INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

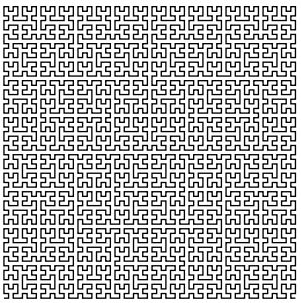
STPI 2A 2024–2025

ALGORITHMIQUE ET PROGRAMMATION 3

Cours et exercices

Hugo Raguet

Sommaire



Introduction

« L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des téléscopes. »

Michael R. Fellows and Ian Parberry— [1]

La conception populaire tend à réduire l'informatique aux outils qui en sont issus : logiciels de bureautique et gestion, réseaux et télécommunication, machines de calculs spécifiques. Mais l'informatique est aussi une discipline scientifique aux implications plus larges. Elle est issue des travaux des mathématiciens Alan TURING et Alonzo CHURCH, qui à partir des années 1930, cherchent à définir formellement *ce qui est caculable*. Le concept théorique de *machine de* TURING, sorte d'ordinateur idéalisé qui serait composé d'une bande sur laquelle un automate écrit des symboles selon des règles prescrites, naît à une époque où les ordinateurs n'existent pas encore. L'idée s'impose qu'une fonction mathématique doit être définie comme un *algorithme*, c'est-à-dire un ensemble de règles de calculs.

Le concept d'algorithme n'est pourtant pas du siècle dernier. Le terme lui-même provient de la translittération latine du nom d'un mathématicien perse du IX^e^ siècle, AL-KHWÂRIZMÎ. Celui-ci n'est cependant pas l'inventeur du concept : au III^e^ siècle avant notre ère, les *Éléments* d'EUCLIDE décrivent en géométrie des méthodes assimilables à des algorithmes, et surtout en arithmétique le célèbre *algorithme d'*EUCLIDE. Bien avant encore, le papyrus Rhind daté du XVII^e^ siècle avant notre ère atteste de la *méthode de multiplication égyptienne*. [2]

Ce cours vise dans un premier temps à introduire aux élèves ingénieurs le formalisme qui permet d'étudier les algorithmes : *l'algorithmique*. Comme on le devine en lisant les paragraphes précédents, ce formalisme repose pour une large part sur les mathématiques, langage universel des sciences. Nous y ajouterons au chapitre 2 un langage adapté à la manipulation des algorithmes, que nous utiliserons tout au long de ce cours.

Il n'est pas dans notre propos de rentrer dans des considération abstraites sur la calculabilité. En revanche, on ne peut se contenter de décrire ce qui est calculable, encore faut-il savoir *comment le calculer* et *à quel prix*. Ce sera en particulier l'objet du chapitre 3.

En pratique, un algorithme s'applique à des *données* selon la façon dont celles-ci sont organisées et dont on y accède : ce qu'on appelle la *structure de données*. Les deux notions sont complémentaires et intrinsèquement liées, à tel point qu'un des ouvrages majeurs de l'informatique des années 1970 s'intitule « *Algorithmes + Structures de données = Programmes* ». [3] Cette notion sera abordée et illustrée aux chapitres 4 et 5.

Le chapitre 6 est transverse, exposant une approche algorithmique qui suggère à la fois des méthodes d'analyse particulières et des structures de données adaptées.

Tout au long du cours, nous nous efforcerons de traduire ces notions en terme de programmation en langage C.

À la suite de ce cours, les élèves ingénieurs devront être capables de :

— décrire des algorithmes dans un langage algorithmique abstrait,

— analyser leur fonctionnement et leur complexité, à l'aide d'outils mathématiques fondamentaux, en particulier le raisonnement par récurrence,

— expliquer les usages, avantages et inconvénients de structures de données simples,

— traduire ces connaissances en programmes informatiques efficaces.

1. "Computer science is no more about computers than astronomy is about telescopes." *Computing Research News*, 5(1):7, 1993.
2. Nous reviendrons sur ces algorithmes aux chapitres 2 et 6.
3. Niklaus WIRTH, *Algorithms + Data Structures = Programs*, 1976.

Comment utiliser ce manuel

Ce manuel est rédigé avec un triple objectif : être complet, c'est à dire contenir toutes les notions abordées dans ce cours, être autosuffisant, c'est-à-dire scientifiquement cohérent en reposant sur le minimum de références extérieures, et enfin être concis, c'est-à-dire n'exposer que ce qui est strictement nécessaire aux deux premiers objectifs.

Il est donc dense et abrupt en première lecture, conçu pour accompagner des explications en cours magistral. Après une séance de cours, il faut prévoir de lire la partie correspondante au plus tard le lendemain ; et s'efforcer de reconnaitre ce qui a été dit et ce qui n'est pas encore compris.

Seule la résolution d'exercices et problèmes peut permettre de comprendre à la fois l'intérêt des notions abordées et les formulations choisies pour les décrire. De plus, la résolution guidée d'un exercice est inutile si elle n'est pas accompagnée d'un effort personnel de réflexion, qui permet au minimum de reconnaitre ses propres lacunes dans la connaissances du cours, mais aussi et surtout d'assimiler durablement les techniques d'analyse.

En conséquence, il est indispensable de tenter de résoudre certains exercices *avant* les séances de travaux dirigés. Ce n'est qu'en revenant sur les notes de cours à la lueur de ces tentatives qu'apparaissent plus clairement les notions les plus fondamentales.

Le nombre et la variété d'exercices et problèmes proposés sont suffisants pour satisfaire même les étudiants les plus passionnés. Les *exercices* sont représentatifs de ce qui doit être su ; parmi lesquels le symbole ☆ repère les plus fondamentaux, et le symbole ★ repère les plus difficiles. Les *problèmes* sont plus ouverts et généralement aussi plus difficiles.

Remerciements

Une partie substantielle de ce manuel est inspirée de l'ancien cours de classe préparatoire aux grandes écoles de Jean-Pierre BERCISPAHIC [4].

Merci aux auteurs des illustrations agrémentant les différents chapitres.

© Jin WICKED, *The universal Turing machine*, 2002.
https://jinwicked.com

4. encore disponible à cette adresse : http://www.info-llg.fr/

1 Programmation et modularité en langage C

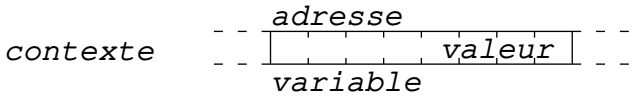
Dans nos cours de programmation, nous utilisons le langage C car son caractère fondamental permet de mieux comprendre le fonctionnement d'un ordinateur, et en conséquence d'écrire des programmes performants. Développé à l'origine dans les années 1970 par Dennis RITCHIE, il est aujourd'hui encore l'un des plus répandus, à tel point qu'on peut dire qu'il « constitue l'ADN de tous les logiciels modernes ». [1]

Nous ne revenons pas sur les notions de *fichier informatique*, *fichier exécutable*, *système d'exploitation* et *d'arborescence* des fichiers et dossiers. Aussi, les étudiants sont censés déjà savoir *naviguer en ligne de commande*, *éditer* un code source en langage C, et le *compiler*.

1 Rappels de programmation en langage C

Cette section rappelle quelques éléments fondamentaux, notamment sur la représentation et le gestion de la mémoire, et l'appel de fonctions.

Pour représenter l'état de la mémoire de l'ordinateur, nous utiliserons le schéma suivant



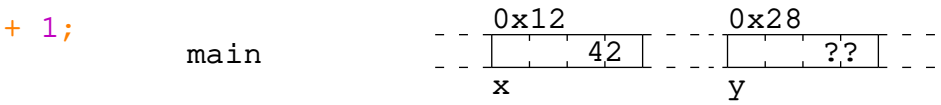
où
adresse est une adresse mémoire (arbitraire), écrite par convention en hexadécimal,
valeur est la valeur encodée par les octets à partir de l'adresse considérée,
variable est le nom d'une variable qui permet de manipuler cette valeur, et enfin
contexte désigne la *portée* où cette *variable* est visible.

1.1 Fonctions, paramètres, arguments

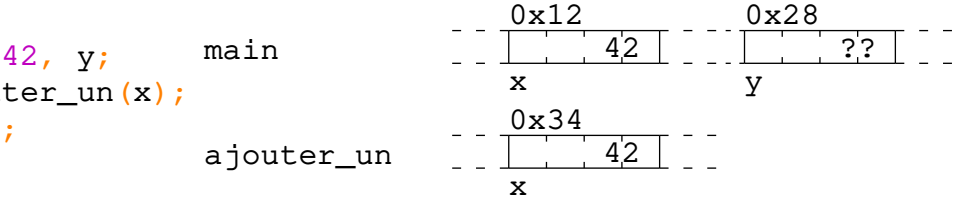
En langage C, les routines sont appelées des fonctions.
Les variables permettant de manipuler les entrées sont les *paramètres*.
Lors de l'appel, ces paramètres sont initialisés avec la valeur de l'*argument* correspondant.
Quand la fonction termine, le résultat est écrit dans l'unique *argument de sortie* s'il existe.

```
1  int ajouter_un(int x)
2  {
3      return x + 1;
4  }
5
6  int main()
7  {
8      int x = 42, y;
9      y = ajouter_un(x);
10     return 0;
11 }
```

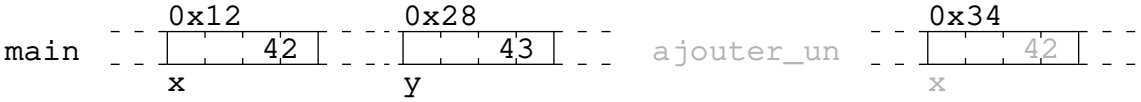
Le programme commence à la ligne 7. Après la ligne 8, la situation est la suivante.



À la ligne 9, `ajouter_un` est appelée. Situation à la ligne 2 :



À la ligne 3, le résultat de la fonction `ajouter_un` est écrit dans sa sortie, c'est-à-dire la variable `y` de la fonction `main`, adresse `0x28` sur le schéma, puis le contexte de la fonction `ajouter_un` est supprimé. Situation après la ligne 9 :



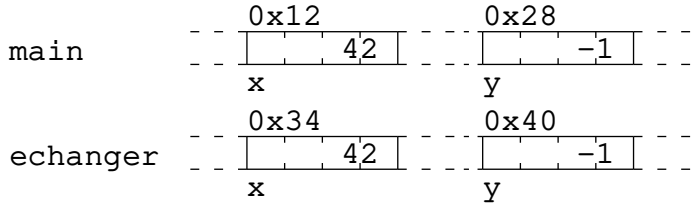
1. Jason Perlow, « Without Dennis Ritchie, there would be no Steve Jobs ». *ZDNet.com*, 9 octobre 2015. http://www.zdnet.com/article/without-dennis-ritchie-there-would-be-no-jobs/

1.2 Passage par adresse

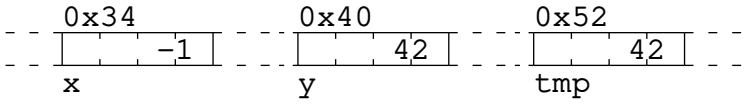
Considérons le programme suivant.

```
1  void echanger(int x, int y)
2  {
3      int tmp = x;
4      x = y;
5      y = tmp;
6  }
7
8  int main()
9  {
10     int x = 42, y = -1;
11     echanger(x, y);
12     return 0;
13 }
```

Il commence à la ligne 9. À la ligne 11, `echanger` est appelée. Situation à la ligne 2 :



À la fin de la ligne 5, le contexte de la fonction `echanger` est le suivant:

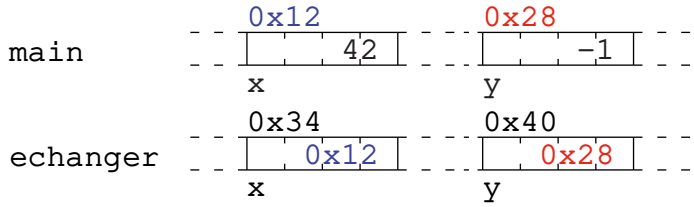


À la fin de la ligne 11 le contexte de la fonction `echanger` est détruit, et l'appel n'a servit à rien : les variables `x` et `y` de la fonction `main` n'ont pas changé.

