta(\log(n))$	< 1s	< 1s	< 1s	< 1s

Le premier algorithme devient rapidement inutilisable. Sur une machine moderne (Intel Core i7 à 2,5 GHz), Fib_{50} prend une minute et demie. Quant à Fib_{55} , la limite à la patience pousse à interrompre son exécution au bout de plusieurs minutes...

Le second algorithme ne pouvait pas s'exécuter à cause de la quantité de mémoire requise sur la machine de l'époque. Les machines récentes ayant au moins 16 Go de RAM, bien évidemment cet algorithme peut désormais supporter des valeurs de n nettement plus grandes.

4.6 Conclusion

Il existe de nombreux travaux en théorie de la complexité. Cette théorie vise à classifier les problèmes en fonction de la complexité du meilleur algorithme permettant de les résoudre. Lorsque l'on parle de « meilleur » algorithme, on ne sous-entend pas le meilleur « connu ». Effet, pour certains problèmes il est possible de déterminer la complexité optimale d'un algorithme résolvant ces problèmes.

C'est le cas d'un tri par comparaison qui ne peut avoir une complexité inférieure à $\theta(n \log(n))$. Il existe de nombreux algorithmes de tri dont certains ont une complexité supérieure, mais aucun une complexité inférieure (dans le cas de tableaux arbitraires sans hypothèses sur leur contenu).

Pour d'autres problèmes au contraire, on ne sait pas prédire la complexité optimale. Certains problèmes sont caractérisés comme intrinsèquement *durs* : on sait qu'il n'existe pas d'algorithmes efficaces pour les résoudre. Au contraire, d'autres sont *faciles* et mènent à des algorithmes efficaces, sans pour autant que l'on ait forcément trouvé à ce jour *le* plus efficace.

Cette étude de la complexité semble dire qu'il est inutile d'essayer de gagner quelques instructions dans une boucle : sa complexité restera linéaire et asymptotiquement elle tendra vers la même limite. En effet, les notation θ et O font disparaître les coefficients k, k_1, k_2 qu'elles impliquent pourtant.

Donc un algorithme qui traitera une donnée de taille n en $10 \times n$ instructions sera de même classe de complexité qu'un autre nécessitant $100000 \times n$ instructions. Asymptotiquement, à l'infini, ils se comporteront pareil, certes, mais pour des données de tailles plus petites, il n'empêchera que le premier s'exécutera 10000 plus vite que le second, ce qui pourra faire une réelle différence.

Donc non, la notion de complexité ne rend pas inutile la chasse aux instructions inutiles, aux optimisations des programmes. Il n'est pas raisonnable de se cacher derrière cette notion pour se soustraire à l'écriture d'algorithmes optimisés même si cela ne change pas leur classe de complexité. Parcourir deux fois un tableau pour trouver l'élément minimal et l'élément maximal sera plus long que le parcourir une fois en déterminant les deux éléments en même temps.

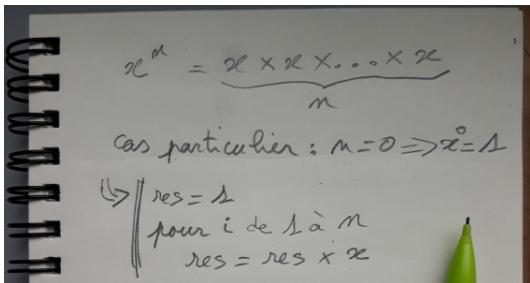
Séance 5

Différents paradigmes de programmation

Dans ce chapitre nous allons examiner trois structures caractéristiques d'algorithmes : « diviser pour régner », la programmation dynamique et enfin la programmation gloutonne. Ces structures générales algorithmiques s'appliquent à différents problèmes. Elles peuvent permettre d'obtenir des algorithmes efficaces ou approchés pour résoudre des problèmes difficiles.

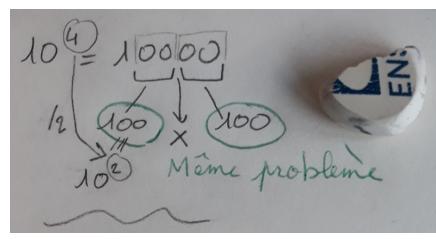
5.1 Diviser pour régner

Considérons le problème de calculer x^n . La première question à se poser est « qu'est-ce que j'ai, qu'est-ce que je veux ? ». Autrement dit, déterminer les domaines de x , n et du résultat. Nous choisissons ici de prendre \mathbb{N} pour tous, et cherchons donc à résoudre le problème de la puissance entière.



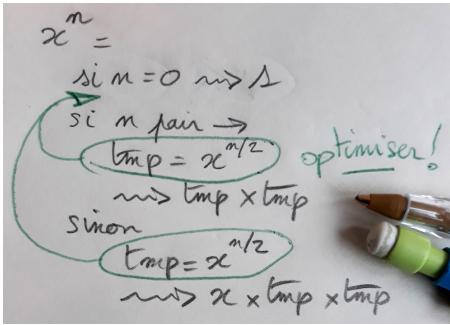
Nous pouvons maintenant nous intéresser à la question du « comment ». La première idée est de considérer que $x^n = x \times x \times \dots \times x$ répété n fois. Il reste le cas où $n = 0$ qui doit retourner 1. Nous obtenons alors un algorithme composé d'une simple boucle dont la complexité temporelle est linéaire ($O(n)$) et la complexité spatiale est constante ($O(1)$). Le cas $n = 0$ est géré dans le cas général par l'initialisation du résultat à 1.

```
int power (int x, int n) {
    int res = 1 ;
    for (int i = 1; i <= n; i++)
        res = res * x ;
    return res ;
}
```



En reprenant le problème sur un exemple, on remarque que (par exemple) 10^4 est égal à 10000. C'est donc le produit de deux termes identiques, 100×100 qui correspondent à 10^2 . Dans ce produit on a donc deux fois des sous-problèmes de *même forme* (ici identiques) à résoudre. Et l'on remarque que l'exposant de ces sous-problèmes est l'exposant initial divisé par 2. Nous sommes ici en présence d'un cas particulier de $x^{a+b} = x^a \times x^b$ avec l'idée de prendre $a = b$ pour ne calculer *qu'une fois* la puissance, donc prendre $a = \frac{n}{2}$. Il reste le

cas embarrassant où n est impair, par exemple dans le calcul de 10^5 . En fait, pas si embarrassant que ça : $10^5 = 10 \times 10^4$, et 10^4 on sait le faire efficacement.



Il en découle un algorithme récursif à trois cas : le cas d'arrêt lorsque $n = 0$, le cas où n est pair et celui où il est impair. Si n est pair, on calcule $x^{n/2}$ que l'on élève ensuite au carré. S'il est impair, on fait de même mais on multiplie le résultat par x . On s'aperçoit que dans les deux cas on calcule $x^{n/2}$: autant ne le faire qu'à un seul endroit, on réduira la taille du code.

```

int power (int x, int n) {
    if (n == 0) return 1 ;
    int tmp = power (x, n / 2) ;
    if (n % 2 == 0)
        return (tmp * tmp) ;
    else return (x * tmp * tmp) ;
}

```

On obtient alors un algorithme nettement plus efficace avec une complexité temporelle en $\theta(\log(n))$. La complexité spatiale a néanmoins perdu même si ce détail est caché. En effet, les appels récursifs nécessitent de la pile alors que la boucle précédente était en mémoire constante. Pour autant, la complexité spatiale sera également logarithmique ce qui n'est pas désastreux. Notons que cet algorithme fonctionne car n est un entier. En effet, s'il avait été un flottant, la division par 2 n'eût jamais fini par donner 0 (en effet, $1/2 = 0.5$) et la boucle eût tourné indéfiniment.

Ainsi, le paradigme « diviser pour régner » consiste à partitionner un problème initial en sous-problèmes de *mêmes formes* (possiblement identiques) mais plus petits. On résout ensuite les sous-problèmes (récur-sivement) puis on recombine les solutions des sous-problèmes pour résoudre le problème initial. On explore ainsi toutes les solutions possibles.

5.2 Programmation dynamique

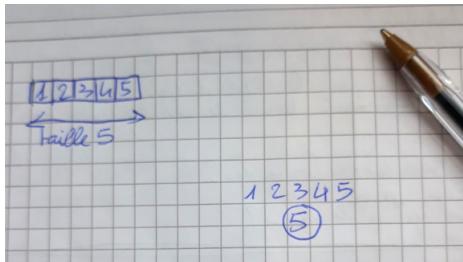
Considérons le problème consistant à trouver une découpe qui permette de maximiser le gain obtenu par la vente des morceaux d'une barre. Nous disposons d'une barre (de n'importe quoi) de longue fixe. En fonction de sa longueur, chaque morceau est vendu à un prix donné :

Longueur	0	1	2	3	4	5	6	7	8
Prix	0	2	3	8	10	13	15	16	21

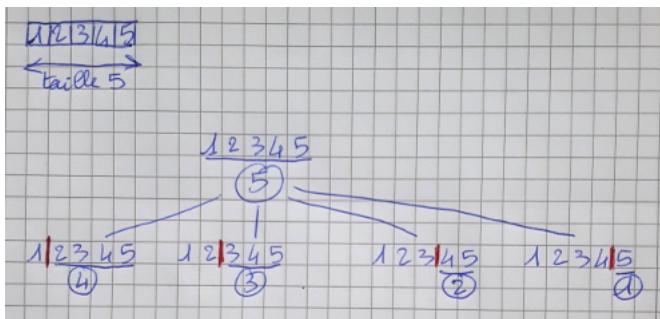
Avec ces prix, une solution optimale est de découper la barre en un bout de longueur 3 et un de longueur 5 : le prix obtenu est 21. Néanmoins, une autre solution optimale est un unique bout de longueur 8.

Un algorithme pour déterminer la découpe optimale consiste à déterminer toutes les découpes possibles, calculer le gain obtenu pour chacune et retourner le plus grand.

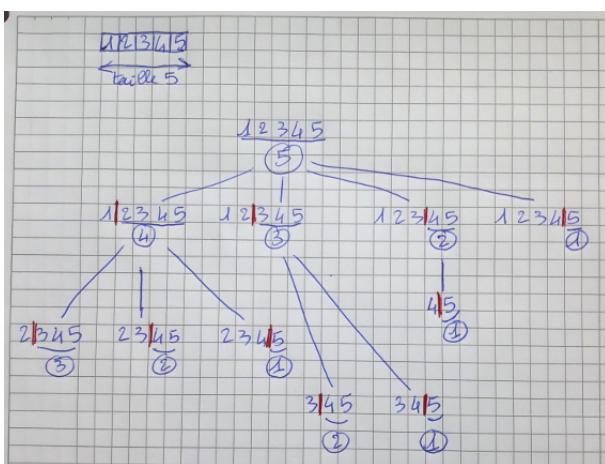
Il se pose désormais la question de calculer *toutes* les découpes possibles. Pour y parvenir, il faut commencer par envisager le cas de la barre non coupée (barre entière). Ensuite, on examine tous les cas avec *une seule* découpe effectuée à toutes les positions possibles de la barre. Et pour chacun, on recommence le même traitement sur ce qui reste de la barre après la découpe. Nous obtenons donc naturellement un algorithme récursif que nous allons illustrer par des schémas.