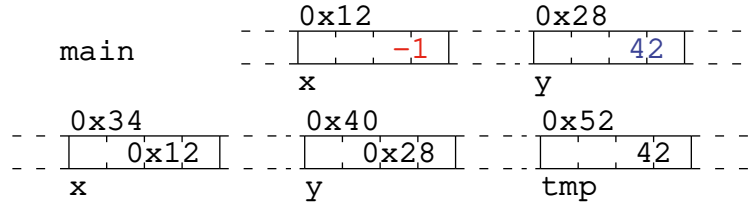
Pour modifier des variables visibles dans une autre portée, on se sert de leur adresse comme suit.

```
1  int echanger(int* x, int* y)
2  {
3      int tmp = *x;
4      *x = *y;
5      *y = tmp;
6  }
7
8  int main()
9  {
10     int x = 42, y = -1;
11     echanger(&x, &y);
12     return 0;
13 }
```

Cette fois-ci, la situation à la ligne 2 est la suivante.



À la fin de la ligne 5, la situation devient :



Les variables qui contiennent des adresses s'appellent des *pointeurs*.
À la ligne 1, l'étoile `*` sert à *déclarer* des pointeurs.
À la ligne 4, le même caractère sert à accéder en lecture et en écriture aux emplacements mémoires aux adresses que contiennent les pointeurs ; on appelle cela *déréférencer*.
Enfin, à la ligne 11, l'esperluette `&` sert à récupérer l'adresse de l'emplacement mémoire correspondant à une variable ; c'est l'opérateur de *référencement*.

1.3 Tableaux statiques

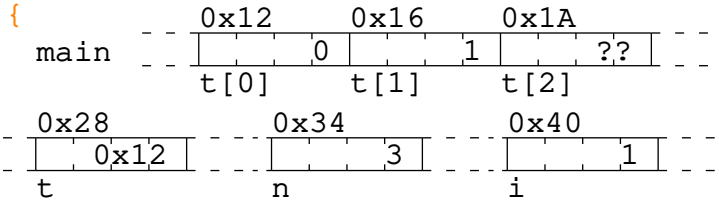
On peut créer des tableaux dits *statiques*, c'est à dire dont la taille est connue au moment de la compilation, avec les crochets `[]`.

Un tableau se comporte comme si on avait déclaré plusieurs variables du même type les unes à la suite des autres, avec la garantie que les emplacements mémoires sont contigus.

Une particularité importante du langage C est que quand on passe un tableau en argument d'une fonction, le paramètre correspondant est en fait un pointeur, qui récupère l'adresse mémoire de la première case du tableau. On dit que le tableau *se réduit* à un pointeur.

```
1  #define N_STAT 3
2  void remplir(int* t, int n)
3  {
4      int i;
5      for (i = 0; i < n; i++){
6          t[i] = i;
7      }
8  }
9
10 int main()
11 {
12     int t[N_STAT];
13     remplir(t, N_STAT);
14     return 0;
15 }
```

À la ligne 12, un tableau statique est déclaré. À la ligne 13, la fonction `remplir` est appelée. Après la deuxième itération de la ligne 6, la situation est la suivante.



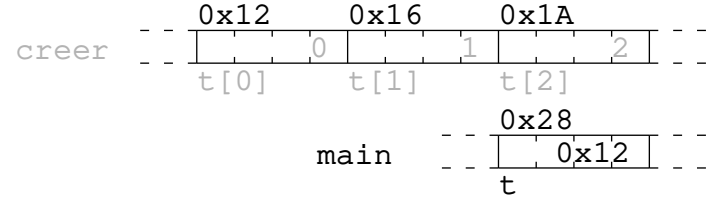
Observons qu'à la ligne 12, les crochets `[]` servent à déclarer le tableau, tandis qu'à la ligne 6, ils servent d'opérateur de déréférencement, selon l'arithmétique des pointeurs : `t[i]` est équivalent à `*(t + i)`, où `t + i` calcule l'adresse contenue par `t` augmentée de `i` fois la taille du type référencé (sur le schéma, on a supposé que `sizeof(int)` vaut 4).

1.4 Allocation de mémoire dynamique

Examinons le cas suivant, qui reprend la définition de `remplir`, mais tente de déclarer le tableau dans une fonction `creer`.

```
10 int* creer()
11 {
12     int t[N_STAT];
13     remplir(t, N_STAT);
14     return t;
15 }
16
17 int main()
18 {
19     int* t = creer();
20     return 0;
21 }
```

À la ligne 19, la fonction `creer` est appelée, mais quand elle termine, son contexte est supprimé. La situation est alors la suivante.



La variable `t` dans la fonction `main` contient une adresse qui n'est plus protégée, rien ne garantit que cette adresse ne sera pas utilisée pour autre chose.

Les variables locales statiques existent dans une zone mémoire qu'on appelle la *pile* (cf. la pile d'exécution, la § 2.2 du chapitre 6) ; il ne faut jamais passer en sortie d'une fonction l'adresse d'une telle variable.

Pour réserver un emplacement mémoire persistent, on utilise une autre zone mémoire, qu'on appelle le *tas*, à l'aide de la fonction `void* malloc(size_t size)` de la *bibliothèque standard* de C, accessible par le fichier d'*en-tête* `stdlib.h`.

Le paramètre `size` est un entier spécifiant le nombre d'octets que l'on souhaite ; ce mécanisme d'*allocation* de mémoire permet en particulier de créer des tableaux dits *dynamiques*, c'est-à-dire dont la taille n'est pas connue au moment de la compilation. [2]

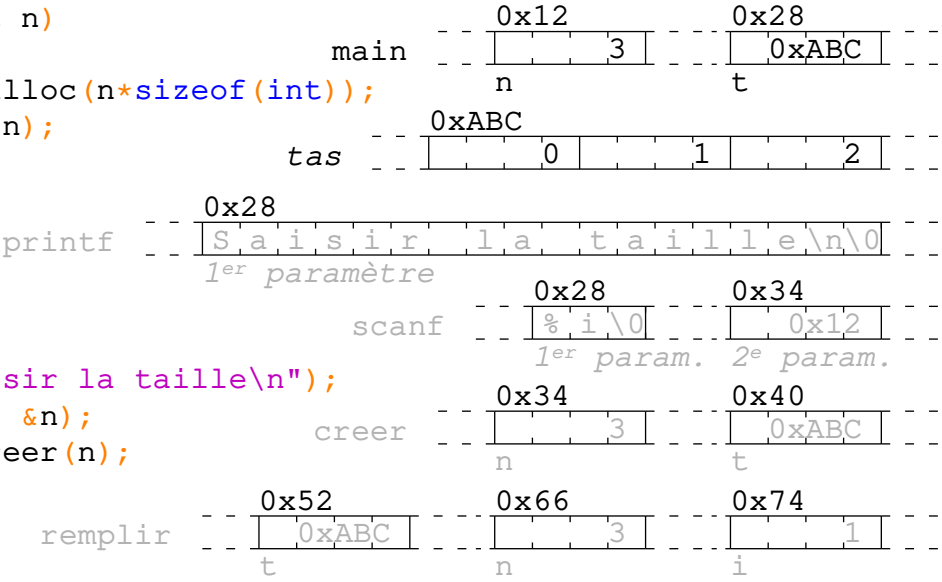
Cette fonction passe en sortie l'adresse mémoire du premier octet ainsi alloué. Tant qu'on a besoin de l'emplacement mémoire, il ne faut jamais perdre cette adresse ; quand on n'en a plus besoin, il faut penser à libérer la mémoire avec la fonction `void free(void* ptr)`.

2. Attention, les tableaux dits *dynamiques* en langage C sont néanmoins des structures de données dites *statiques* au sens algorithmique, cf. la § 1.1 du chapitre 4.

```c
#include <stdio.h>
#include <stdlib.h>
int* creer(int n)
{
    int* t = malloc(n*sizeof(int));
    remplir(t, n);
    return t;
}

int main()
{
    int n;
    printf("Saisir la taille\n");
    scanf("%i", &n);
    int* t = creer(n);
    free(t);
    return 0;
}
```

En supposant que 3 est saisi à la ligne 23, schématisons l'état de la mémoire après la ligne 24.



On peut noter à l'occasion des lignes 22 et 23 que les chaînes de caractères littérales sont des tableaux statiques se terminant par le caractère nul `'\0'`. Pour augmenter la lisibilité du schéma, nous nous sommes permis une simplification : strictement parlant, ces tableaux statiques n'existent pas tels quels dans les fonction `printf` et `scanf`, ils sont dans un emplacement mémoire temporaire et passés par adresse comme expliqué plus haut.

Soulignons enfin que les adresses sur le schéma sont arbitraires, mais cohérentes : on a représenté le fait que quand une fonction termine, la mémoire occupée par son contexte est libérée, et peut donc être réutilisée.

1.5 Structures

Les structures servent à rassembler plusieurs données ensemble, un peu comme le fait un tableau statique, mais avec des types de données potentiellement différents.

Avant de pouvoir créer une variable d'une telle structure, il faut préciser les types de données qu'on veut y mettre : on doit la *définir*.

```c
struct Nombre_rationnel
{
    int numer, denom;
    double approx;
};
```

Cette définition n'est pas une déclaration de variable, mais seulement la création d'un nouveau type abstrait `struct Nombre_rationnel`.

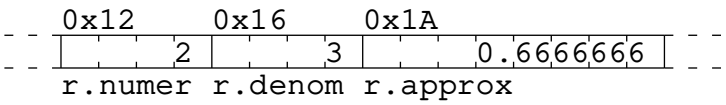
On peut utiliser `typedef` pour créer un alias,

```c
typedef struct Nombre_rationnel Nombre_rationnel;
```

et manipuler le type `Nombre_rationnel` comme n'importe quel autre type C.

Les composantes sont appelés *champs nommés* : on y accède avec un point `.` avec le nom qu'on leur a donné lors de la définition de la structure.

```c
int main()
{
    Nombre_rationnel r;
    r.numer = 2;
    r.denom = 3;
    r.approx = 2.0/3;
    return 0;
}
```



On manipule souvent les structures avec des pointeurs, pour deux raisons.
Premièrement, pour modifier une structure passée à une fonction ; il faut alors utiliser le passage par adresse rappelé en § 1.2. On parle d'un accès à la structure en *lecture-écriture*.

```
void mettre_a_deux_tiers(Nombre_rationnel* r)
{
    r->numer = 2; r->denom = 3; r->approx = 2.0/3;
}
```

Deuxièmement, pour éviter de copier une structure passée à une fonction ; car une structure peut avoir une très grande taille, on préfèrera passer un pointeur qui contient seulement son adresse. On parle d'un accès en *lecture-seule*. Pour distinguer ce cas du premier et garantir qu'une structure ne sera pas modifiée, on utilise le mot-clé `const`.

```
void afficher_nombre_rationnel(const Nombre_rationnel* r)
{
    printf("%i/%i ~= %f", r->numer, r->denom, r->approx);
}
```

Dans les deux cas, on remarquera la syntaxe spécifique avec la flèche `->`, qui permet à la fois de déréférencer le pointeur et d'accéder à un champ : `r->numer` est équivalent à `(*r).numer`, mais plus lisible.

2 Modularité en langage C

On parle de *modularité* quand un système est composé de nombreux composants élémentaires, qu'on peut facilement agencer, modifier, retirer sans avoir à modifier les autres composants. On qualifie de tels composants de *modules*.

2.1 Intérêt et principes

En informatique, qui est l'art de construire des algorithmes complexes à partir d'instructions simples, c'est fondamental. La modularité permet en particulier trois choses.
Premièrement, elle permet de réutiliser une même fonctionnalité dans divers programmes. C'est le rôle des *bibliothèques logicielles*, par exemple la bibliothèque standard du langage C : pas besoin de réécrire systématiquement `stdio` (et heureusement).
Deuxièmement, elle permet d'éviter d'avoir à recompiler toutes les composantes d'un gros programme dès qu'on fait une modification quelque part : seules les parties concernées seront recompilées.
Troisièmement, elle permet de mieux organiser un projet, en identifiant des rôles bien spécifiques à chaque module, et permettant de travailler sur chacun d'entre eux séparément.

En langage C, on écrit dans des codes sources séparés les fonctionnalités propres à chaque *module*.

Chaque module doit pouvoir être compilé *séparément* en un fichier en langage machine, qu'on appelle un fichier *objet*. Un fichier objet n'est pas encore un programme exécutable.

Un programme en langage C doit contenir une fonction `int main()` qui est l'entrée du programme. Il peut aussi être compilé séparément, mais s'il dépend de fonctionnalités définies dans d'autres fichiers objets, le compilateur doit pouvoir les lier ensemble pour en faire un programme exécutable. Cette étape s'appelle *l'édition de liens*.

Pour que le code source d'un module puisse faire appel à des fonctionnalités d'autres modules, tout en pouvant être compilé séparément, il doit disposer de quelques informations élémentaires sur ces fonctionnalités.

Dans ce but, on utilise un *fichier d'en-tête* : il contient les informations que doit partager un module avec tous les autres modules qui utilisent ses fonctionnalités. Ces informations sont incorporées par la *directive de préprocesseur* `#include` *`fichier`*, où le nom du fichier est entre guillemets droits `""` pour les fichiers locaux et entre chevrons <> pour les fichiers système. Par convention, l'extension des fichiers d'en-tête est `.h`, de l'anglais *header*.

2.2 Illustration rudimentaire

Imaginons un ensemble de logiciels qui visent à manipuler des objets géométriques dans le plan. Dans ce contexte, un composant de base serait le *point*. D'autres composants construits à partir du point pourrait être le *segment*, défini par deux points (les extrémités), et le *cercle*, défini par un point (le centre) et un rayon.

D'un côté, un programme qui souhaite manipuler des segments ne souhaite pas nécessairement manipuler des cercles, ou réciproquement. Il est donc pratique de séparer ces deux composants.

D'un autre côté, puisque ces deux composants nécessitent de manipuler des points, il est pratique de réutiliser pour cela le même composant de base. Le fichier 1.1 propose l'en-tête pour le module point.

```
#ifndef POINT_H
#define POINT_H
typedef float reel_t; /* type représentant les nombres réels */
/* un point est défini par deux coordonnées réelles */
struct Point { reel_t x, y; };
typedef struct Point Point;
void afficher_point(const Point*); /* fonction d'affichage */
#endif
```

FICHIER 1.1 – En-tête `point.h`.

Il définit d'abord le type de base `reel_t` ; tous les codes sources qui incluent cette en-tête y auront accès, garantissant la cohérence du type choisi pour représenter des nombres réels. Si d'aventure le type `float` ne s'avérait pas satisfaisant, il suffirait de le changer ici, par exemple en `double`, pour que le changement se répercute dans tous les codes sources qui en dépendent.

Ensuite, il définit le type composite `Point` avec deux coordonnées.

Enfin, il *déclare* une fonction permettant d'afficher un tel point, *sans la définir*. En effet, un module qui utilise cette fonction n'a pas besoin de connaitre la définition pour être compilé en un fichier objet ; il a seulement besoin de connaître sa *signature*, c'est-à-dire son nom, le type de chacun de ses paramètres, et le type de sa valeur de retour.

Les fichiers 1.2 et 1.3 proposent ensuite les en-têtes pour les modules segment et cercle, et le fichier 1.4 propose un programme qui les utilise tous les deux.

```
#include "point.h"
/* un segment est défini par deux points */
struct Segment { Point a, b; };
typedef struct Segment Segment;
void afficher_segment(const Segment*); /* fonction d'affichage */
```

FICHIER 1.2 – En-tête `segment.h`.

```
#include "point.h"
/* un cercle est défini par un centre et un rayon */
struct Cercle { Point centre; reel_t rayon; };
typedef struct Cercle Cercle;
void afficher_cercle(const Cercle*); /* fonction d'affichage */
```

FICHIER 1.3 – En-tête `cercle.h`.

Le programme comprend par exemple l'instruction `segment.a.y = 0.0`. Pour compiler, le programme doit connaitre l'existence du type `Segment`, ainsi que sa définition pour pouvoir manipuler le champ `a`. De même, ce champ étant définit comme un point et puisqu'on accède à son tour à son champ `y`, la définition du type `Point` doit être disponible.

```c
#include <stdio.h>
#include "segment.h"
#include "cercle.h"
int main()
{
    Segment segment;
    segment.a.x = segment.a.y = 0.0;
    segment.b.x = segment.b.y = 1.0;
    Cercle cercle;
    cercle.centre.x = cercle.centre.y = 0.5;
    cercle.rayon = 2.0;
    printf("Segment : "); afficher_segment(&segment); printf("\n");
    printf("Cercle : "); afficher_cercle(&cercle); printf("\n");
    return 0;
}
```

FICHIER 1.4 – Programme `dessin.c`.

La directive `#include` agit *comme si on la remplaçait par le contenu du fichier inclus*. Puisque l'en-tête `segment.h` inclut à son tour l'en-tête `point.h`, tout se passe comme si cette dernière était écrite mot pour mot dans le code source `dessin.c`.

2.3 Compilation modulaire en langage C

Puisque `cercle.h` inclut aussi `point.h`, le contenu de ce dernier se retrouve en fait deux fois dans le code source `dessin.c`. Cela pose problème, car il est interdit de déclarer plusieurs fois la même structure ou la même fonction en langage C.

Pour éviter ce problème, on a pris soin de rajouter des *gardes-fous* : les deux premières et la dernière lignes de `point.h`, fichier 1.1.

Ce sont des directives de préprocesseur *conditionnelles* : tout le code source compris entre `#ifndef POINT_H` et `#endif` n'est pris en compte par le compilateur qu'à la condition que la macro `POINT_H` ne soit pas définie. Or, la directive `#define POINT_H` définit précisément cette macro ; ainsi, l'en-tête `point.h` ne peut jamais être prise en compte plus d'une fois par le compilateur au sein d'un module.

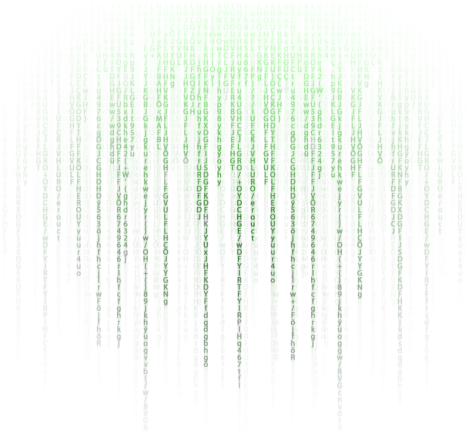
Il faut en règle générale mettre des garde-fous à chaque fichier d'en-tête.

Le processus qui traduit un ensemble de codes sources en un fichier exécutable, qu'on appelle compilation par abus de langage, est donc en fait décomposé en trois grandes étapes. Tout d'abord, le *préprocesseur* applique au code source toutes les directives (`#include`, `#if`, `#define`, etc.).
Ensuite, chaque module est *compilé séparément* en un fichier objet.
Enfin, *l'édition de lien* identifie quel module définit chaque fonctionnalité du programme principal, et agence le langage machine de chaque fichier objet en un fichier exécutable.

Pour terminer, mentionnons deux aspects importants de la modularité.
Premièrement, on peut fournir à l'éditeur de lien n'importe quel fichier objet compatible, sans avoir besoin de son code source pour le compiler. C'est ce qu'il se passe quand on utilise les fonctionnalités de `stdio.h` : l'éditeur de lien utilise la bibliothèque C standard installée sur le système d'exploitation. Il existe différents types de bibliothèques en C, en particulier statiques (code machine intégré dans le programme compilé) et dynamiques (bibliothèque chargée lors de l'exécution). La plupart du temps, il faut mentionner au compilateur où se trouvent les bibliothèques impliquées sur le système d'exploitation.
Les travaux pratiques, ainsi que le projet d'informatique de fin d'année (pré-orientation GSI) seront l'occasion de voir des exemples, en particulier via l'utilisation de bibliothèques graphiques.

Deuxièmement, comme mentionné en § 2.1, la compilation de gros logiciels peut être coûteuse, et il est inutile de recompiler tous les modules impliqués lorsque seulement l'un d'entre eux a été modifié. Il existe pour cela des utilitaires de compilation qui identifient automatiquement quel module doit être recompilé.
À notre niveau, nous ne compilons que des programmes simples et ce genre d'optimisation n'est pas utile pour notre usage. Nous nous contenterons de recompiler toutes les sources à chaque fois ; pour le cas considéré dans notre illustration, avec la commande suivante

```
gcc dessin.c point.c segment.c cercle.c -o <exécutable>
```



modifié depuis https://pixabay.com/

Exercices et problèmes

Exercice 01. Écrire les fichiers `point.c`, `segment.c` et `cercle.c` qui vont avec les entêtes données par les fichiers 1.1–1.3, afin qu'après compilation et exécution du programme donné par le fichier 1.4, la console affiche le texte suivant.

```
Segment : [(0.0, 0.0), (1.0, 1.0)]
Cercle : (---2.0---(0.5, 0.5)---2.0---)
```

Dans les exercices suivants, il est demandé de mettre en œuvre en langage C les algorithmes et les structures de données abordés aux chapitres 2 et 4–6. Il est indispensable de tester les solutions proposées ; pour cela, il faut toujours les accompagner de fonctions d'affichages permettant de vérifier sur des exemples simples qu'on obtient bien le résultat attendu.

Exercice 02. ☆ Traduire en langage C les solutions proposées pour les exercices 06, 07 et 09–0B au chapitre 2.

Exercice 03. ☆ On considère les listes chaînées, décrites au chapitre 4.

(a) Mettre en œuvre des listes chaînées en langage C selon les solutions proposées pour l'exercice 24.

(b) Traduire en langage C les solutions proposées pour les exercices 1D–21.

Exercice 04. ☆ On considère les piles et les files de capacité limitée, décrites au chapitre 5.

(a) Mettre en œuvre de telles structures avec des tableaux en langage C, en suivant les descriptions données en §§ 2.1 et 2.3.

(b) Traduire en langage C les solutions proposées pour les exercices 25–2A.

Exercice 05. ★ On considère le tri par fusion décrit au chapitre 6.

(a) Mettre en œuvre le tri par fusion sur des tableaux en langage C en suivant la description donnée en § 3.