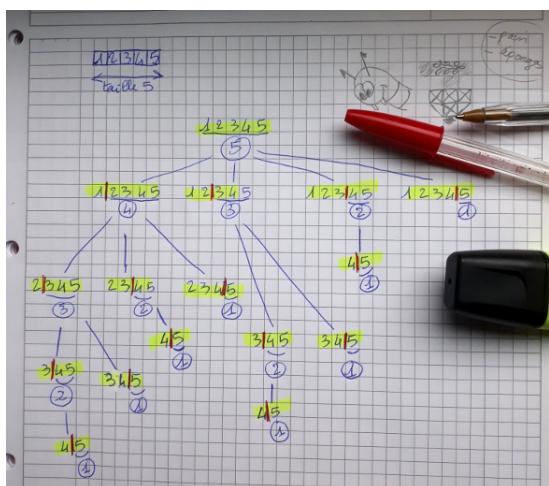
Nous représentons la barre avec les positions où couper par des petits carrés. Lorsque l'on examine le cas sans découpe, on obtient *un seul* morceau de la taille de la barre d'origine (5).



Ensuite, il faut envisager tous les cas où il est possible de couper la barre. Si sa longueur est n , il y a $n-1$ découpes possibles. De gauche à droite, nous pouvons couper après le bloc 1 et il reste une barre de taille 4, après le bloc 2 et il reste une barre de taille 3, après le bloc 3, etc. La prochaine étape est donc de réitérer ce *même* raisonnement sur *chacune* des quatre barres restantes obtenues.



Le processus continue alors récursivement sur chacune des nouvelles barres obtenues. Après avoir effectué deux découpes, nous obtenons l'arbre de possibilités suivant.



Après avoir récursivement appliqué ce processus jusqu'au bout, on obtient donc les possibilités suivantes dans lesquelles on remarque que le cas où l'on ne coupe pas la barre est la *partie restante* du cas père d'une suite de découpes. Par exemple, dans la barre 12|345, le cas « pas de découpe » dans 345 est père des deux enfants 3|45 et 34|5.

La longueur de la barre initiale étant n , nous avons 2^{n-1} possibilités. Le nombre de cas, donc la complexité sont donc *exponentiels*, ce qui est de mauvais augure pour l'efficacité de l'algorithme.

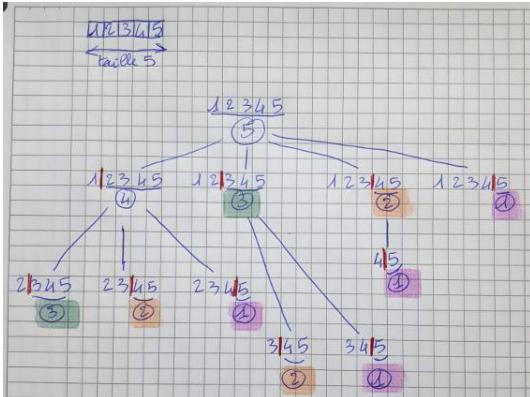
Il nous faut maintenant déterminer à chaque étape quel est le meilleur gain. Il s'agit donc du *max*, pour chaque découpe, du prix de la partie coupée (partie gauche) plus *le meilleur gain* que l'on peut obtenir dans la partie restante. On retrouve ici la formulation récursive qui consiste à calculer le *meilleur gain* à une étape à partie du *meilleur gain* des sous-problèmes.

$$\text{meilleur gain} = \max \left\{ \begin{array}{l} \text{Pour chaque découpe, (taille barre après découpe } \in [1; n]) \\ \text{prix de la partie coupée +} \\ \text{meilleur gain de ce que l'on peut tirer du reste de la découpe} \end{array} \right.$$

Nous en déduisons alors l'implantation suivante qui suit exactement la formulation récursive présentée.

```
#define PSIZE (9)
/* Size = 0 => price = 0 => prices[0] = 0 */
int prices[PSIZE] = { 0, 2, 3, 8, 10, 13, 15, 16, 21 } ;

int cut (int size) {
    if (size <= 0) return 0 ;
    int best = 0 ; /* Best price, recursively. */
    int cut_size = 1 ; /* Size of the removed part. */
    while (cut_size <= size) { /* <= size allows no cut. */
        best = max (best, prices[cut_size] + cut (size - cut_size)) ;
        cut_size++ ;
    }
    return best ;
}
```



Revenons sur l'efficacité de cet algorithme. Rien que lorsque l'on considère une profondeur de deux découpes, nous voyons que le problème de taille 3 est calculé 2 fois, celui de taille 2 est calculé 3 fois, celui de taille 1 est calculé 4 fois. Comme nous l'avons remarqué, cet algorithme est exponentiel, mais parce que l'on fait et refait sans cesse le même travail ! Il faut donc éviter cette répétition des mêmes calculs.

L'idée est donc de mémoriser les calculs déjà faits pour les occurrences futures. Pour cela, nous avons besoin d'un tableau de taille n . Avant de calculer la solution d'un sous-problème, on va regarder si son résultat n'a pas déjà été mémorisé dans le tableau. Si

Cette technique est la même que celle utilisée pour améliorer le calcul du n ième terme de la suite de Fibonacci dans l'algorithme de la section 4.5.3. Nous obtenons donc l'algorithme suivant de complexité $O(n^2)$, dans lequel les parties modifiées par rapport à l'algorithme précédent sont mises en relief.

```
#define PSIZE (9)
int prices[PSIZE] = { 0, 2, 3, 8, 10, 13, 15, 16, 21 } ;
# Initially, no intermediate prices are recorded.
int memo_prices[PSIZE] = { -1, -1, -1, -1, -1, -1, -1, -1, -1 };

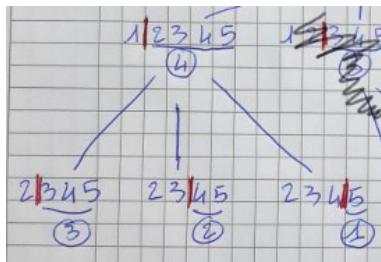
int cut (int size) {
    if (size <= 0) return 0 ;
    if (memo_prices[size] >= 0) return memo_prices[size];
    int best = 0 ; /* Best price, recursively. */
    int cut_size = 1 ; /* Size of the removed part. */
```

```

while (cut_size <= size) { /* <= size allows no cut. */
    best = max (best, prices[cut_size] + cut (size - cut_size));
    cut_size++;
}
memo_prices[size] = best;
return best ;
}

```

L'algorithme obtenu reste récursif. Il nécessite donc l'utilisation de la pile pour chaque appel récursif, ainsi sa complexité spatiale n'est pas constante. Est-il possible de faire mieux ? Si l'on examine un exemple, celui du calcul pour une barre de taille 4, on remarque que la solution de ce problème se calcule par :



$$\begin{aligned} \text{memo_prices}[4] &= \\ \max \left\{ \begin{array}{l} \text{prices}[1] + \text{memo_prices}[3] \\ \text{prices}[2] + \text{memo_prices}[2] \\ \text{prices}[3] + \text{memo_prices}[1] \\ \text{prices}[4] + \text{memo_prices}[0] \end{array} \right. \end{aligned}$$

Autrement dit, pour une barre de taille n , nous avons besoin des valeurs mémorisées pour les sous-problèmes de taille $n - 1$ à 0 : $\text{memo_prices}[i] = \max_{j=1 \text{ à } i} \{ \text{prices}[j] + \text{memo_prices}[i - j] \}$. Donc si nous remplissons le tableau `memo_prices` de 0 à n , nous n'avons plus besoin d'effectuer d'appels récursifs, une simple boucle de 0 à n (inclus) suffit.

```

int prices [PSIZE] = { 0, 2, 3, 8, 10, 13, 15, 16, 21 } ;
int memo_prices[PSIZE] = { -1, -1, -1, -1, -1, -1, -1, -1, -1 } ;

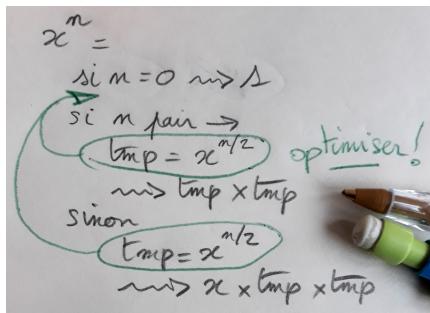
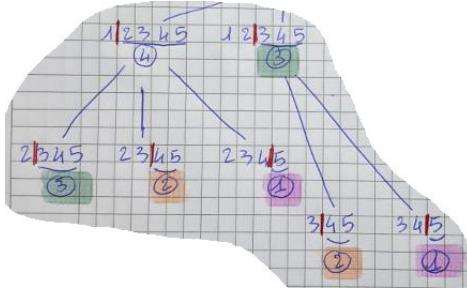
int cut (int size) {
    for (int i = 0 ; i <= size; i++) {
        int best = 0 ;
        for (int j = 1; j <= i; j++)
            best = max (best, prices[j] + memo_prices[i - j]) ;
        memo_prices[i] = best ;
    }
    return memo_prices[size] ;
}

```

Ainsi la complexité spatiale devient constante ($O(1)$) et même si nous avons maintenant 2 *boucles imbriquées*, la complexité temporelle reste $O(n^2)$.

Ainsi, le paradigme de la programmation dynamique consiste à décomposer un problème initial en sous-problèmes de la *même forme* mais plus petits (jusque là, comme pour « diviser pour régner »), mais à également mémoriser les résultats *intermédiaires*. On explore encore *toutes* les configurations possibles mais l'on évite de refaire plusieurs fois les mêmes calculs grâce à la *mémoïsation*.

Un autre des intérêts de ce paradigme est sa capacité à fonctionner sur des sous-problèmes n'étant pas *indépendants*. Par exemple, les calculs des solutions des barres de taille 3 et 4 ne sont pas indépendants puisque dans les deux cas il faut connaître le résultat pour la taille 2.