(b) Reprendre la mise en œuvre des listes chaînées de l'exercice 03, et traduire en langage C les solutions proposées pour l'exercice 32.

Problème 01. ★ Mettre en œuvre en langage C les solutions proposées pour les problèmes 02 et 03. On écrira au minimum la structure `Variable`, et les fonctions suivantes, qui simulent les opérations élémentaires correspondantes dans notre pseudo-langage.

```c
typedef struct Variable Variable;
void definition_affectation(Variable*, Variable*);
void definition_reference(Variable*, Variable*);
void affectation_reference(Variable*, Variable*);
```

[ ]
* &

2+ Algorithmique et pseudo-langage

1 Rappels sur le pseudo-langage

Rappelons que le cours de première année introduit la notion de pseudo-langage, illustré avec la méthode de multiplication égyptienne, évoquée au chapitre introductif. Elle permet de multiplier rapidement des nombres entiers en utilisant seulement la table de 2 et des additions, reposant sur la simple propriété suivante : $m \times n = \begin{cases} (m/2) \times (2n) & \text{si } m \text{ est pair,} \\ (m-1) \times n + n & \text{sinon.} \end{cases}$

L'intuition est qu'on peut se ramener à un problème « plus simple » avec un nombre m plus petit, tout en prenant soin de faire avancer le calcul ; dans le premier cas, en multipliant n par 2, et dans le second cas, en ajoutant m au résultat.

Le pseudo-langage nous permet de traduire cette intuition en une suite d'instructions univoques, mais sans les contraintes d'un langage de programmation, puisque la description s'adresse à un humain et non un ordinateur.

```
Algorithme multiplication_égyptienne : (m, n) ⟶ p selon
   p ⟵ 0
   Tant que m ≠ 0 répéter
      Si m \ 2 = 0 alors m ⟵ m / 2, n ⟵ n × 2
      sinon p ⟵ p + n, m ⟵ m − 1 .
   .
.
```

Soulignons qu'un pseudo-langage ne dit rien sur les mécanismes qui permettraient de représenter concrètement les objets et les opérations concernés. Il est d'autant plus intéressant que nous programmons par ailleurs en langage C, qui est dit « bas niveau » d'abstraction ou encore « proche de la machine » : il exige une gestion manuelle de la mémoire. À chaque concept introduit, il est intéressant de se demander dans quelle mesure et comment le langage C permet de le mettre en œuvre ; voir en particulier les exercices 0C–10 et problèmes 02 et 03.

Bien au contraire du langage C, de nombreux langages de programmation tentent d'abstraire au maximum la syntaxe et la gestion de la mémoire. C'est le cas en particulier des langages interprétés comme Python (généraliste et modulaire), GNU Octave (spécialisé dans le calcul matriciel), ou R (statistiques). Notre pseudo-langage est donc aussi l'occasion d'appréhender l'utilisation de tels langages dits « haut niveau » d'abstraction.

Nous étendons dans la suite le pseudo-langage de première année, dont nous listons ici les concepts déjà introduits.
Mécanisme de création et modification de *variables non typées* `nom ⟵ valeur`,
opérateurs arithmétiques `+ − × ÷ / \`,
valeurs booléennes vrai faux et *opérateurs logiques* **`ou et non`**,
opérateurs de comparaison `= ≠` et de *relation d'ordre* `< ⩽ > ⩾`,
création de tableaux `nom ⟵ [nombre]`, ainsi que l'accès au *nombre d'éléments* **`nélém(`***`nom`***`)`** et *indexation* `nom(indice)` avec $1 \leqslant$ *indice* $\leqslant$ **`nélém(`***`nom`***`)`**,
branchements conditionnels **`Si`** *condition 1* **`alors`** *corps 1*
`sinon si` *condition 2* **`alors`** *corps 2* **`sinon`** *corps alternatif* **`.`**,
boucles conditionnelles **`Tant que`** *condition* **`répéter`** corps **`.`**,
boucles énumérées **`Pour`** *compt* **`parcourant (`***`prem, ..., dern`***`) répéter`** *corps* **`.`**,
et enfin *algorithmes* **`Algorithme`** *nom* **`:`** *entrées* ⟶ *sorties* **`selon`** *corps* **`.`**
et *appels de routine* `résultat ⟵ nom(arguments)`.

2 Exceptions

Les *exceptions* servent à signaler des situations spécifiques qui ne peuvent être gérées par l'algorithme ; par exemple si l'on passe une valeur pour laquelle l'algorithme n'a pas de sens. Dans notre pseudo-langage, un tel cas particulier peut être mis en évidence à l'aide de l'instruction **`exception(`**"description"**`)`**, où " description" décrit succinctement la situation rencontrée.

```
Algorithme division_euclidienne : (a, b) → (q, r) selon
   Si b = 0 alors exception(“division par zéro”) .
   q ← a / b, r ← a \ b
.
```

L'intention n'est pas de lister dans l'algorithme toutes les valeurs qui n'ont pas de sens (par exemple dans l'algorithme ci-dessus, faire une exception pour des nombres non entiers), mais de mettre en évidence des situations particulières avec des entrées autorisées (zéro est bien un nombre entier).

3 Structures de données en pseudo-langage

Une *structure de données* est une réunion de données, avec potentiellement des relations particulières entre elles. Pour comprendre le lien étroit entre algorithmes et structures de données, Il nous faut un mécanisme explicite pour les manipuler.

3.1 Construction composite

Nous définissons les *constructions composites* comme des *réunions ordonnées*. En principes, on peut donc directement utiliser des tableaux. Par exemple pour représenter un point dans un espace en deux dimensions avec des coordonnées données `x` et `y`

```
P ← [2], P(1) ← x, P(2) ← y
```

Cependant, cette syntaxe ne met pas en évidence la réunions des données. Nous lui préférons la syntaxe usuelle en mathématiques pour définir des n-uplets, où les *éléments constitutifs* sont groupés avec des parenthèses et séparés par des virgules **`(`***`expression 1`***`, ..., `***`expression N`***`)`** ; l'exemple ci-dessus peut alors s'écrire simplement

```
P ← (x, y)
```

Une construction composite peut bien sûr être elle-même un élément constitutif d'une construction composite ; par exemple un point pondéré **`((`**`x, y`**`), p)`** ou un système de points pondérés **`(((`**`x1, y1`**`), p1), ((`**`x2, y2`**`), p2), ((`**`x3, y3`**`), p3))`**, etc.

Il pourra aussi être utile de considérer des structures vides, qu'on représentera avec le symbole $\emptyset$.

3.2 Affectations composites

On peut utiliser la construction composite pour faire des *affectations composites*. Par exemple pour affecter une construction composite à une unique variable

```
S ← (((3,14, 1,62), 1,0), ((0,69, 2,72), 2,0), ((0,56, 1,41), 3,0))
```

À l'inverse, on peut décomposer une construction composite en de multiples variables ; pour que la syntaxe soit valide, on doit pouvoir affecter sans ambiguïté les variables et constructions composites de droite aux variables et constructions composites de gauche.

```
(((x1, y1), p1), ((x2, y2), p2), ((x3, y3), p3)) ← S
```

Dans notre pseudo-langage *c'est ainsi que l'on accède aux élément d'une construction composite*. Ce mécanisme est très différent de la syntaxe équivalente en langage C, où les structures utilisent des *champs nommés*, voir l'exercice 0E.

En théorie, on peut mélanger les deux.

```
(P1, (M2, p2), ((x3, y3), p3)) ← (((x1, y1), p1), ((x2, y2), p2), P3)
```

La syntaxe est valide si `P3` est compatible avec un point pondéré `((x3, y3), p3)`.[1] En pratique, les constructions que nous utilisons ne dépassent jamais quelques éléments constitutifs.

3.3 Modification par références

Jusqu'à présent, nous manipulons les variables par valeurs ; par exemple l'instruction `j ← i` récupère la valeur de `i` dans la variable `j` ; par la suite l'instruction `j ← 3,14` change la valeur de `j` mais la valeur de `i` n'a pas changé.

Pour exprimer des modifications sur des structures de données complexes, on veut pouvoir exprimer qu'on modifie seulement certains de ses éléments constitutifs. Pour cela, notre pseudo-langage utilise alors un autre genre de variable : des *références*.

En sus, ce mécanisme servira aussi à exprimer le fait qu'une routine a vocation à modifier ses entrées.

Définition de références

On crée une référence et lui affecte une *variable référencée* avec la *définition de référence* : *nom_réf* ← **réf** *nom_var*, où *nom_var* est une autre variable.
Si la variable *nom_var* est déjà une référence, alors *les deux variables partageront la même variable référencée*.[2]
Quand une référence apparait dans une expression à évaluer, en particulier à droite d'une affectation, elle prend la valeur de sa variable référencée.
Pour modifier une variable par le biais d'une référence, *l'affectation par référence* utilise la même syntaxe que l'affectation simple : *nom* ← *valeur*.
Si l'on veut changer la variable référencée par une référence, il suffit de faire une autre définition de référence.

La suite d'instructions suivantes illustre ces règles sur des variables simples.

```
i ← 1, j ← 42
r ← réf i  # r est une référence vers i
r ← −1     # i prend la valeur −1
s ← réf r  # s est une référence vers i
r ← réf j  # r est maintenant une référence vers j
s ← r      # la valeur de i est maintenant égale à 42
```

Références composites

On peut créer des références dans des constructions composites.

```
P ← ((réf x, réf y), p) # le premier élément de P contient deux références
```

Si on décompose une variable composite par référence, on obtient des références sur ses éléments constitutifs.

```
((s, t), q) ← réf P # subtilité : s et t réfèrent à x et y
q ← −1,0            # modifie le deuxième élément de P (mais pas p)
```

Dans notre pseudo-langage *c'est ainsi que l'on modifie la valeur d'un élément d'une structure composite*.

Remarque. Les indexations des tableaux se comportent en fait comme des références composites : `tab(i)` est une référence sur l'élément constitutif numéro `i` de `tab` ; comme si on avait créé les références suivantes : `(tab(1), ..., tab(nélém(tab))) ← réf tab`.

1. Vérifier qu'une affectation multiple est valide n'est pas trivial dans le cas général, voir le problème 12 au chapitre 5.
2. On ne peut donc pas faire de référence sur une référence, subtilité à laquelle nous serons peu confrontés.

Passage par référence

Rappelons que la modification de la valeur d'une variable d'entrée dans une routine n'est pas visible en dehors de la routine ; on dit qu'une telle entrée est disponible en *lecture seule*.

Si l'on veut pouvoir modifier une variable dans une routine en utilisant le mécanisme des références décrit ci-dessus, il faudrait passer une référence lors de l'appel, suivant la construction *résultat* ← *nom*`(`**réf** *arguments*`)`

Il est cependant plus pertinent de spécifier le *passage d'arguments par référence* dans la définition de la routine, selon la construction suivante :

Algorithme *nom* **: réf** *entrées* → *sorties* **selon** *corps* **.**

Cela indique que la routine a vocation à modifier une variable d'entrée ; on dit qu'une telle entrée est disponible en *lecture écriture*. Il devient inutile de préciser la référence lors de l'appel.

Une routine peut se contenter de modifier les variables passées en entrée, sans calculer de sortie ; ces routines sont parfois appelée des *procédures*. On spécifie alors la sortie vide `()`.

```
Algorithme remplir_tableau : (réf tab, x) → () selon
   # rappel : les indexations se comportent comme des références
   Pour i parcourant (1, ..., nélém(tab)) répéter tab(i) ← x .
.
```

L'appel se fait alors sans valeur de sortie : `remplir_tableau(tab, 42)`.

On peut aussi spécifier un *passage en sortie par référence*, avec la construction suivante :

Algorithme *nom* **: réf** *entrées* → **réf** *sorties* **selon** *corps* **.**

Il est cependant exigé de préciser aussi lors de l'appel de la routine si l'on veut récupérer une référence, sans quoi le passage en sortie par référence est ignoré.

Une telle routine prend généralement aussi ses entrées par référence, pour que les références de sortie puissent référencer les variables d'entrée. Dans ce cas de figure, on pourra utiliser *l'affectation par référence en sortie* pour se passer de variable intermédiaire.

```
Algorithme abscisse : réf p → réf x selon
   (x, y) ← réf p # définition de référence composite
.

p ← (1,62, 2,72)      # encore un point du plan
x ← abscisse(p)       # x vaut 1,62
x ← 1,41              # p n'est pas modifiée
s ← réf abscisse(p) # s est une référence sur l'abscisse de p
s ← 3,14              # on modifie l'abscisse de p
abscisse(p) ← 3,14    # instruction équivalente à la précédente
                      # utilisant l'affectation par référence en sortie
```

© The Trustees of the British Museum — Papyrus Rhind

Exercices et problèmes

Algorithmique

Exercice 06. ☆ Écrire un algorithme `factorielle` qui accepte en entrée un nombre entier naturel et qui calcule sa factorielle.

Exercice 07 (Algorithme d'Euclide). ★ Écrire un algorithme `pgcd` qui accepte en entrée deux nombres entiers naturels, et qui calcule leur *plus grand diviseur commun*.
Indication: le plus grand diviseur commun de a et b est le même que le plus grand diviseur commun de b et du reste de la division euclidienne de a par b.

Dans les exercices 08–0A, on considère des valeurs que l'on peut comparer avec l'opérateur de relation d'ordre $\leqslant$.

Exercice 08. Écrire un algorithme `tri_plet` qui accepte en entrée un triplet de valeurs, et qui les ordonne par ordre croissant. On fera deux versions : l'une qui crée un nouveau triplet ordonné par ordre croissant, l'autre qui modifie le triplet donné en entrée.

Exercice 09 (Tri par sélection). ☆

(a) On considère un tableau et un entier pouvant indexer le tableau. Écrire un algorithme `sélectionner_minimum`, qui cherche l'élément minimum entre l'indice *inclus* et la fin du tableau, et qui échange cet élément minimum avec l'élément de l'indice.

(b) Écrire un algorithme `tri_sélection` qui accepte en entrée un tableau et qui le trie par ordre croissant à l'aide de la routine `sélectionner_minimum`.

Exercice 0A (Tri par insertion). ☆

(a) On considère un tableau et un entier pouvant indexer le tableau. On suppose que le tableau est ordonné par ordre croissant jusqu'à l'indice *exclu*. Écrire un algorithme `insérer_indice`, qui ordonne le tableau par ordre croissant jusqu'à l'indice *inclus*.

(b) Écrire un algorithme `tri_insertion` qui accepte en entrée un tableau et qui le trie par ordre croissant à l'aide de la routine `insérer_indice`.

Exercice 0B (Méthode de Newton). ★ La méthode de NEWTON est un algorithme générique pour trouver numériquement une approximation d'un zéro (ou racine) d'une fonction : $f : \mathbb{R} \to \mathbb{R}$, on cherche $x \in \mathbb{R}$ tel que $f(x) \approx 0$. Étant donné un réel positif x_0, la linéarisation de f autour de x_0 s'écrit $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$; ce qui suggère $f(x_1) \approx 0$ avec $x_1 = x_0 - f(x_0)/f'(x_0)$. Si l'approximation n'est pas satisfaisante, on réitère le processus.

(a) Proposer un critère pour évaluer si l'approximation est satisfaisante.

(b) Écrire un algorithme `inverse`, qui accepte en entrée un nombre réel et un critère d'approximation, et calcule une approximation de l'inverse du nombre donné, en n'utilisant que des additions et des multiplications.

(c) Écrire un algorithme `racine_carrée`, qui accepte en entrée un nombre réel positif et un critère d'approximation, et calcule une approximation de la racine carrée du nombre donné, en n'utilisant que des additions et des multiplications.

Il s'agit de la méthode de HÉRON, *connue depuis l'antiquité.*

(d) Écrire un algorithme `sécante`, qui accepte en entrée une fonction représentée par une routine et un critère d'approximation, et qui tente de trouver un zéro de la fonction avec la méthode de *Newton* où la dérivée de la fonction est estimée par différences finies. Décomposer l'algorithme avec le plus de routines intermédiaires possibles.

Cette variante est appelée méthode de la sécante.

Pseudo-langage et programmation

Exercice 0C. Notre pseudo-langage manipule les nombres entiers relatifs et les nombres réels. Rappeler la façon dont les nombres sont représentés sur un ordinateur, et les écueils à éviter.

Exercice 0D. ☆ Notre pseudo-langage définit un mécanisme de références avec le mot-clé **`réf`**; cf. § 3.3. Par quel moyen ce genre de mécanisme peut être mis en œuvre en langage C? Quels sont les points communs et les différences avec notre pseudo-langage?

Exercice 0E. ☆ Notre pseudo-langage manipule des structures de données comme des réunions de données dans des constructions composites; cf. § 3. Que prévoit le langage C pour créer des types composites? Comparer les syntaxes pour accéder et manipuler les éléments des types complexes dans chacun des langages.

Exercice 0F. ☆ Notre pseudo-langage traite les tableaux comme des constructions composites que l'on peut indexer; cf. § 3.1. Comparer avec la manipulation de tableaux en C.
Indication : penser en particulier aux types, à la copie, au passage en argument de fonction.

Exercice 10. L'opérateur de comparaison `=` peut s'appliquer aux structures composites comme suit : *deux constructions composites sont égales si, et seulement si, ils ont le même nombre d'éléments et leurs éléments sont égaux deux à deux*. Les éléments d'une structure composite pouvant être eux-mêmes des structures composites, leur comparaison fait de nouveau appel à cette définition : on dit que c'est une définition *récursive*.
Écrire un algorithme `est_égal` qui accepte en entrée deux variables simples ou composites, et teste leur égalité de valeur. Dans cet exercice on traitera les structures comme si c'était des tableaux.

Problème 02. ★★ Les variables de notre pseudo-langage ne sont pas typées : on peut affecter à la même variable des valeurs de différents types. Décrire une structure `Variable` en langage C qui permette ce mécanisme.

Problème 03. ★★★ Notre pseudo-langage permet de manipuler des constructions composites et des tableaux; cf. § 3.

(a) À partir des exercice 0F et problème 02, améliorer la structure `Variable` pour mettre en œuvre en C les constructions composites du pseudo-langage. Considérer dans un premier temps que la création d'une variable composite nécessite la copie de toutes les données impliquées.

(b) Proposer une mise en œuvre évitant la duplication de toutes les données.

Si l'on ne recopie pas les données lors de la création d'une variable composite, c'est que plusieurs variables partagent les mêmes données physiques.

```
a ← 1, b ← (2, a), a ← 3 # comment s'assurer que b n'est pas modifiée ?
b ← (4, b)               # comment définir b ?
c ← b, (r1, r2) ← réf b,  r2 ← 5 # comment s'assurer que c ne change pas ?
```

(c) Proposer encore une modification de la structure `Variable` et un mécanisme permettant de dupliquer les données seulement quand c'est nécessaire, ainsi qu'un mécanisme compatible pour mettre en œuvre les références.

Enfin, il arrive que des données ne soient plus accessibles ni par des variables, ni par des références; c'est le cas par exemple lorsqu'une routine se termine et que certains objets ne sont pas passée en sortie. La mémoire occupée par ces objets devient inutilisable : on appelle cela une *fuite mémoire*, pouvant aller jusqu'à épuiser la mémoire disponible et provoquer un arrêt brutal du système.

(d) Expliquer comment le mécanisme du point (c) peut être exploité pour identifier les objets que l'on peut supprimer de la mémoire.

3 Analyse et complexité

Quand on veut mettre en œuvre un algorithme, il faut s'assurer qu'il se termine en un temps fini, c'est-à-dire après un nombre fini d'opérations élémentaires, et qu'il effectue bien le calcul voulu. On appelle ces propriétés essentielles la *terminaison* et la *correction*.

De plus, en pratique on souhaite s'assurer qu'on peut le mettre en œuvre avec des ressources raisonnables, en particulier avoir un ordre de grandeur du nombre d'opérations et d'informations nécessaires ; on appelle cela le *coût* ou la *complexité* de l'algorithme.

1 Terminaison et correction

Si l'algorithme n'est qu'une suite linéaire d'instructions, la terminaison est évidente, et la correction est le résultat d'une suite d'égalités mathématiques mises bout à bout.

Cependant, en présence de structures de contrôle itératives (les boucles ou, comme nous le verrons au chapitre 6, les appels récursifs), la preuve peut s'avérer plus délicate. Remarquons que les valeurs successives des variables d'une algorithme contenant une structure itérative définissent en fait une *suite*.

Considérons l'algorithme de la multiplication égyptienne du chapitre 2.

```
Algorithme multiplication_égyptienne : (m, n) → p selon
   p ← 0
   Tant que m ≠ 0 répéter
      Si m \ 2 = 0 alors m ← m / 2, n ← n × 2
      sinon p ← p + n, m ← m − 1 .
   .
.
```

Les valeurs successives des variables `m`, `n` et `p` définissent des suites par récurrence : $(m_i)_{i\in\mathbb{N}}$, $(n_i)_{i\in\mathbb{N}}$ et $(p_i)_{i\in\mathbb{N}}$, où m_0 et n_0 sont initialisées avec les entrées, $p_0 = 0$, et pour tout $i \in \mathbb{N}$,

$$\begin{cases} m_{i+1} = m_i/2, & n_{i+1} = 2n_i & \text{et } p_{i+1} = p_i & \text{si } m_i \neq 0 \text{ et } m_i \text{ est pair,} \\ m_{i+1} = m_i - 1, & n_{i+1} = n_i & \text{et } p_{i+1} = p_i + n_i & \text{si } m_i \neq 0 \text{ et } m_i \text{ est impair,} \\ m_{i+1} = m_i, & n_{i+1} = n_i & \text{et } p_{i+1} = p_i & \text{si } m_i = 0 \text{ (arrêt de l'algorithme).} \end{cases}$$

1.1 Invariant de boucle

Pour pouvoir prouver des résultats sur un nombre quelconque d'itérations, le principe fondamental est bien sûr le *principe de récurrence*.

Théorème (Principe de récurrence). Soit $\mathcal{P}$ un *prédicat* sur $\mathbb{N}$. Si $\mathcal{P}_0$ et $\forall n \in \mathbb{N}$, $\mathcal{P}_n \implies \mathcal{P}_{n+1}$, alors $\forall n \in \mathbb{N}$, $\mathcal{P}_n$.

Remarque. L'implication $\mathcal{P}_n \implies \mathcal{P}_{n+1}$ peut être remplacée par $(\forall i \in \{1, \dots, n\}, \mathcal{P}_i) \implies \mathcal{P}_{n+1}$, permettant d'utiliser une *hypothèse de récurrence* plus générale.
Aussi, c'est le principe de récurrence qui assure qu'une *suite récurrente* dans un ensemble E est bien définie par $u_0 \in E$ et $\forall n \in \mathbb{N}, u_{n+1} = f_n(u_n)$, dès que $\forall n \in \mathbb{N}, f_n : E \to E$ (ou plus généralement, $u_{n+1} = f_n(u_0, \dots, u_n)$ avec $f_n : E^n \to E$).