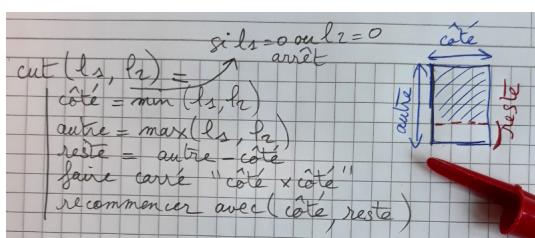
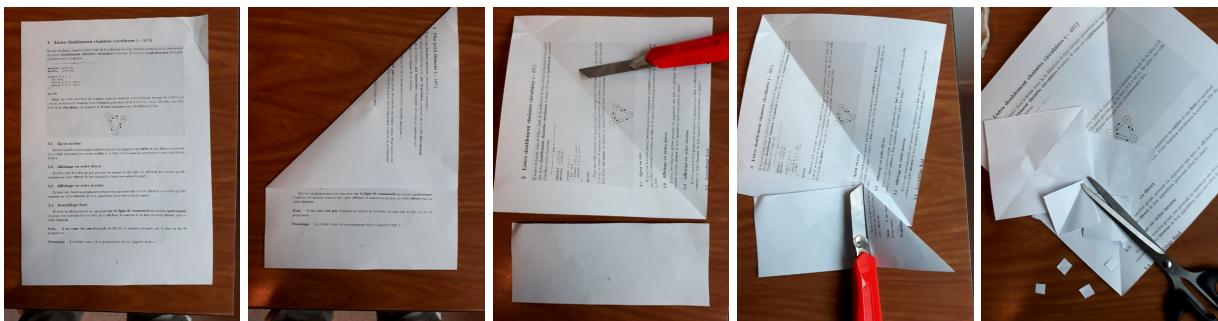
Revenons quelques instants sur l'algorithme du calcul de x^n présenté dans la section 5.1. Nous avions remarqué qu'il était possible de calculer $x^{n/2}$ puis de retourner cette valeur au carré. De fait, nous avons évité de recalculer plusieurs fois cette puissance. Le sous-problème que nous avons résolu pouvait être vu comme deux sous-problèmes tellement dépendants qu'ils étaient identiques. Cet algorithme de type « diviser pour régner » rentre donc également dans le cadre de la programmation dynamique de manière triviale. La mémorisation du résultat intermédiaire a été faite dans une simple variable, un seul résultat a été mémorisé, mais il y a bien eu mémoïsation, même « triviale ».

5.3 Programmation gloutonne

Soit une feuille de dimensions $l_1 \times l_2$ que nous considérons entières (i.e. $\in \mathbb{N}$). Nous souhaitons découper cette feuille en carrés en minimisant le nombre de carrés formés.

Une première solution pourrait être d'explorer toutes les découpes possibles. On se rend rapidement compte que c'est illusoire tant le nombre de possibilités est gigantesque : la complexité serait déraisonnable.

Une autre solution repose sur l'idée de découper un carré le plus grand possible à chaque fois, avec l'espoir de minimiser le nombre de carrés créés (une plus grande surface retirée laissera moins de papier à découper aux prochaines itérations). Ainsi, à chaque étape cette solution choisit une solution *localement optimale*.



Sur la base de cette réflexion, nous pouvons en déduire un algorithme simple dans lequel le côté du carré à découper est la plus petite dimension de la feuille. Nous calculons alors les dimensions du rectangle restant et nous appliquons le même processus sur ce dernier. L'algorithme est alors encore récursif.

```

void cut (int dim1, int dim2) {
    if ((dim1 != 0) && (dim2 != 0)) {
        int sq_size, rem ;
        if (dim1 < dim2) {
            sq_size = dim1 ; rem = dim2 - dim1 ;
        }
        else {
            sq_size = dim2 ; rem = dim1 - dim2 ;
        }
        printf ("Make %d x %d.\n", sq_size, sq_size) ;
        cut (sq_size, rem) ;
    }
}

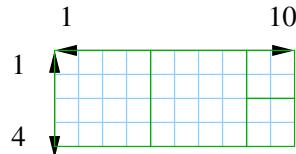
```

Exécutons le programme obtenu et examinons les résultats pour s'assurer de l'optimalité de l'algorithme. Si nous prenons une feuille de 10×4 , nous obtenons les découpages suivants : deux carrés de 4×4 et deux carrés de 2×2 . Cette solution est effectivement *optimale*.

```

./cutsquare 10 4
Make 4 x 4
Make 4 x 4
Make 2 x 2
Make 2 x 2

```

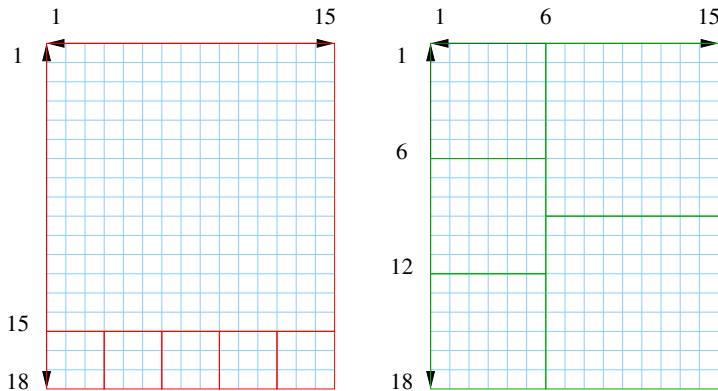


Testons maintenant la découpe d'un carré de 15×18 . Nous obtenons un carré de 15×15 et cinq carrés de 3×3 . Cependant, une découpe produisant deux carrés de 9×9 et trois carrés de 6×6 est meilleure car elle produit un carré de moins que la solution trouvée par l'algorithme. Ainsi, ce dernier n'a pas fourni de solution optimale sur ce problème.

```

./cutsquare 15 18
Make 15 x 15
Make 3 x 3
Make 3 x 3
Make 3 x 3
Make 3 x 3

```



Ainsi, le paradigme de la programmation gloutonne s'applique dans le cas où l'exploration de toutes les solutions possibles *n'est pas* envisageable en terme de complexité. Il convient alors de déterminer une autre stratégie pour trouver une bonne solution au problème. L'idée est alors de choisir, à chaque étape, une solution *optimale* au problème *local* dans l'espoir que la recombinaison des optima locaux fournira une *bonne* solution au problème *global*. Le terme « bonne » n'est pas choisi au hasard : on accepte de ne pas atteindre l'optimalité au profit d'un algorithme ayant une complexité acceptable. Ainsi, un algorithme de type glouton est basé sur une *heuristique*.

5.4 Conclusion

Nous avons exploré dans ce chapitre trois formes générales de résolution de problème permettant d'obtenir des algorithmes plus efficaces qu'une méthode par force brute.

Le paradigme « diviser pour régner » permet de décomposer un problème en sous-problèmes similaires (voire identiques) de taille plus petite. Les résultats de ces sous-problèmes sont alors recombinés pour fournir le résultat du problème courant. L'ensemble des solutions possibles est alors exploré, ce qui doit rester réalisable en terme de complexité.

Dans le cadre de la programmation dynamique, l'exploration de toutes les solutions possibles reste d'actualité mais l'on mémorise les résultats des sous-problèmes intermédiaires afin d'éviter de les recalculer inutilement et de permettre ainsi de baisser la complexité.

Enfin, la programmation gloutonne abandonne l'exploration de toutes les solutions possibles et recherche une heuristique permettant de choisir, à chaque étape d'un problème, une solution localement optimale avec l'espoir que la recombinaison de ces optima locaux fournit une bonne solution au problème global.

Séance 6

Initiation sommaire à la preuve d'algorithmes

6.1 Introduction

Tout au long de ce cours, nous avons vu comment concevoir des algorithmes en suivant une démarche systématique et scientifique d'analyse, de formalisation, de conception puis d'implantation.

Nous avons décrit l'importance de se poser des questions sur les domaines des entrées et des sorties de nos algorithmes. Nous avons vu comment, par raffinements successifs, il était possible de décomposer un problème initial, *récursivement*, jusqu'à obtenir des sous-problèmes suffisamment simples pour être résolus aisément et *indépendamment* du langage utilisé lors de la phase d'implantation. Nous avons également rapidement évoqué le test des programmes implantant nos algorithmes. Ceci visait à s'assurer que les programmes se comportaient bien selon nos attentes. Néanmoins, tester un programme, quand bien même nous ferions beaucoup de tests, ne peut totalement nous rassurer sur le fait qu'il est conforme à nos attentes. « Testing shows the presence, not the absence of bugs » disait Dijkstra dans les années 60.

Or, rien de plus inutile qu'un algorithme bien conçu, efficace, mais ne répondant pas au problème initialement posé. Pour renforcer notre quête de conception de programmes efficaces et « qui marchent », il est intéressant d'aborder la correction au travers de méthodes rigoureuses et mathématiques, dites « méthodes formelles ».

Cette introduction se veut superficielle et relativement intuitive. Il n'est pas question d'aborder les innombrables domaines et techniques relatifs aux méthodes formelles. Le but est de montrer qu'il est possible d'utiliser des outils relevant de *preuves mathématiques* pour démontrer qu'un algorithme (donc, dans une certaine mesure un programme) est *correct* vis-à-vis de ce que l'on attend de lui : *sa spécification*.

6.2 Spécification

« Prouver un programme » n'a aucun sens dans l'absolu. On peut néanmoins prouver que son *exécution* vérifie certaines *propriétés*. Ainsi, pour prouver qu'un programme est correct, il faut au préalable avoir identifié quelles propriétés décrivent ce que l'on attend des exécutions de ce programme. Ces propriétés sont sa *spécification*.

Pour pouvoir raisonner formellement, mathématiquement sur une propriété, il est nécessaire de l'énoncer dans un langage non ambigu. Celui des mathématiques est donc particulièrement adapté.

6.3 Rappels de logique

Nous utiliserons les symboles habituels permettant d'énoncer des formules logiques du premier ordre.

\wedge	Conjonction (« et »)	\vee	Disjonction (« ou »)
\Rightarrow	Implication (« si »)	\Leftrightarrow	Équivalence (« si et seulement si »)
\neg	Négation (« non »)	\forall	Quantification universelle (« pour tout »)
\exists	Quantification existentielle (« il existe »)		

Nous utiliserons également le mécanisme de preuve par récurrence qui énonce que pour prouver qu'une propriété de la forme $\forall n \in \mathbb{N}, P(n)$ est vraie :

1. il faut prouver que $P(0)$ est vraie
2. et prouver que, si l'on suppose $P(n)$ vraie pour un n quelconque, alors on peut prouver que $P(n+1)$ est vraie.

$$\forall P, (P(0) \wedge \forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)) \Rightarrow \forall n \in \mathbb{N}, P(n)$$

J'ai volontairement laissé dans les lignes ci-dessus la locution « est vraie » car elle fait souvent partie des démonstrations plus moins formelles. Les spécialistes de la logique pourront m'objecter avec raison que ce « est vraie » est redondant, d'autant plus que la signification de « être vraie » est gentiment passée sous silence ici.

6.4 Preuve d'algorithmes récursifs

6.4.1 Correction partielle, correction totale

Pour prouver une propriété sur un algorithme récursif, on utilisera fort naturellement le mécanisme de *preuve par récurrence*. On prouvera donc la propriété dans les *cas de base* puis dans les *cas récursifs*. La correction est dite *partielle* si la propriété est prouvée sous réserve que *l'algorithme termine*. Pour une preuve de correction *totale*, il faut également prouver la *terminaison* de l'algorithme.

6.4.2 Terminaison de fonction récursive

De manière un peu informelle, pour prouver la terminaison d'une fonction récursive, il faut montrer qu'il existe un (ou des) *cas d'arrêt* et que chaque appel récursif est effectué avec des arguments *plus petits* que ceux de l'appel courant et *restant dans le domaine de définition de la fonction*. Nous reviendrons plus en détail, en section 6.4.9, sur ce point pour le rendre plus général. Il convient alors de disposer d'un *ordre bien fondé* sur les arguments (cf. ci-dessous).

Une relation d'ordre $<$ sur un ensemble E est bien fondée s'il n'existe pas de suite infinie décroissante d'éléments de E vis-à-vis de $<$ (ou encore si tout sous-ensemble de E a un élément minimal). Un ordre bien fondé est l'ordre strict associé à une telle relation.

Soit deux ensembles munis d'une relation d'ordre $(X, <_X)$ et $(Y, <_Y)$. On appelle *composition lexicographique* de $<_X$ et $<_Y$ la relation $<_{X \times Y}$ telle que $(x, y) <_{X \times Y} (x', y')$ ssi $x <_X x' \vee (x = x' \wedge y <_Y y')$. On peut démontrer que la composition lexicographique de deux ordres bien fondés est un ordre bien fondé, que l'on nomme *ordre lexicographique*.

L'*ordre produit* est défini comme $(x, y) <_{X \times Y} (x', y')$ ssi $x <_X x' \wedge y <_Y y'$. On peut également démontrer que si $<_X$ et $<_Y$ sont bien fondés alors leur ordre produit l'est également.

6.4.3 Valeur absolue

```
abs (x) =
  si x > 0 alors retourner x
```

```
sinon retourner -x
```

Une propriété concourant à la correction de cet algorithme est que pour un x donné, $\text{abs}(x)$ est toujours positif. Notons que cette propriété ne suffit pas pour caractériser la correction de abs en tant que fonction « valeur absolue », mais c'est celle que nous choisissons de démontrer.

Propriété à prouver $P : \forall x \in \mathbb{Z}, \text{abs}(x) \geq 0$.

Écrivons l'équation de la fonction

$$\text{abs}(x) = \begin{cases} x & \text{si } x > 0 \\ -x & \text{si } x \leq 0 \end{cases}$$

Prouvons P par cas sur x La fonction n'étant pas récursive, il n'y a pas besoin de faire appel à une preuve par récurrence ni de prouver sa terminaison.

- Cas $x > 0$ (1)

Prouvons que $\text{abs}(x) \geq 0$.

○ Par définition de abs , nous avons $\text{abs}(x) = x$ (2)

○ Par (1), (2), $\text{abs}(x) > 0$. (3)

○ Par (3), $\text{abs}(x) \geq 0$. \square

- Cas $x \leq 0$ (1)

Prouvons que $\text{abs}(x) \geq 0$.

○ Par définition de abs , nous avons $\text{abs}(x) = -x$. (2)

○ Par (1) nous déduisons $-x \geq 0$. (3)

○ Par (1), (3), nous avons $\text{abs}(x) \geq 0$. \square

\square

6.4.4 Division euclidienne (1)

```
div (a, b) =
  si a < b alors retourner 0
  sinon retourner 1 + div (a - b, b)
```

Une propriété concourant à la correction de cet algorithme est que pour $a \geq 0$ et $b > 0$, on a $0 \leq \text{div}(a, b) \times b \leq a$.

Propriété à prouver $P : \forall a \in \mathbb{N}, \forall b > 0 \in \mathbb{N}, 0 \leq \text{div}(a, b) \times b \leq a$.

Écrivons l'équation de la fonction

$$\text{div}(a, b) = \begin{cases} 0 & \text{si } a < b \\ 1 + \text{div}(a - b, b) & \text{si } a \geq b \end{cases}$$

Prouvons que div termine

Puisque la fonction est récursive, il faut prouver sa terminaison. Nous devons donc montrer que chaque appel récursif est effectué sur un problème strictement plus petit, restant dans $(\mathbb{N} \times \mathbb{N})$. La fonction a 1 appel récursif, la preuve a donc 1 cas récursif.

- Si $a < b$ alors div termine trivialement.
- Montrons $a \geq b \Rightarrow (a - b, b) < (a, b)$ et que $(a - b) \in \mathbb{N}$ et $b \in \mathbb{N}$.
 - Puisque $a \geq b$, nous avons $a - b \geq 0 \in \mathbb{N}$. Et puisque $b \in \mathbb{N}$... il y reste !
 - Considérons l'ordre lexicographique fondé sur celui des entiers. (1)

- Puisque $b > 0$, et $a \geq 0$, nous avons $(a - b) < a$. (2)
- Par (1), (2), nous avons $(a - b, b) < (a, b)$.

□

Prouvons P par récurrence

Puisque la fonction est récursive, une preuve par récurrence est nécessaire.

- Cas $a < b$ (1)

Prouvons que $0 \leq \text{div}(a, b) \times b \leq a$.

- Par définition de div , nous avons $\text{div}(a, b) = 0$. (2)

- Par (2), $\text{div}(a, b) \times b = 0$. (3)

- Par (3), nous avons $0 \leq \text{div}(a, b) \times b$. (4)

- Puisque $a \in \mathbb{N}$, par (3) nous avons $0 = \text{div}(a, b) \times b \leq a$. (5)

- Par (4), (5) nous avons $0 \leq \text{div}(a, b) \times b \leq a$. □

- Cas $a \geq b$ (1)

Prouvons que $0 \leq \text{div}(a, b) \times b \leq a$.

- Par définition de div , nous avons $\text{div}(a, b) = 1 + \text{div}(a - b, b)$. (2)

Donc nous devons prouver $0 \leq (1 + \text{div}(a - b, b)) \times b \leq a$

càd $0 \leq b + \text{div}(a - b, b) \times b \leq a$ (3)

- Par hypothèse de récurrence, nous avons $\forall i < a, 0 \leq \text{div}(i, b) \times b \leq i$. (4)

- Puisque $a \geq b$ par (1) et $b > 0$, nous avons $a - b < a$. (5)

- Par (5), nous pouvons appliquer l'hypothèse de récurrence (4) et nous avons $0 \leq \text{div}(a - b, b) \times b \leq (a - b)$. (6)

- Par (6), puisque $b > 0$, nous avons $0 \leq b + \text{div}(a - b, b) \times b \leq (a - b) + b$
càd $0 \leq b + \text{div}(a - b, b) \times b \leq a$, ce que nous devions prouver en (3). □

□

6.4.5 Multiplication (1)

```
mult (x, y) =
  si y = 0 alors retourner 0
  sinon retourner x + mult (x, y - 1)
```

Prouvons que cet algorithme est correct vis-à-vis de la multiplication d'entiers positifs ou nuls.

Propriété à prouver $P : \forall x, y \in \mathbb{N}, \text{mult}(x, y) = x \times y$.

Écrivons l'équation de la fonction

$$\text{mult}(x, y) = \begin{cases} 0 & \text{si } y = 0 \\ x + \text{mult}(x, y - 1) & \text{si } y > 0 \end{cases}$$

Prouvons que mult termine

Puisque la fonction est récursive, il faut prouver sa terminaison. Nous devons donc montrer que chaque appel récursif est effectué sur un problème strictement plus petit. La fonction a 1 appel récursif, la preuve a donc 1 cas récursif.

- Si $y = 0$ alors mult termine trivialement.
- Montrons que $y > 0 \Rightarrow (x, y - 1) < (x, y)$ et que $x \in \mathbb{N}$ et $(y - 1) \in \mathbb{N}$.
 - Puisque $y > 0$ nous avons $y - 1 \geq 0 \in \mathbb{N}$ et pour x , il est par hypothèse dans \mathbb{N} .
 - Considérons l'ordre lexicographique fondé sur celui des entiers.

(1)

- Nous avons $x = x$ et $y - 1 < y$. (2)
- CQFD par (1), (2)

□

Prouvons P par récurrence

Puisque la fonction est récursive, une preuve par récurrence est nécessaire.

- Cas $y = 0$ (1)

Prouvons que $\text{mult}(x, 0) = x \times 0$.

- Par définition de mult , on a $\text{mult}(x, 0) = 0$ (2)

- Nous savons que $\forall n \in \mathbb{N}, 0 = n \times 0$ donc $0 = x \times 0$. (3)

- CQFD par (2), (3). □

- Cas $y > 0$ (1)

Prouvons que $\text{mult}(x, y) = x \times y$.

- Par définition de mult , nous avons $\text{mult}(x, y) = x + \text{mult}(x, y - 1)$ (2)

Donc nous devons prouver $x + \text{mult}(x, y - 1) = x \times y$ (3)

- Par (1), $y - 1 < y$ donc hypothèse de récurrence applicable. (4)

- Par hypothèse de récurrence, nous avons $\forall i < y, \text{mult}(x, i) = x \times i$. (5)

- Par (4), (5) nous avons $\text{mult}(x, (y - 1)) = x \times (y - 1)$. (6)

- Par (3), (6) nous devons prouver $x + x \times (y - 1) = x \times y$.

- Donc que $x \times (1 + y - 1) = x \times y$. □

□

6.4.6 Puissance entière (1)

```
pow (x, n) =
    si n = 0 alors retourner 1
    sinon
        soit t = pow (x, n / 2)
        si n % 2 = 0 alors retourner t * t
        sinon retourner x * t * t
```

Prouvons que cet algorithme est correct vis-à-vis de la puissance entre entiers positifs ou nuls.

Propriété à prouver $P : \forall x, n \in \mathbb{N}, \text{pow}(x, n) = x^n$.

Prouvons que pow termine

Puisque la fonction est récursive, il faut prouver sa terminaison. Nous devons donc montrer que chaque appel récursif est effectué sur un problème strictement plus petit. La fonction a 1 appel récursif, la preuve a donc 1 cas récursif.

- Si $n = 0$ alors pow termine trivialement.
- Montrons que $n > 0 \Rightarrow (x, n/2) < (x, n)$ et que $x \in \mathbb{N}$ et $n/2 \in \mathbb{N}$.
 - x est déjà par hypothèse dans \mathbb{N} , quant à $n/2$ il l'est aussi puisque $n \in \mathbb{N}$ et nous faisons une division (entière).
 - Considérons l'ordre lexicographique fondé sur celui des entiers. (1)
 - Nous avons $x = x$ et $n/2 < n$ (puisque $n > 0$). (2)
 - CQFD par (1), (2)

□

Prouvons P par récurrence

Puisque la fonction est récursive, une preuve par récurrence est nécessaire. *Notons que désormais, pour s'épargner quelques étapes de preuve, nous appliquerons l'hypothèse de récurrence en s'assurant silencieusement que les conditions nécessaires sont vérifiées (application sur un problème strictement plus petit).*

- Cas $n = 0$

Prouvons que $\text{pow}(x, 0) = x^0$.

- Par définition de pow , on a $\text{pow}(x, 0) = 1$. (1)

- Nous savons que $\forall n \in \mathbb{N}, n^0 = 1$. (2)

- CQFD par (1), (2). \square

- Cas $n > 0$

Prouvons que $\text{pow}(x, n) = x^n$.