Pour étudier une boucle, on utilise le principe de récurrence en cherchant un prédicat qui reste vrai d'une itération à l'autre. On appelle cela un *invariant de boucle*.

Pour étudier la correction de la multiplication égyptienne, posons l'invariant de boucle suivant. Pour $i \in \mathbb{N}$, $\mathcal{P}_i : m_i n_i + p_i = m_0 n_0$.
Initialisation : $p_0 = 0$, donc $\mathcal{P}_0$.

Hérédité : soit $i \in \mathbb{N}$, on suppose $\mathcal{P}_i$. Si $m_i = 0$, alors immédiatement $m_{i+1}n_{i+1} + p_{i+1} = m_i n_i + p_i$. Sinon, si m_i est pair, alors $m_{i+1}n_{i+1} + p_{i+1} = (m_i/2)(2n_i) + p_i = m_i n_i + p_i$. Si m_i est impair, $m_{i+1}n_{i+1} + p_{i+1} = (m_i - 1)n_i + p_i + n_i = m_i n_i + p_i$. Dans tous les cas, $\mathcal{P}_i$ implique $\mathcal{P}_{i+1}$. D'après le principe de récurrence, pour tout $i \in \mathbb{N}$, $\mathcal{P}_i$: il s'agit bien d'un invariant de boucle. Or, si l'algorithme termine, c'est qu'à la dernière itération $i \in \mathbb{N}$, $m_i = 0$. D'après $\mathcal{P}_i$, on a alors $p_i = m_0 n_0$; p vaut bien le produit des entrées, ce qui prouve que l'algorithme est correct.

Cependant, le fait que la boucle s'arrête reste à prouver.

1.2 Convergent

Pour prouver que l'algorithme termine, on peut considérer un autre invariant de boucle. Posons pour $i \in \mathbb{N}$, $\mathcal{P}_i$: *m_i est un entier naturel vérifiant $m_i = 0$ ou $1 \leqslant m_i \leqslant m_0 - i$.*
Initialisation : $\mathcal{P}_0$ est évident dès que les entrées sont des entiers naturels.
Hérédité : soit $i \in \mathbb{N}$, on suppose $\mathcal{P}_i$. Si $m_i = 0$, alors $m_{i+1} = 0$. Sinon, si m_i est pair, alors $m_{i+1} = m_i/2$ est un entier, et $m_i \geqslant 2$ donc $1 \leqslant m_{i+1} = m_i - m_i/2 \leqslant m_i - 1$. Si m_i est impair, alors $m_{i+1} = m_i - 1$. Dans tous les cas, $\mathcal{P}_i$ implique $\mathcal{P}_{i+1}$.

D'après le principe de récurrence, pour tout $i \in \mathbb{N}$, $\mathcal{P}_i$. En particulier, $\mathcal{P}_{m_0}$ assure que $m_{m_0} = 0$: le nombre d'itérations ne peut dépasser la valeur de la première entrée. Chaque itération impliquant un nombre fini d'opérations, l'algorithme termine.

Dans l'argument ci-dessus, nous montrons l'existence d'une quantité (la variable `m`) qui décroit vers une valeur critique (0) qui termine l'algorithme. On appelle une telle quantité un *convergent*, qui permet de conclure dès qu'elle vérifie les propriétés suivante :

- elle appartient à un ensemble E *bien fondé*, c'est-à-dire dont tout sous-ensemble admet un *élément minimal*, ici $E = \mathbb{N}$;
- la boucle termine dès qu'elle atteint un *ensemble d'arrêt* $A \subset E$, ici $A = \{0\}$;
- en dehors de cet ensemble, elle décroit *strictement* à chaque itération, ici d'au moins 1.

Remarque. Cette définition est générale et peut s'appliquer à une *relation d'ordre partiel*. Considérer par exemple l'ordre produit sur $\mathbb{N}^2$: $(x_1, y_1) \leqslant (x_2, y_2) \iff x_1 \leqslant x_2$ et $y_1 \leqslant y_2$; $(0, 1)$ et $(1, 0)$ ne sont pas comparables. Dans ce cas, un élément minimal m d'un ensemble $F \subset E$ est un élément qui n'admet pas d'élément strictement plus petit : $\forall x \in F$, $x \leqslant m \implies x = m$.

2 Complexité

2.1 Coût spatial, coût temporel, opérations élémentaires

On sait maintenant que l'algorithme de multiplication égyptienne termine ; mais quand, et à quel prix ? Quand il s'agit de mettre en œuvre un algorithme, on considère généralement deux types de ressources : le temps que cela prend ou *coût temporel*, et la quantité de mémoire nécessaire ou *coût spatial*.

Faire une évaluation précise de ces quantités dépend de nombreux facteurs matériels ou logiciels qui sortent du domaine de l'algorithmique. Nous ne pouvons donc attribuer une valeur absolue ni à la quantité de mémoire requise ni au temps d'exécution d'un algorithme donné.

Cependant, il est possible d'évaluer l'*ordre de grandeur* ; cela permet d'avoir une idée de la faisabilité d'un problème, ou bien d'identifier l'algorithme le plus efficace au sein d'un ensemble d'algorithmes résolvant le même problème.

Dans un premier temps, *il faut identifier les unités de mesures qui sont pertinentes au regard du problème posé*. En général, pour avoir une idée du coût temporel, on compte le nombre d'opérations élémentaires ; encore faut-il s'entendre sur ce terme.

Typiquement, dans un algorithme de calcul numérique, on compte les opérations arithmétiques ; quant au coût spatial, il est associé au nombre de variables intermédiaires qu'on

doit garder en mémoire. Dans un algorithme de tri, on compte les comparaisons entre les éléments et les accès en lecture ou écriture ; le coût spatial est cette fois associé au besoin de copier tout ou une partie du tableau.

Pour donner un exemple plus concret, revenons sur l'algorithme de multiplication égyptienne. Si l'on suppose que la multiplication de deux entiers naturels quelconques ne compte que pour une seule opération élémentaire, alors cet algorithme est inutile !

Si en revanche on suppose qu'on ne sait calculer que les multiplications et divisions euclidiennes par 2, ou qu'une multiplication par une autre quantité coûte plus cher, [1] alors la question se pose de savoir quel algorithme appliquer.

À chaque itération, l'algorithme de multiplication égyptienne effectue deux comparaisons, une division euclidienne par 2, puis soit une multiplication par 2 suivie d'une autre division euclidienne par 2, soit une addition et une soustraction ; en tout, quatre opérations élémentaires. De plus, la preuve de terminaison nous montre que le nombre d'itérations ne peut dépasser la valeur m de la première entrée, soit un total de, au plus, $4m$ opérations élémentaires.

Considérons alors une dernière alternative :

```
Algorithme : multiplication_belge (m, n) ⟶ p selon
   p ⟵ 0, Tant que m ≠ 0 répéter p ⟵ p + n, m ⟵ m − 1 .
.
```

À chaque itération, cet algorithme effectue une comparaison, une addition, et une soustraction, soit trois opérations élémentaires. De plus, il est très facile d'établir qu'il effectue exactement m itérations, où m est la valeur de la première entrée.

Cet algorithme naïf nécessite en tout exactement $3m$ opérations élémentaires. À première vue, il semble plus efficace que la multiplication égyptienne. Mais pour ce dernier nous n'avons qu'une borne supérieure grossière du nombre d'itérations, où nous avons utilisé le fait que `m / 2` vaut au plus `m − 1` quand `m` est supérieur à 1. Ne peut-on pas faire mieux ?

2.2 Coût asymptotique

Dans les analyses ci-dessus, il apparait que l'important n'est pas tant le nombre d'opérations par itération, mais bien le nombre d'itérations.

Par exemple, passer de trois à quatre opérations par itération augmente la charge de calcul d'un petit facteur, mais ce facteur n'est pas très pertinent au regard du fait que les opérations élémentaires sont déjà délicates à comparer entres elles et dépendent de l'ordinateur qui les effectue.

En revanche, si m est mille fois plus grand, le nombre d'opérations sera beaucoup plus fortement impacté. Pour l'algorithme naïf présenté ci-dessus, il sera aussi mille fois plus grand. C'est cet ordre de grandeur qui nous intéresse, et c'est pourquoi nous nous intéressons au *coût (ou complexité) asymptotique* : ce qui se passe pour *les grandes tailles des données*.

Ce qu'on appelle la taille des données ici dépend encore une fois du contexte : pour les algorithmes de multiplication considérés ci-dessus, c'est la valeur de la première entrée qui est le facteur déterminant ; pour un algorithme de tri, c'est la taille du tableau, etc.

Remarque. Dans nos exemples, les coûts ne dépendent que d'un seul facture ; mais en toute généralité, il peuvent dépendre de plusieurs facteurs.

Notations de Landau

La taille des données peut toujours s'exprimer par des entiers. Si c_n est le coût de l'algorithme pour une taille de problème n, sa complexité est donc le comportement asymptotique de la suite $(c_n)_{n\in\mathbb{N}}$. Pour l'exprimer, on utilise les notations de Landau de la table 3.1.

1. ce qui est le cas pour un humain, et reste vrai pour certaines machines ; voir aussi l'exercice 19.

TABLE 3.1 – Notations de LANDAU pour la comparaison asymptotique des suites.

Signification	Notation	Définition mathématique
u est dominée par v	$u = \mathrm{O}(v)$	$\exists n_0 \in \mathbb{N}, \exists M > 0, \forall n > n_0,\ u_n \leqslant M v_n$
u domine v	$u = \Omega(v)$	$v = \mathrm{O}(u)$
u et v ont même ordre de grandeur	$u = \Theta(v)$	$u = \mathrm{O}(v)$ et $u = \Omega(v)$

Remarque. On utilise aussi $u_n = \mathrm{O}(v_n)$, $u_n = \Omega(v_n)$ ou $u_n = \Theta(v_n)$ pour expliciter la dépendance en l'indice de la suite. Notons aussi qu'il n'est pas nécessaire que la suite de comparaison soit définie pour tous les termes, puisque nous nous intéressons au comportement asymptotique : il suffit qu'elle soit définie à partir d'un certain rang.

La notation $c = \mathrm{O}(u)$ indique que, dans le pire des cas, la croissance du coût ne dépassera pas celle de u. Par exemple, nous avons montré que la complexité temporelle de l'algorithme de multiplication égyptienne est $c_m = \mathrm{O}(m)$.
La notation $c = \Omega(u)$ donne quant à elle une minoration du meilleur des cas. Cependant, son usage seul est limité, car il ne donne aucune certitude sur le pire des cas, ou même sur le cas moyen (voir plus loin).
La notation $c_n = \Theta(u_n)$ est plus pertinente quand le pire et le meilleur des cas coïncident : on sait que dans tous les cas, le coût est à peu près proportionnel à u_n. Par exemple, l'algorithme de multiplication naïve par additions successives est $c_m = \Theta(m)$.

Remarque. Généralement, on essaie d'avoir une estimation la plus favorable possible quand on domine la complexité avec $c = \mathrm{O}(u)$. Par exemple, si on sait que $c_n = \mathrm{O}(n)$, il n'est pas pertinent d'écrire $c_n = \mathrm{O}(n^2)$, même si c'est mathématiquement vrai. Ainsi, cette notation est souvent abusivement utilisée en sous-entendant qu'il existe des configurations des entrées pour lesquelles c_n se comporte effectivement comme u_n, et donc qu'on ne peut pas avoir de meilleures garanties. L'exercice 19 montre que ce n'est pas ce que nous avons fait ci-dessus pour l'algorithme de la multiplication égyptienne.