- Par définition de pow , nous avons $t = \text{pow}(x, n/2)$. (1)

- Par hypothèse de récurrence nous avons $\text{pow}(x, n/2) = x^{n/2}$. (2)

- Donc par (2), nous avons $t = x^{n/2}$. (3)

- Deux cas sont à prouver.

 - Cas $n \% 2 = 0$ (4)

 - Par (4) nous avons $n = 2 \times \frac{n}{2}$. (5)

 - Donc $x^n = x^{(2 \times \frac{n}{2})} = x^{n/2} \times x^{n/2}$. (6)

 - Par définition de pow , nous avons $\text{pow}(x, n) = t \times t$. (7)

 - CQFD par (3), (6), (7). \square

 - Cas $n \% 2 \neq 0$ (8)

 - Par (8) nous avons $n = 2 \times \frac{n}{2} + 1$. (9)

 - Donc $x^n = x^{(2 \times \frac{n}{2} + 1)} = x^{n/2} \times x^{n/2} \times x$. (10)

 - Par définition de pow , nous avons $\text{pow}(x, n) = x \times t \times t$. (11)

 - CQFD par (3), (10), (11). \square

6.4.7 Une fonction qui ne termine pas

Revenons quelques instants sur une variante de notre fonction mult de la section 6.4.5, réécrivons-la de la manière incorrecte suivante et intéressons-nous uniquement à sa terminaison.

```
horror (x, y) =
  si y = 0 alors retourner 0
  sinon retourner x + horror (x - 1, y + 1)
```

Nous avons juste effectué l'appel récursif sur $x - 1, y + 1$. Manifestement cette fonction ne termine plus. Mais comment notre preuve de terminaison va-t-elle échouer dans le cas récursif?

Nous sommes bien tentés de dire comme précédemment, « *considérons l'ordre lexicographique fondé sur celui des entiers, nous avons bien $(x - 1, y + 1) < (x, y)$ (car $x - 1 < x$), donc nous faisons un appel récursif sur un problème strictement plus petit, donc la fonction termine*

L'erreur de raisonnement est que nous n'avons pas vérifié que l'argument (en fait, le couple composé des deux arguments) de l'appel récursif restait bien dans le domaine de définition de horror . Pour que ce soit le cas, il faut que $x - 1 \in \mathbb{N}$ et $y + 1 \in \mathbb{N}$.

Pour $y + 1$ c'est bien le cas puisque par hypothèse $y \in \mathbb{N}$. Par contre, pour $x - 1$, nous avons juste l'hypothèse que $x \in \mathbb{N}$, donc si $x = 0$, la valeur de $x - 1$ n'est plus dans \mathbb{N} . D'ailleurs existe-t-elle? Donc nous ne pouvons pas prouver cette obligation et notre preuve échoue.

6.4.8 Une fonction qui ne termine (toujours) pas

Faisons une autre modification incorrecte de notre fonction mult d'origine :

```

horror2 (x, y) =
  si x = 0 alors retourner 0
  sinon retourner x + horror2 (x, y - 1)

```

Nous avons sciemment remplacé le test par $x = 0$. Ainsi, nous effectuons le test d'arrêt sur le mauvais paramètre. Lorsque nous allons essayer de prouver que $(x, y - 1)$ reste dans $\mathbb{N} \times \mathbb{N}$, nous allons échouer pour $y - 1$ puisque nous n'avons comme hypothèse que $y \in \mathbb{N}$.

Ainsi, même si l'ordre lexicographique sur x et y aurait permis d'exhiber une décroissance stricte, cette décroissance n'aurait pas été bornée. La preuve de terminaison échoue donc à nouveau.

6.4.9 Terminaison de fonction récursive un peu plus en détail

En section 6.4.2, nous avons un peu rapidement dit que « *les appels récursifs devaient se faire sur des arguments strictement plus petits* » (et « *restant dans le domaine de la fonction* »). Nous avions alors invoqué un ordre bien fondé à appliquer aux arguments de nos fonctions. Par chance, à chaque fois que nous avions un seul argument, nous prenions l'ordre habituel $<$ sur les entiers. Et à nouveau par chance, lorsque nous avions deux arguments, l'ordre lexicographique convenait.

Pour être plus précis et plus général, on ne compare pas forcément *directement* les arguments pour vérifier leur décroissance. Ce que l'on compare c'est une *fonction des arguments* et c'est elle qui doit strictement décroître. Pour le cas de la factorielle où nous n'avions qu'un seul argument et où nous avions conclu que $n - 1 < n$, la fonction appliquée à l'argument était trivialement l'identité.

L'important est que cette fonction des arguments ait un résultat dans un ensemble muni d'un ordre bien fondé, garantissant ainsi la terminaison par absence de chaîne infinie décroissante. Lorsque cette fonction est à résultat dans \mathbb{N} , on l'appelle parfois une *mesure* et elle a donc naturellement la propriété d'être ≥ 0 (puisque à résultat dans \mathbb{N}). Lorsque l'on détermine une mesure, il est nécessaire de démontrer qu'elle et bien toujours ≥ 0 pour réussir la preuve.

Présentons de manière la plus générale la manière de prouver la terminaison d'une fonction récursive :

- Soit f une fonction récursive de A dans B .
- Trouver une application μ de A dans $(E, <_E)$ un ensemble muni d'un ordre bien fondé. Notons que, comme dit précédemment, lorsque E est \mathbb{N} muni de $<$, la fonction μ est appelée une *mesure*. Le plus difficile est de trouver cette fonction μ , l'ensemble E et l'ordre $<_E$ afin qu'ils permettent la suite de la preuve.
- Déterminer :
 - \mathcal{B} le sous-ensemble de A tel que $\forall x \in \mathcal{B}, \mu(x)$ soit un élément minimal de $\mu(A)$. Cet ensemble représente les cas de *base* de la récursion.
 - \mathcal{I} le sous-ensemble de A tel que les arguments des appels récursifs soient *invalides* (i.e. $\notin A$).
- Prouver que :
 - f termine pour tout $x \in \mathcal{B} \cup \mathcal{I}$
 - à chaque appel récursif de $f(x)$ de la forme $f(x')$ on a $\mu(x') <_E \mu(x)$

Comme précisé, μ doit être trouvée en réfléchissant « bien ». En règle générale, on a une idée de la raison pour laquelle notre fonction f termine, on « sent » ce qui diminue jusqu'à faire « toucher le fond » et s'arrêter. C'est donc sur la base de cette intuition que l'on détermine μ . Mais ce n'est pas toujours simple...

Le choix de cette fonction est totalement libre. Pour une fonction f ayant un unique argument entier, ce sera naturellement l'identité induisant l'unique cas de base $\{0\}$.

Pour une fonction avec deux arguments entiers, on pourra choisir également l'identité, impliquant que $E = (\mathbb{N} \times \mathbb{N})$ muni de l'ordre lexicographique sur les entiers, induisant l'unique cas de base $\{(0, 0)\}$. Mais nous pourrions choisir $\mu(x, y) = x$ (multiples cas de base $\{(0, y)\}$ quel que soit y) ou $\mu(x, y) = x + y$ (multiples cas de base $\{(0, y)(x, 0)\}$ quels que soient x et y).

Pour une fonction à un argument étant une liste, on pourra prendre la longueur de la liste (son nombre d'éléments) avec $E = \mathbb{N}$ et l'ordre habituel sur les entiers.

Application 1 : reprenons l'exemple de la fonction *mult* de la section 6.4.5 et appliquons ce schéma de preuve de terminaison à la lettre.

$$\text{mult}(x, y) = \begin{cases} 0 & \text{si } y = 0 \\ x + \text{mult}(x, y - 1) & \text{si } y > 0 \end{cases}$$

- Choisissons $\mu(x, y) = y$ donc $E = \mathbb{N}$ muni de $<$.
- Déterminons :
 - \mathcal{B} le sous-ensemble de $(\mathbb{N} \times \mathbb{N})$ tel que $\mu(x, y) = y$ soit un élément minimal de $\mu(\mathbb{N}, \mathbb{N}) = \mathbb{N}$. Il vient naturellement que $\mathcal{B} = \{(x, 0)\}$ quel que soit x .
 - \mathcal{I} le sous-ensemble de $(\mathbb{N} \times \mathbb{N})$ tel que les arguments récursifs soient invalides, donc x ou $y \notin \mathbb{N}$. Pour être valide $y - 1$ doit être ≥ 0 , donc $y \geq 1$. D'où $\mathcal{I} = \{(x, 0)\}$ quel que soit x .
- Prouvons que :
 - f termine pour tout $x \in \mathcal{B} \cup \mathcal{I} = \{(x, 0)\}$: vrai puisque le test $y=0$ nous permet de retourner 0 sans aucun appel récursif.
 - $\mu(x, y - 1) < \mu(x, y)$ donc que $y - 1 < y$, ce qui est vrai.

La fonction *mult* termine donc.

Remarquons que nous aurions pu choisir $\mu(x, y) = (x, y)$ (i.e. l'identité) et $E = (\mathbb{N} \times \mathbb{N})$ muni de l'ordre lexicographique. Nous aurions trouvé les mêmes \mathcal{B} et \mathcal{I} . La preuve de $\mu(x, y - 1) < \mu(x, y)$ aurait été celle que $(x, y - 1) < (x, y)$, qui tombe immédiatement par définition de l'ordre lexicographique.

Application 2 : essayons de prouver la terminaison d'une fonction pour laquelle l'ordre lexicographique ne convient pas. Nous définissons la fonction *weird* par :

```
weird (x, y) =
  si x < 2 ou y < 2 alors retourner 0
  sinon
    si une condition aléatoire retourner (x - 2, y + 1)
    sinon retourner weird (x + 1, y - 2)
```

Même si nous ne savons pas comment représenter la « condition aléatoire » interne, nous avons suffisamment d'information pour prouver que la fonction termine. L'ordre lexicographique est clairement inutile ici puisque dans un cas x décroît et dans l'autre il croît (idem pour y). Par contre, nous pouvons remarquer que l'argument qui décroît le fait toujours plus que celui qui croît. Donc la somme des deux arguments ne fait que décroître. Elle semble donc une bonne candidate en tant que fonction μ .

- Choisissons $\mu(x, y) = x + y$ donc $E = \mathbb{N}$ muni de $<$.
- Déterminons :
 - \mathcal{B} le sous-ensemble de $(\mathbb{N} \times \mathbb{N})$ tel que $\mu(x, y) = x + y$ soit un élément minimal de $\mu(\mathbb{N}, \mathbb{N}) = \mathbb{N}$. Il vient naturellement que $\mathcal{B} = \{(x, 0)(0, y)\}$ quels que soient x et y .
 - \mathcal{I} le sous-ensemble de $(\mathbb{N} \times \mathbb{N})$ tel que les arguments récursifs soient invalides, donc $\notin \mathbb{N}$. Pour $x + 1$ et $y + 1$ il n'y a pas de cas invalides. Par contre, pour être valide $x - 2$ doit être ≥ 0 , donc $x \geq 2$. Idem pour $y - 2$. D'où $\mathcal{I} = \{(0, y)(1, y)(x, 0)(x, 1)\}$ quels que soient x et y .
- Prouvons que :
 - f termine pour tout $x \in \mathcal{B} \cup \mathcal{I} = \{(0, y)(1, y)(x, 0)(x, 1)\}$: vrai puisque le test $x < 2$ ou $y < 2$ nous permet de retourner 0 sans aucun appel récursif dans chacun de ces 4 cas.
 - $\mu(x - 2, y + 1) < \mu(x, y)$ c'est à dire que $x - 2 + y + 1 < x + y$, donc $x + y - 1 < x + y$: vrai
 - $\mu(x + 1, y - 2) < \mu(x, y)$ c'est à dire que $x + 1 + y - 2 < x + y$, donc $x + y - 1 < x + y$: vrai

La fonction *weird* termine donc.

Nous pouvons remarquer que le test $x < 2$ ou $y < 2$ est primordial pour que *weird* soit totalement définie sur son domaine, donc termine. Si nous l'avions écrite comme :

```
weird (x, y) =
  si x = 0 ou y = 0 alors retourner 0
```

```

sinon
  si une condition aléatoire retourner (x - 2, y + 1)
  sinon retourner weird (x + 1, y - 2)

```

alors si par malheur, en calculant $\text{weird}(3, 2)$, la condition aléatoire était toujours vraie, la fonction n'aurait pas terminé ! Il nous manquerait alors des cas de base dans la définition de la fonction, nous permettant de « sauter sous 0 » pour x dans cet exemple.

Nous l'aurons vu dans la preuve en vérifiant si f termine pour tout $x \in \mathcal{B} \cup \mathcal{I} = \{(0, y)(1, y)(x, 0)(x, 1)\}$ puisque pour $(1, y)$ et $(x, 1)$ nous n'aurions pas eu de terminaison.

6.5 Preuve d'algorithmes avec des boucles

Dès lors qu'un algorithme comporte une boucle, tout comme lorsqu'il est récursif, il n'est plus possible de raisonner linéairement en s'assurant que la propriété à démontrer est vraie. En effet, une boucle peut s'appuyer sur des calculs d'une itération précédente.

Sans formaliser la logique sous-jacente (logique de Hoare), nous allons voir ici quels principes appliquer pour démontrer une propriété en présence de boucles.

Pour démontrer la correction d'un tel algorithme, tout comme dans le cas récursif, il faut démontrer la *terminaison* et la propriété caractérisant la *correction*. L'ordre de réalisation de ces deux preuves n'a pas d'importance, mais bien souvent la preuve de terminaison donne une bonne idée de comment mener celle de correction.

Par convention, lorsque nous avons une affectation $x = \dots x \dots$, nous notons x la valeur de la variable *avant* l'affectation et x' sa valeur *après* l'affectation. Afin d'éviter la confusion, dans les algorithmes et dans les preuves, entre l'affectation et l'égalité, la première sera syntaxiquement notée $\mathbf{:=}$ et la seconde $=$.

6.5.1 Correction (partielle)

Pour exprimer qu'un algorithme est correct (vis-à-vis d'une propriété P), pour chaque boucle il faut exhiber une propriété I nommée *invariant*, permettant de *déduire* P , telle que :

- I est *vraie* avant le *premier passage* dans la boucle,
- si I est *vraie* en début d'itération de boucle, alors I est *vraie* en fin d'itération de boucle. La preuve que I est *vraie* en fin de boucle pourra nécessiter l'hypothèse que la condition de boucle est *vraie*. Formellement, on prouvera $\text{cond-boucle} \Rightarrow I$.

Le corollaire de cette définition est que l'invariant sera *vrai* lorsque la boucle sera *terminée* (sortie de la boucle).

Une fois la preuve que l'invariant est vérifié en sortie de boucle, il faudra en déduire la propriété P recherchée. Cette preuve pourra bien sûr s'appuyer sur l'hypothèse que la condition de boucle est devenue *fausse*. Formellement, on prouvera $(\neg \text{cond-boucle} \wedge I) \Rightarrow P$.

6.5.2 Terminaison de boucle

Pour démontrer la terminaison d'une boucle, il faut identifier un *variant* V , c'est-à-dire une fonction des variables du programme (une expression) :

- ≥ 0 tant que l'on *rentre* dans la boucle,
- qui *décroît strictement* à chaque itération.

Le corollaire de cette définition est que, arrivé à une certaine valeur, le variant rend la condition d'exécution de la boucle fausse : la boucle se *termine*.

La garantie de terminaison provient du fait qu'il n'existe pas de suites infinies décroissantes dans \mathbb{N} .

Il peut exister plusieurs variants possibles pour une même boucle. Un variant peut être une fonction des variables du programme au sens le plus large. Ainsi, il est possible de le définir par cas. Considérons l'algorithme suivant :

```

foo (x) =
tant que x > 0
x := -3 * x + 15

```

On peut se convaincre que la fonction ι définie par les cas suivants est un variant de boucle.

$$\iota(x) = \begin{cases} \text{Si } x \leq 0 & \rightarrow 0 \\ \text{Si } x = 1 \text{ ou } 2 \text{ ou } 3 & \rightarrow 2 \\ \text{Si } x = 4 & \rightarrow 3 \\ \text{Si } x \geq 5 & \rightarrow 1 \end{cases}$$

On peut remarquer que cette fonction est en fait le nombre de tours de boucle effectués en fonction de la valeur de x .

x_0	Suite des x_i	Nombre tours
0	[]	0
1	[12, -21]	2
2	[9, -12]	2
3	[6, -3]	2
4	[3, 6, -3]	3
5	[0]	1
6	[-3]	1
7	[-6]	1

6.5.3 Identité

```

id (x) =
i := x
r := 0
tant que i > 0
    r := r + 1
    i := i - 1
retourner r

```

Pour commencer doucement, considérons la fonction id qui n'est rien d'autre que l'identité sur les entiers naturels, écrite de façon bien compliquée pour ce qu'elle fait. La correction de cet algorithme nécessite que id termine et que pour tout $x \geq 0$, r soit égal à x .

Propriété à prouver $P : \forall x \in \mathbb{N}, r = id(x) \Rightarrow r = x$.

Terminaison

La boucle tourne tant que $i > 0$ donc i est bien positif ou nul tant que l'on rentre dans la boucle et i est bien strictement décroissant car $i' = i - 1$. Donc i est bien un variant de boucle.

Correction

On remarque qu'à chaque tour de boucle, $x = r + i$ et d'ailleurs, à la fin i sera égal à 0 donc on aura $x = r$. Donc prenons l'invariant $I : x = r + i$ et prouvons qu'il est bien un invariant.

- Prouvons I avant le premier passage dans la boucle.
 - Prouvons que $x = r + i$.
 - Par définition de id , nous avons $r = 0$ et $i = x$. (1)
 - Donc par (1) on a bien $x = r + i = 0 + x$ \square

- Prouvons I à la fin d'un passage dans la boucle sous l'hypothèse que I était vraie en début de boucle.

$$r' = r + 1 \quad (1)$$

$$i' = i - 1 \quad (2)$$

Par hypothèse de récurrence, $x = r + i$. (3)

Prouvons que $x = r' + i'$

- Par (1), (2), prouvons que $x = r + 1 + i - 1$

- Donc que $x = r + i$

- CQFD par (3), (4) \square

\square

Nous avons démontré que I est bien un invariant de boucle. Revenons maintenant à notre propriété de correction P . Nous devons la déduire de I . Nous savons qu'à la fin de la boucle, $i \leq 0$ (1) puisque la condition de boucle est fausse. Puisque initialement $i = x$, que par hypothèse de P , $x \in \mathbb{N}$, et que i diminue de 1 à chaque fois, nous déduisons de (1) que nous avons $i = 0$. Donc par (I) nous avons bien $x = r + i = r + 0$, donc $x = r$.

Si l'on ne rentre pas dans la boucle, alors $i = x$, $r = 0$ et $i \leq 0$. Comme i est décrémenté à chaque tour de boucle, $i = 0$. Donc $r = x$. \square

6.5.4 Fausse l'identité

Modifions très légèrement notre algorithme précédent en remplaçant la condition de boucle $i > 0$ par $i \geq 0$ et regardons l'impact sur la preuve de correction.

```
wrongid (x) =
    i := x
    r := 0
    tant que i >= 0
        r := r + 1
        i := i - 1
    retourner r
```

L'intuition nous dit que la boucle va faire un tour de plus, donc que cette fonction retournera $x + 1$ et non x comme dans le cas de id . Ainsi, la correction de cet algorithme nécessite que $wrongid$ termine et que pour tout $x \geq 0$, r soit égal à $x + 1$.

Terminaison

La boucle tourne tant que $i \geq 0$ donc i est bien positif ou nul tant que l'on rentre dans la boucle et i est bien strictement décroissant car $i' = i - 1$. Donc i est bien un variant de boucle.

Correction

On remarque que, hormis le nombre de tours, le comportement de $wrongid$ est le même dans la boucle que celui de id . Sans surprise l'invariant de boucle sera le même que dans la section 6.5.3, ainsi que la preuve qui en découle.

Mais pourquoi cette preuve, qui permettait de démontrer la correction de id , reste-t-elle valide pour démontrer celle de $wrongid$?

La réponse réside dans le fait que la propriété P n'est pas la même que celle de id et que la valeur de i en sortie de boucle n'est pas non plus la même. Terminons la preuve pour s'en rendre compte.

Nous avons démontré que I est bien un invariant de boucle. Revenons maintenant à notre propriété de correction P . Nous devons la déduire de I . Nous savons qu'à la fin de la boucle, $i < 0$ (1) puisque la condition de boucle est fausse. Puisque initialement $i = x$, que par hypothèse de P , $x \in \mathbb{N}$, et que i diminue de 1 à chaque fois, nous déduisons de (1) que nous avons $i = -1$. Donc par (I) nous avons bien $x = r + i = r - 1$, donc $x + 1 = r$.

Par hypothèse de P on a $x \in \mathbb{N}$. Donc il est impossible de ne pas rentrer dans la boucle.

□

6.5.5 Division euclidienne (2)

```
div (a, b) =
  r := a
  q := 0
  tant que r >= b
    r := r - b
    q := q + 1
  retourner q, r
```

La correction de cet algorithme nécessite que div termine et que pour $a \geq 0$ et $b > 0$, q soit tel que $a = b \times q + r$ et r soit tel que $0 \leq r < b$.

Propriété à prouver $P : \forall a \in \mathbb{N}, \forall b > 0 \in \mathbb{N}, (q, r) = div(a, b) \Rightarrow a = b \times q + r \wedge 0 \leq r < b$.

Terminaison

Trouvons une fonction de r et de b qui est ≥ 0 tant que l'on entre dans la boucle et qui décroît strictement à chaque tour de boucle.