Remarque. Quand le pire et le meilleur des cas ne coïncident pas, on peut chercher à calculer la *complexité en moyenne*. Cependant, cela nécessite de connaître le coût et la probabilité associés à chaque configuration des entrées. Ce cadre est souvent compliqué, et nous ne le rencontrerons que rarement ; on peut voir cependant les exercice 1A et problème 06.

Complexité usuelles

Les complexités les plus fréquemment rencontrées sont décrites dans la table 3.2.

TABLE 3.2 – Complexités usuelles, et tailles indicatives de problèmes que les ordinateurs personnels récents peuvent résoudre en un temps imparti. On se base sur 10^{10} opérations par seconde ; le cas polynomial est donné pour $p = 3$, et le cas exponentiel pour $p = 2$.

Suite de référence	Qualificatif usuel	Tailles indicatives de problème résolubles en un temps T		
		$T = 1/10\,\mathrm{s}$	$T = 1\,\mathrm{min}$	$T = 1\,\mathrm{mois}$
1	constante	*(limité seulement par la mémoire)*		
$\log n$	logarithmique			
n	linéaire	1×10^9	6×10^{11}	3×10^{16}
$n \log n$	quasi-linéaire	4×10^7	2×10^{10}	5×10^{14}
n^2	quadratique	3×10^4	8×10^5	2×10^8
n^p, $p \geqslant 2$	polynomiale	1×10^3	8×10^3	3×10^5
p^n, $p > 1$	exponentielle	3×10^1	4×10^1	6×10^1

Exercices et problèmes

Correction, terminaison

Exercice 11. Prouver le principe de récurrence énoncé en § 1.

Exercice 12. Prouver la correction et la terminaison des solutions proposées pour les exercices 06, 07, 09 et 0A au chapitre 2.

Exercice 13. ☆ On représente un polynôme $P(X) = \sum_{i=0}^{n} a_i X^i$ avec un tableau `P`, où `P(i)` représente le coefficient d'indice `i` moins 1. Déterminer un invariant de boucle pour établir le rôle de l'algorithme ci-dessous :

```
Algorithme mystère1 : (P, x) → y selon
   y ← 0,0, i ← nélém(P)
   Tant que i ⩾ 1 répéter y ← x × y + P(i), i ← i − 1 .
.
```

Exercice 14. Prouver la terminaison de l'algorithme suivant et déterminer ce qu'il calcule.

```
Algorithme mystère2 : x → y selon
   y ← 0, s ← 0, d ← 1
   Tant que s < x répéter
      y ← y + 1
      s ← s + d
      d ← d + 2
   .
.
```

Problème 04. ★ Prouver la terminaison de l'algorithme suivant et déterminer ce qu'il calcule.

```
Algorithme mystère3 : x → y selon
   y ← 0, r ← 1, p ← 1
   Tant que x > 0 ou r > 0 répéter
      b ← x \ 2
      Si (r + b) \ 2 = 1 alors y ← y + p .
      r ← (r + b) / 2
      p ← 2 × p
      x ← x / 2
   .
.
```

Complexité

Exercice 15. Soit $(u_n)_{n\in\mathbb{N}}$, $(v_n)_{n\in\mathbb{N}}$, $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ des suites réelles strictement positives.

(a) Montrer que les complexités peuvent s'additionner et se multiplier, c'est-à-dire

$$u = \mathrm{O}(a) \text{ et } v = \mathrm{O}(b) \implies u + v = \mathrm{O}(a + b) \text{ et } uv = \mathrm{O}(ab) .$$

(b) Montrer que les complexités peuvent s'additionner asymptotiquement, c'est-à-dire

$$u = \mathrm{O}(a) \text{ et } \forall N \in \mathbb{N}, v_N = \sum_{n=0}^{N} u_n \text{ et } b_N = \sum_{n=0}^{N} a_n \implies v = \mathrm{O}(b) .$$

(c) Montrer que les complexités sont transitives, c'est-à-dire

$$u = \mathrm{O}(a) \text{ et } a = \mathrm{O}(b) \implies u = \mathrm{O}(b) .$$

(d) ☆ Montrer que les complexités peuvent se simplifier, c'est-à-dire

$$u = \mathrm{O}(a + b) \text{ et } a = \mathrm{O}(b) \implies u = \mathrm{O}(b) .$$

(e) Établir les résultats analogues aux points (a)–(d) pour les relations Ω, Θ.

(f) ☆ On suppose $\forall n \in \mathbb{N}$, $u_n = 2n\log(n+1) + 4n - 1$. Identifier, parmi les assertions suivantes, lesquelles sont vraies, et laquelle est la plus pertinente.
(i) $u_n = O(n)$; (ii) $u_n = \Omega(1)$; (iii) $u_n = \Theta(n)$;
(iv) $u_n = O(2n\log(n+1))$; (v) $u_n = \Omega(8n)$; (vi) $u_n = \Theta(n\log n)$.

Exercice 16. ☆ Déterminer la complexité de l'algorithme pour chacun des corps suivants.

```
Algorithme boucles1 : (m, n) → (i, j) selon
```

(a)
```
    i ← 1, j ← 1, Tant que i ⩽ m et j ⩽ n répéter
       i ← i+1, j ← j+1
    .
```

(b)
```
    i ← 1, j ← 1, Tant que i ⩽ m répéter
       Si j ⩽ n alors j ← j+1 sinon i ← i+1 .
    .
```

(c)
```
    i ← 1, j ← 1, Tant que i ⩽ m répéter
       Si j ⩽ n alors j ← j+1 sinon i ← i+1, j ← 1 .
    .
```

```
.
```

Exercice 17. ☆ Déterminer la complexité de l'algorithme pour chacun des corps suivants.

```
Algorithme boucles2 : n → (i, j) selon
```

(a)
```
    i ← 1, Tant que i ⩽ n répéter i ← i+1
       j ← 1, Tant que j ⩽ n répéter j ← j+1 .
    .
```

(b)
```
    i ← 1, Tant que i ⩽ n répéter i ← i+1
       j ← 1, Tant que j ⩽ i répéter j ← j+1 .
    .
```

(c)
```
    i ← 1, Tant que i ⩽ n répéter i ← i+1
       j ← 1, Tant que j × j ⩽ i répéter j ← j+1 .
    .
```

(d)
```
    i ← n, Tant que i ⩾ 1 répéter i ← i / 2 .
```

(e)
```
    i ← n, Tant que i ⩾ 1 répéter i ← i / 2
       j ← 1, Tant que j ⩽ i, j ← j+1 .
    .
```

```
.
```

Exercice 18. ☆ Étudier la complexité des solutions proposées pour les exercices 09 et 0A au chapitre 2.

Exercice 19. Dans l'algorithme de la multiplication égyptienne présenté en § 1.1, décrire le comportement de la variable `m` dans le meilleur et le pire des cas. En déduire un invariant de boucle et la complexité de temporelle de l'algorithme.

Exercice 1A. ☆ On considère la recherche d'un élément dans un tableau.

(a) Écrire un algorithme `rechercher_séquentiel` qui accepte en entrée un tableau et un élément quelconque et qui calcule l'indice de l'élément dans le tableau s'il s'y trouve, ou la taille du tableau plus 1 sinon.

(b) Étudier la complexité de l'algorithme, dans le meilleur et le pire des cas.

(c) On suppose qu'on utilise l'algorithme dans un contexte ou chaque valeur de sortie possible est équiprobable. Donner la complexité en moyenne.

Exercice 1B. ☆ Reprendre l'exercice 1A ; on suppose maintenant les éléments peuvent être comparés avec l'opérateur $\leqslant$, et que le tableau est *trié* par ordre croissant.

(a) Écrire un algorithme `rechercher_dichotomie` qui améliore `rechercher_séquentiel`.

(b) Étudier la complexité de l'algorithme, dans le meilleur et le pire des cas.

Problème 05. ★ Reprendre l'exercice 1A ; on suppose maintenant qu'on utilise l'algorithme de recherche séquentielle dans un contexte dans lequel :

— les éléments du tableau appartiennent à un ensemble E à n éléments, et

— chaque élément du tableau peut être chaque élément de E avec la même probabilité, et indépendamment des autres éléments du tableau.

Étudier la complexité en moyenne de l'algorithme de recherche séquentielle, en fonction de n et de la taille du tableau d'entrée.

Problème 06. ★ Étudier la complexité des solutions proposées pour les exercices 07 et 0B au chapitre 2.

Problème 07. ★★ *Les Shadoks* ont conçu une fusée pour aller sur Terre. Cette fusée supporte n angles de tirs différents, un seul convient ; le problème est qu'ils ne savent pas lequel. Si l'angle est trop grand, la fusée rate la Terre et est perdue dans l'espace ; si l'angle est trop petit, la fusée retombe sur la planète Shadok et peut être réutilisée pour un nouvel essai.

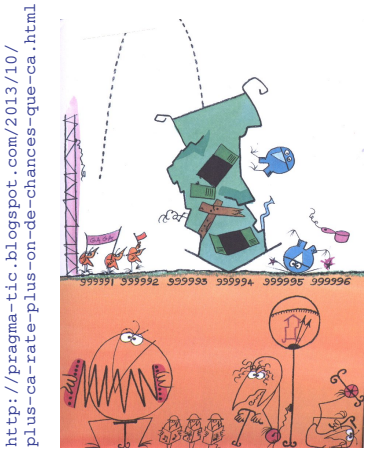
Le professeur Shadoko dispose de k fusées, et, contrairement à l'adage Shadok usuel, [1] souhaite déterminer le bon angle en faisant le minimum d'essais.

(a) Si $k = 1$, un seul algorithme raisonable permet de déterminer le bon angle à coup sûr. Quel est, dans le pire des cas, son coût en nombre d'essais ?

(b) Si $k \geqslant \lceil \log n \rceil$, proposer un algorithme nécessitant $O(\log n)$ essais.

(c) Si $k < \lceil \log n \rceil$, proposer un algorithme nécessitant $O\left(k + \frac{n}{2^{k-1}}\right)$ essais.

(d) Si $k = 2$, proposer un algorithme nécessitant $O\left(\sqrt{n}\right)$ essais.



1. *« Ce n'est qu'en essayant continuellement que l'on finit par réussir ; en d'autres termes : plus ça rate, et plus on a de chances que ça marche ! »*

4 Listes chaînées

1 Structures de données linéaires

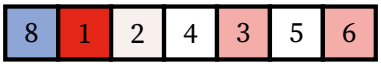
Une *structure de donnée* est caractérisée par la façon dont on *accède* aux objets qu'elle contient, et la façon dont on la modifie, typiquement en *ajoutant* ou en *supprimant* des objets.

Les *listes chaînées* sont, au même titre que les tableaux, des *structures de données linéaires*, c'est-à-dire représentable par des suites finies ordonnées.

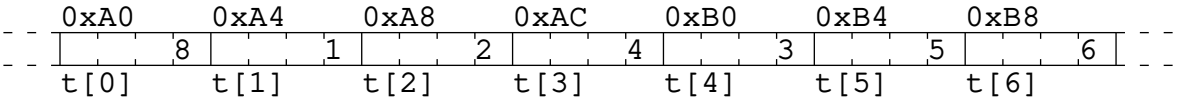
Bien que dans la partie algorithmique de ce cours nous omettons autant que possible les détails de la représentation des objets en mémoire, il nous faut nous pencher sur la question pour comprendre ce qui différencie les listes chaînées et les tableaux, et leurs avantages respectifs.