La boucle tourne tant que $r \geq b$, donc tant que $r - b \geq 0$. Donc $r - b$ est bien positif ou nul tant que l'on rentre dans le boucle et lorsque $r - b < 0$ la boucle s'arrête.

Nous avons bien $r - b$ strictement décroissant car $r' = r - b$ avec $b > 0$ par hypothèse. Donc c'est bien un variant de boucle.

Correction

On remarque qu'à chaque tour de boucle, $a = b \times q + r$. Nous allons prendre cette propriété comme invariant en la renforçant par $0 \leq r$. La raison de ce renforcement, a priori non justifié, provient du fait que sans lui, il nous sera impossible de prouver P . Il arrive parfois que l'on choisisse un invariant et que l'on se rende compte en cours de preuve qu'il est trop faible. Prenons donc $I : a = b \times q + r \wedge 0 \leq r$ et prouvons que c'est bien un invariant.

- Prouvons I avant le premier passage dans la boucle.
 - Prouvons $a = b \times q + r$.
 - Par définition de div , nous avons $r = a$ et $q = 0$. (1)
 - Donc par (1) on a bien $r = b \times 0 + r$. □
 - Prouvons $0 \leq r$.
 - Par définition de div , nous avons $r = a$. (1)
 - Par hypothèse de P , $a \in \mathbb{N}$. (2)
 - Par (1), (2), $0 \leq a = r$ □
- Prouvons I à la fin d'un passage dans la boucle sous l'hypothèse que P était vraie en début de boucle.

$$r' = r - b \quad (1)$$

$$q' = q + 1 \quad (2)$$
 Par hypothèse de récurrence, $a = b \times q + r$. (3)
 Par hypothèse de condition de boucle, on a $r \geq b$ (4)
 - Prouvons $a = b \times q' + r'$.
 - Par (1), (2), $b \times q' + r' = b \times (q + 1) + (r - b)$
 - $= b \times q + b + r - b$
 - $= b \times q + r$ (5)
 - $= a$ par (3), (5). □

- Prouvons $0 \leq r'$.
 - Par (1) il faut prouver que $0 \leq r - b$.
 - Par (4), $r \geq b$ donc $b \leq r$ donc $0 \leq r - b$. □

Nous avons démontré que I est bien un invariant de boucle. Revenons maintenant à notre propriété de correction P . Nous devons la déduire de I . Nous savons qu'à la fin de la boucle, $r < b$ (1) puisque la condition de boucle est fausse.

Nous devons prouver $a = b \times q + r \wedge 0 \leq r < b$. L'invariant nous donne directement $a = b \times q + r \wedge 0 \leq r$. Il nous il nous reste donc à prouver $r < b$ qui est directement (1).

Remarquons que si nous n'avions pas renforcé notre invariant, avec uniquement $a = b \times q + r$ et $r < b$ nous n'aurions pas pu prouver $0 \leq r$.

Si l'on ne rentre pas dans la boucle, alors $r = a$, $q = 0$ et $r < b$. On a donc bien $a = b \times 0 + a$ et $0 \leq a < b$. □

6.5.6 Division euclidienne (3)

```
div (a, b) =
  r := a
  q := 0
  tant que r >= b
    r := r - b
    q := q + 1
  retourner q
```

Nous reprenons l'algorithme de la section 6.5.5 en ne lui faisant retourner que le quotient q . Démontrons une autre propriété.

Propriété à prouver $P : \forall a \in \mathbb{N}, \forall b > 0 \in \mathbb{N}, \text{div}(a, b) \times b \leq a$.

Terminaison

Cette démonstration a déjà été faite en section 6.5.5 puisque nous n'avons pas changé d'algorithme (à l'exception de la valeur qu'il retourne).

Correction

Tentons de prendre directement la propriété P comme invariant I .

- Prouvons I avant le premier passage dans la boucle.
 - Par définition de div , il faut prouver que $q \times b \leq a$.
 - Par définition de div , on a : $q = 0$. (1)
 - Par hypothèse de P , on a : $a \geq 0$. (2)
 - Par (1), (2), on a bien $0 \times b \leq a$. □
- Prouvons I à la fin d'un passage dans la boucle sous l'hypothèse que P était vraie en début de boucle.
 - $q' = q + 1$ (1)
 - Par hypothèse de récurrence, $q \times b \leq a$. (2)
 - Prouvons que $q' \times b \leq a$.
 - Par (1), (2), $q' \times b = (q + 1) \times b$
 - $= q \times b + b$ (3)
 - Par (2), (3) on a : $q \times b + b \leq a + b$.
 - ...Et là on est coincé car l'on cherche à démontrer $q \times b + b \leq a$.

Il se trouve que nous avons directement pris notre propriété comme invariant de boucle. Or cet invariant est *trop faible* pour démontrer notre propriété. Il nous faut un invariant plus fort, nous permettant de savoir

quelque chose sur b et dont nous pourrons *ensuite déduire* P . Et sans surprise, celui utilisé en section 6.5.5 fera l'affaire.

6.5.7 Puissance entière (2)

```
pow (x, n) =
  p := 1
  m := n
  tant que m > 0
    p := p * x
    m := m - 1
  retourner p
```

Prouvons que cet algorithme est correct vis-à-vis de la puissance entre entiers positifs ou nuls.

Propriété à prouver $P : \forall x, n \in \mathbb{N}, \text{pow}(x, n) = x^n$.

Terminaison

La boucle tourne tant que $m > 0$, donc m est bien positif ou nul tant que l'on rentre dans la boucle et m est bien strictement décroissant car $m' = m - 1$. Donc m est bien un variant de boucle.

Correction

En examinant l'algorithme, on remarque que p accumule les puissances à chaque tour de boucle. Au premier tour $m = n$ et $p = x^1$, au second tour $m = n - 1$ et $p = x^2$, etc. Quand on arrive à la fin de la boucle $m = 0$ et $p = x^n$. Cela nous suggère l'invariant $I : p = x^{n-m}$. Prouvons donc que I est bien un invariant.

- Prouvons I avant le premier passage dans la boucle.
 - Par définition de pow , on a $p = 1$. (1)
 - Par définition de pow , on a $m = n$. (2)
 - Par (1), (2) on a $x^{m-n} = x^{m-m} = x^0 = 1 = p$. \square
- Prouvons I à la fin d'un passage dans la boucle sous l'hypothèse que I était vraie en début de boucle.

$$p' = p \times x \quad (1)$$

$$m' = m - 1 \quad (2)$$
 Par hypothèse de récurrence, $p = x^{n-m}$ (3)
 Prouvons que $p' = x^{n-m'}$
 - Par (1), (2), il faut prouver que $p \times x = x^{n-(m-1)}$.
 - Donc que $p \times x = x^{n-m+1}$.
 - Donc que $p \times x = x \times x^{n-m}$.
 - CQFD par (3) \square

\square

Nous avons démontré que I est bien un invariant de boucle. Revenons maintenant à notre propriété de correction P . Nous devons la déduire de I . Nous savons qu'à la fin de la boucle, $p = x^{n-m}$ (1) et que $m \leq 0$ (2) puisque la condition de boucle est fausse. Puisque initialement $m = n$, que par hypothèse de P , $n \in \mathbb{N}$, et que m diminue de 1 à chaque fois, nous déduisons de (2) que nous avons $m = 0$. Donc $p = x^{n-m} = x^{n-0} = x^n$ et p est bien la valeur renournée par l'algorithme. \square

Si l'on ne rentre pas dans la boucle, alors $m = n = 0$ et $p = 1$, donc on a bien $\text{pow}(x, n) = \text{pow}(x, 0) = x^0 = 1$. \square

6.5.8 Multiplication (2)

```

mult (a, b) =
  p := 0
  y := b
  tant que y > 0
    p := p + a
    y := y - 1
  retourner p

```

Prouvons que cet algorithme est correct vis-à-vis de la multiplication d'entiers positifs ou nuls.

Propriété à prouver $P : \forall a, b \in \mathbb{N}, p = \text{mult}(a, b) \Rightarrow p = a \times b$.

Terminaison

La condition de boucle et la décrémentation de la variable de boucle sont les mêmes que pour l'algorithme de calcul de puissance de la section 6.5.7. Aussi, le variant et sa preuve sont identiques à ce qui a été démontré dans cette précédente section.

Correction

En examinant l'algorithme, on remarque que p accumule les produits à chaque tour de boucle. Au premier tour $p = a \times 1$ et il reste encore $b - 1$ tours à faire. Au second tour $p = a \times 2$ et il reste encore $b - 2$ tours à faire. Cela nous suggère l'invariant $I : p = a \times (b - y)$. Prouvons donc que I est bien un invariant.

- Prouvons I avant le premier passage dans la boucle.
 - Par définition de mult , on a : $p = 0$. (1)
 - Par définition de mult , on a : $y = b$. (2)
 - Par (1), (2), on a : $a \times (b - y) = a \times (b - b) = a \times 0 = p$. \square
- Prouvons I à la fin d'un passage dans la boucle sous l'hypothèse que I était vraie en début de boucle.

$$p' = p + a \quad (1)$$

$$y' = y - 1 \quad (2)$$
 Par hypothèse de récurrence, $a \times (b - y)$ (3)
 Prouvons que $p' = a \times (b - y')$
 - Par (1), (2), il faut prouver que $p + a = a \times (b - (y - 1))$.
 - Donc que $p + a = a \times (b - y + 1)$.
 - Donc que $p + a = a \times b - a \times y + a$.
 - Donc que $p = a \times b - a \times y$.
 - Donc que $p = a \times (b - y)$.
 - CQFD par (3). \square

\square

Nous avons démontré que I est bien un invariant de boucle. Revenons maintenant à notre propriété de correction P . Nous devons la déduire de I . Nous savons qu'à la fin de la boucle, $p = a \times (b - y)$ (1) et que $y \leq 0$ (2) puisque la condition de boucle est fausse. Puisque initialement $y = b$, que par hypothèse de P , $b \in \mathbb{N}$, et que y diminue de 1 à chaque fois, nous déduisons de (2) que nous avons $y = 0$. Donc par (1), $p = a \times (b - y) = a \times (b - 0) = a \times b$ et p est bien la valeur renvoyée par l'algorithme. \square

Si l'on ne rentre pas dans la boucle, alors $y = b = 0$ et $p = 0$ donc on a bien $\text{mult}(a, b) = \text{mult}(a, 0) = a \times 0 = 0$. \square

6.5.9 Minimum dans un tableau

```

min (t, size) =
  i := 1
  r := t[0]
  tant que i < size

```

```

    si t[i] < r alors r := t[i]
    i := i + 1
retourner r

```

Prouvons que cet algorithme est correct c'est-à-dire qu'il retourne le plus petit élément du tableau t dont la taille $size$ est supposée strictement supérieure à 0 (on évite ainsi le cas du tableau vide qui serait une erreur).

Propriété à prouver $\forall i \in \mathbb{N}, \forall size > 0 \in \mathbb{N}, i < size \Rightarrow r = \min(t, size) \Rightarrow r \leq t[i]$.

Notons que cette formule n'est pas tout à fait bien exprimée car on n'introduit pas r et t qui devraient également être quantifiés universellement. Cette liberté est prise par soucis de concision. De plus, nous utilisons la notation $t[0; i]$ pour représenter la séquence des éléments d'un tableau t entre les indices 0 inclus et i exclus et $t[0; i]$ quand i est inclus.

Terminaison

Trouvons une fonction de i et de $size$ qui est ≥ 0 tant que l'on reste dans la boucle et qui décroît strictement à chaque tour de boucle.

La boucle tourne tant que $i < size$, donc tant que $0 < size - i$, donc $size - i$ est bien positif ou nul tant que l'on rentre dans la boucle et $size - i$ est bien strictement décroissant car $i' = i + 1$ et $size$ est constant. Donc $size - i$ est bien un variant de boucle.

Correction

En examinant l'algorithme, on remarque qu'au fur et à mesure de l'avancée dans le tableau, r est le plus petit élément parmi ceux déjà balayés. Cela nous suggère l'invariant $I : \forall j < i, r \leq t[j]$. Prouvons donc que I est bien un invariant.

- Prouvons I avant le premier passage dans la boucle.
 - Par définition de \min , on a : $i = 1$. (1)
 - Par définition de \min , on a : $r = t[0]$. (2)
 - Donc par (1), $j \in [0; 0]$. (3)
 - Donc par (2), (3), on a : $r = t[0] \leq t[0]$. \square
- Prouvons I à la fin d'un passage dans la boucle sous l'hypothèse que I était vraie en début de boucle.
 - $r' = \text{si } t[i] < r \text{ alors } t[i] \text{ sinon } r$ (1)
 - $i' = i + 1$ (2)
 - Par hypothèse de récurrence, $\forall j < i, r \leq t[j]$. (3)

Prouvons que $\forall j < i', r' \leq t[j]$.

 - Par (1), (2), il faut prouver que $\forall j < i + 1, r' \leq t[j]$.
 - Examinons le cas $j = i$.
 - Si $t[i] < r$
 - Alors par (1) issue de la définition de \min , on a $r' = t[i]$.
 - Donc on a $r' \leq t[i]$ et $r' < r$. (4)
 - Or par (2) r était le plus petit des éléments dans $t[0; i]$.
 - Puisque par (4) $r' < r$, on a $r' \leq t[0; i]$. \square
 - Sinon, $t[i] \geq r$ (5)
 - Alors par définition de \min , on a $r' = r$. (6)
 - Or par (3) r était le plus petit des éléments dans $t[0; i]$.
 - Puisque par (6) $r' = r$, par (5) on a $r' \leq t[0; i]$. \square

\square

Nous avons démontré que I est bien un invariant de boucle. Revenons maintenant à notre propriété de correction P . Nous devons la déduire de I . Nous savons qu'à la fin de la boucle, $i \geq size$ (1) et que

$\forall j < i, r \leq t[j]$. Puisque initialement $i = 1$ et i augmente de 1 à chaque itération, nous en déduisons que $i = \text{size}$. Donc par (1), nous avons $\forall j < \text{size}, r \leq t[j]$ et r est bien la valeur renournée par l'algorithme.

Si l'on ne rentre pas dans la boucle, alors $\text{size} = 1$, r est le seul élément du tableau, donc son élément minimal. \square

6.5.10 Appartenance à un tableau

```
mem (x, t, size) =
  tr := faux
  i := 0
  tant que i < size et non tr
    si t[i] = x alors tr := vrai
    i := i + 1
  retourner tr
```

Prouvons que cet algorithme est correct, autrement dit qu'il renvoie vrai si x appartient au tableau t et renvoie faux si x n'appartient pas au tableau t . Par hypothèse $\text{size} \geq 0$.

Propriété à prouver $P : \forall x, t, \forall \text{size} \in \mathbb{N}, \text{tr} = \text{mem}(x, t, \text{size}) \Leftrightarrow x \in t[0; \text{size}]$. Nous reformulons cette propriété pour l'abréger en utilisant tr au lieu de mem sous la forme $P : \forall x, t, \forall \text{size} \in \mathbb{N}, \text{tr} \Leftrightarrow x \in t[0; \text{size}]$. Une fois de plus, par soucis de simplification et de concision, cette formule n'est pas tout à fait bien exprimée puisque l'on ne donne pas le domaine de x ni de t .

Terminaison

Trouvons une fonction de i , et de size qui est ≥ 0 tant que l'on reste dans la boucle et qui décroît strictement à chaque tour de boucle. Notons qu'il n'est pas nécessaire de faire intervenir tr puisque si la fonction trouvée décroît strictement, cela suffira à terminer la boucle.

Comme nous avons la même condition d'arrêt sur i et la même décrémentation de i que dans l'algorithme de la section 6.5.9, nous avons le même variant.

Correction

En examinant l'algorithme, on remarque que tr est vrai si et seulement si, pour un i courant donné, x a été trouvé dans $t[0; i]$. Cela nous suggère l'invariant $\text{tr} \Leftrightarrow x \in t[0; i]$ qui est une propriété plus « locale » que la propriété P que nous voulons prouver. Afin de ne pas bloquer sur la dernière étape de preuve (*spoil*), nous allons renforcer l'invariant par une relation entre i et size . Nous prenons donc $I : \text{tr} \Leftrightarrow (x \in t[0; i] \wedge i \leq \text{size})$.

- Prouvons I avant le premier passage dans la boucle.
 - Prouvons $\text{tr} \Rightarrow (x \in t[0; 0] \wedge 0 \leq \text{size})$.
 - Par définition de mem , on a $\text{tr} = \text{faux}$.
 - Donc faux impliquant vrai, CQFD. \square
 - Prouvons $(x \in t[0; 0] \wedge 0 \leq \text{size}) \Rightarrow \text{tr}$.
 - x ne peut pas appartenir au tableau vide donc la partie gauche de l'implication est forcément fausse.
 - Donc faux impliquant vrai, CQFD. \square
- Prouvons I à la fin d'un passage dans la boucle sous l'hypothèse que I était vraie en début de boucle.
 - $\text{tr}' = \text{vrai}$ si $t[i] = x$ sinon tr (1)
 - $i' = i + 1$ (2)
 - Par hypothèse de récurrence, $\text{tr} \Leftrightarrow (x \in t[0; i] \wedge i \leq \text{size})$ (3)

Prouvons que $\text{tr}' \Leftrightarrow (x \in t[0; i'] \wedge i' \leq \text{size})$.

 - Par (2), il faut prouver que $\text{tr}' \Leftrightarrow (x \in t[0; i + 1] \wedge i + 1 \leq \text{size})$.
 - Donc que $\text{tr}' \Leftrightarrow (x \in t[0; i] \wedge i < \text{size})$.
 - Par condition de la boucle, on a $i < \text{size}$. (4)

- Par (4) il reste à prouver $tr' \Leftrightarrow x \in t[0; i]$.
- Examinons le cas $t[i] = x$.
 - Dans ce cas, par (1), on a : $tr' = vrai \wedge x \in t[0; i]$. (5)
 - CQFD par (5) (puisque $(A \wedge B) \Rightarrow (A \Leftrightarrow B)$).
- Examinons le cas $t[i] \neq x$.
 - Par (1), on a $tr' = tr$. (6)
 - Par (3), (6), on a $tr' \Leftrightarrow x \in t[0; i]$ et donc $x \in t[0; i]$. (7)
 - Par (7), soit tr' est vrai, alors $x \in t[0; i]$ et donc $x \in t[0; i]$.
 - Par (7), soit tr' est faux, alors $x \notin t[0; i]$ et comme $t[i] \neq x$, on a $x \notin t[0; i]$ \square

Nous avons démontré que I est bien un invariant de boucle. Revenons maintenant à notre propriété de correction P . Nous devons la déduire de I . Nous savons qu'à la fin de la boucle, $tr = true \vee i \geq size$ (1) puisque la condition de boucle est fausse. Examinons les deux valeurs possibles de tr .

- Cas $tr = faux$. Par (1), on a $i \geq size$. Puisque i est incrémenté à chaque itération, nous en déduisons que $i = size$. Donc par I , $x \notin t[0; size]$, c'est-à-dire que x n'apparaît pas dans toute « l'étendue » du tableau.
- Cas $tr = vrai$. Par I on a $x \in t[0; i \wedge i \leq size]$, donc $x \in t[0; size]$.

Si l'on ne rentre pas dans la boucle, alors $size \leq 0$, donc le tableau est vide et $tr = faux$. \square

6.6 Conclusion

Plutôt qu'une présentation des concepts théoriques sous-tendant les différentes techniques de preuves, ce chapitre est volontairement orienté vers la présentation de deux méthodologies de preuves : l'une pour les programmes écrits dans le style *fonctionnel* (i.e. sans boucles ni affectations), l'autre pour ceux écrits dans le style *impératif* (avec séquences, boucles et affectations).

De nombreuses autres techniques de preuves existent, de nombreuses logiques existent, que nous avons volontairement passées sous silence, ce cours n'étant ni un cours de logique ni un cours de preuves formelles. Cette section se veut uniquement une ouverture vers les aspects théoriques afférents la programmation.

D'ailleurs, nous n'avons même pas clairement défini le cadre logique que nous avons utilisé, qui s'apparente à la logique du premier ordre (aussi appelée calcul des prédictats).

Les travaux théoriques menés sur la logique et la preuve de programmes depuis des *décennies* ont abouti à de très nombreux outils de preuve automatique ainsi qu'à des outils d'aide à la preuve. Le théorème de Rice indique que la vérification *automatique* de propriétés arbitraires sur des programmes arbitraires est *indécidable*. De ce fait, soit l'on choisit d'utiliser un outil de preuve automatique qui pourra échouer à démontrer une propriété pourtant démontrable, soit l'on choisit de travailler plus, en faisant cette démonstration soi-même, mais aidé par un outil qui simplifiera la méthodologie et certaines étapes de preuve. De tels outils ont la propriété d'être basés sur des résultats mathématiques solides qui leur permettent de vérifier les preuves qu'ils produisent ou celles rédigées par l'utilisateur. C'est donc une forte garantie de correction, obtenue parfois (souvent ?) au prix d'un travail minutieux et conséquent de la part de l'utilisateur (dans le cas d'un outil d'aide à la preuve).

Pour ne citer que quelques uns de ces nombreux outils, mentionnons les assistants de preuve :

- Coq : <https://coq.inria.fr>
- ISABELLE : <https://isabelle.in.tum.de>
- PVS : <https://pvs.csl.sri.com>
- WHY3 (dans un style différent) : <http://why3.lri.fr>

et les prouveurs automatiques (qui ne traitent pas forcément les mêmes classes de problèmes, mais qui peuvent être utilisés par certains assistants de preuve) :

- Z3 : <https://github.com/Z3Prover>
- VAMPIRE : <http://www.vprover.org>
- ZENON : <http://zenon.inria.fr>

- ALT-ERGO : <http://alt-ergo.lri.fr>