1.1 Retour sur les tableaux

Nous schématisons habituellement un tableau sous la forme suivante.



Pour être efficace, les éléments d'un tableau doivent être *contigus en mémoire*, selon la représentation plus détaillée suivante



où les entiers sont représentés sur 4 octets, et l'indexation emprunte la notation C.

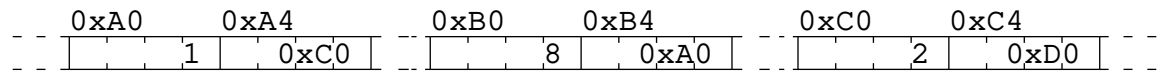
Cette propriété permet facilement d'accéder à n'importe quel élément du tableau dès qu'on connait l'adresse du premier élément (ici, `0xA0`), la taille de chaque élément (ici, 4 octets), et l'indice de l'élément recherché : il suffit d'un petit calcul arithmétique. Ainsi, l'accès à un élément du tableau s'effectue *en temps constant*, au sens où il ne dépend pas de la taille du tableau.

En revanche, si l'on veut rajouter un élément au tableau, c'est plus compliqué. En effet, une fois l'espace mémoire réservé (ici, $7 \times 4 = 28$ octets), d'autres opérations peuvent modifier la mémoire alentour, et les octets adjacents (ici, à partir de l'adresse `0xBC`) ne peuvent être présumés disponibles. Pour maintenir la propriété de contigüité, on se retrouve obligé de réserver un plus grand espace mémoire ailleurs, et d'y copier la totalité du tableau. Ainsi, l'ajout d'un élément a un coût, à la fois en temps et en espace, *linéaire en la taille du tableau*. De même, la suppression d'un élément nécessite de déplacer tous les éléments suivants, ce qui représente un coût temporel linéaire dans le pire des cas ; voir l'exercice 1C.

On qualifie une telle structure, dont la taille ne peut pas être facilement modifiée, de *statique*.[1]

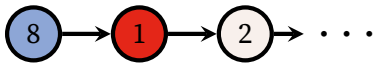
1.2 Une alternative : les listes chaînées

Pour être moins contraint sur les emplacement mémoire des éléments, une solution est de stocker à côté de chaque élément l'adresse mémoire de l'élément suivant :



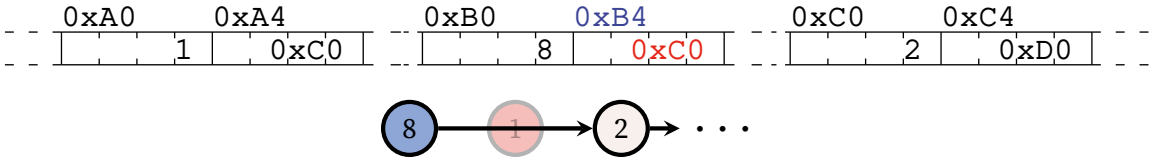
1. Attention, en langage C on parle de « tableau dynamique » pour désigner le mécanisme d'allocation de mémoire permettant de manipuler des tableaux dont la taille n'est pas connue à compilation. Il s'agit d'un sens différent du mot dynamique ; les tableaux dynamiques présentent exactement les inconvénients décrits en § 1.1, et restent à ce titre des structures de données statiques.

On parle alors de liste chaînée, qu'on représente plus schématiquement comme suit.

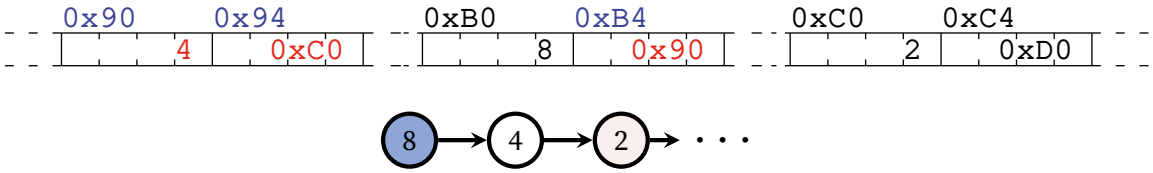


Comme pour les tableaux, il suffit de connaitre l'adresse du premier élément (ici, `0xB0`) pour accéder à n'importe quel autre élément. En revanche, au contraire des tableaux, on ne peut plus accéder à un élément arbitraire à l'aide d'une simple opération arithmétique : on est obligé de passer par chaque élément intermédiaire. Dans le pire des cas, on parcourt toute la liste pour accéder au dernier élément, la complexité temporelle est donc *linéaire en la taille de la liste*.

Cependant, la suppression d'un élément est facilitée : il suffit de faire pointer l'élément précédent sur l'élément suivant. Pour supprimer l'élément 1 dans l'illustration, on remplace, à l'adresse `0xB4`, `0xA0` par `0xC0`.



De même, on peut rajouter un élément en faisant pointer l'élément précédent sur le nouvel élément, et le nouvel élément sur l'élément suivant. Pour ajouter l'élément 4 entre 8 et 2 dans l'illustration, on crée l'élément à l'adresse arbitraire `0x90`, puis on remplace, à l'adresse `0xB4`, `0xC0` par `0x90`, et enfin on spécifie à l'adresse `0x94` l'information `0xC0`.



Dans les deux cas, on peut constater que l'ajout et la suppression s'effectue *en temps constant*, dès lors qu'on connait l'adresse de l'élément qui précède. On qualifie une telle structure, dont la taille peut être facilement modifiée, de *dynamique*.

TABLE 4.1 – Complexités temporelle des opérations élémentaires sur les tableaux et les listes chaînées en fonction de leur taille n. Pour l'ajout ou la suppression d'un élément sur une liste chaînée, on suppose ici qu'on a accès au prédécesseur.

Opération	Tableau	Liste chaînée
Accès au premier élément	$\Theta(1)$	$\Theta(1)$
Accès à un élément arbitraire	$\Theta(1)$	$O(n)$
Ajout ou suppression d'un élément	$O(n)$	$\Theta(1)$

Les tableaux et les listes chaînées ont donc chacun leurs avantages ; on peut aussi remarquer qu'une liste chaînée nécessite de stocker une adresse pour chaque élément, et donc est plus coûteuse en mémoire qu'un tableau ayant le même nombre d'éléments. Le choix de la structure de données dépend toujours étroitement de la situation.

Rajoutons que le principe présenté ici est très général, et les mises en œuvres diffèrent d'une situation à une autre. Il existe plusieurs variantes utiles, par exemple les listes *doublement chaînées*, où chaque cellule donne accès à la précédente et la suivante, ou les listes chaînées *circulaires*, où la dernière cellule donne accès vers la première.

2 Mise en œuvre des listes chaînées en pseudo-langage

On peut maintenant mettre de coté les aspects mémoires décrits précédemment pour se concentrer sur les opérations élémentaires qui caractérisent les listes chaînées, résumées dans la table 4.1.

2.1 Construction

L'information immédiatement disponible avec une liste chaînée est donc le premier élément, qu'on appellera la *tête*. Puis, il faut pouvoir accéder à l'élément suivant. On peut remarquer que celui-ci constitue lui aussi la tête d'une liste chaînée, qu'on appellera *la queue*.

Ainsi, une liste chaînée peut être représentée dans notre pseudo-langage par une construction composite `(tête, queue)`, où `tête` est un élément d'intérêt (les données proprement dites, des entiers sur les illustrations en § 1.2), et `queue` est *elle-même une liste chaînée*.

Remarque. Les constructions qui, comme les listes chaînées, interviennent dans leur propre définition, sont dites *récursives*. Nous reviendrons sur cet aspect important au chapitre 6.

Dans le contexte des listes chaînées, la construction composite `(tête, queue)` s'appelle une *cellule*.

Remarquons que la dernière cellule d'une liste chaînée n'a pas de queue. Dans notre pseudo-langage, la fin d'une liste chaînée est signalée par la valeur spéciale ∅, mentionnée en § 1 au chapitre 2, que nous appellerons dans ce contexte la *liste vide* ou *cellule vide*.

Création manuelle et décomposition d'une liste chaînée

```
1 l ← (8, (1, (2, ∅)))              # accumule vite les parenthèses
2 l ← (2, ∅), l ← (1, l), l ← (8, l) # souligne la structure récursive
3 (t, q) ← l  # t contient la tête 8, q contient la queue (1, (2, ∅))
```

À la ligne 1 la liste est écrite explicitement à l'aide de constructions composites imbriquées. Cette formulation met en évidence la séquence des éléments, mais accumule les parenthèses et devient vite confuse.
La ligne 2 est équivalente à la ligne 1, mais souligne la structure récursive de la liste : on commence par la queue, qu'on utilise pour construire les versions suivantes successives en ajoutant des éléments en tête.
Enfin, la ligne 3 montre comment on utilise l'affectation composite décrite en § 3.2 au chapitre 2 pour accéder aux différents éléments d'une liste chaînée.

Conversion d'un tableau en une liste chaînée

Cet algorithme généralise le mécanisme des ajouts en tête successifs de la ligne 2.

```
Algorithme tableau_vers_liste : t → l selon
   l ← ∅, i ← nélém(t) # on commence par la fin
   Tant que i ⩾ 1 répéter l ← (t(i), l), i ← i − 1 .
.
```

2.2 Opérations fondamentales

Accès aux éléments constitutifs

Pour manipuler les éléments constitutifs d'une liste chaînée, on peut utiliser des routines pour effectuer la décomposition de la § 2.1, ligne 3.

```
Algorithme tête : réf l → réf t selon
   Si l = ∅ alors exception(“liste vide”) . # pas d'élément disponible
   (t, q) ← réf l # on récupère la tête en décomposant la liste
.
```

```
Algorithme queue : réf l → réf q selon
   Si l = ∅ alors q ← réf l # la queue d'une liste vide est vide
   sinon (t, q) ← réf l . # on récupère la queue en décomposant la liste
.
```

Remarquons qu'on a pris soin de passer les entrées et sorties par référence, comme expliqué en § 3.3 au chapitre 2, ce qui permet de modifier les éléments à l'aide de ces routines. Si au contraire on veut seulement récupérer la valeur de ces éléments sans les modifier, il suffit de ne pas spécifier de référence en sortie lors de l'appel.

Insérer ou extraire un élément en tête

On peut encore utiliser des procédures inspirées de la § 2.1, lignes 2 et 3. Le but étant de modifier la liste en entrée, on doit obligatoirement la passer par référence.

```
Algorithme insérer_tête : (réf l, x) → () selon l ← (x, l) .
```

```
Algorithme extraire_tête : réf l → x selon
   Si l = ∅ alors exception("liste vide") . # pas d'élément disponible
   (x, l) ← l
.
```

Recherche d'un élément dans une liste chaînée

Si l'on souhaite pouvoir modifier la liste à partir de l'élément recherché, on utilise encore des passages par référence.

```
Algorithme rechercher_cellule : (réf l, x) → réf c selon
   c ← réf l
   Tant que c ≠ ∅ et tête(c) ≠ x répéter c ← réf queue(c) .
. # si l'élément n'est pas dans la liste, la sortie est vide
```

Ici, la variable `c` parcourt les cellules de la liste chaînées jusqu'à trouver l'élément recherché ou arriver à la fin ; on prend bien soin d'utiliser une référence, pour la passer en sortie.

Remarque. En accord avec la table 4.1, on se convainc facilement que les complexités des routines `tête`, `queue`, `insérer_tête`, et `extraire_tête` sont de complexités temporelles constantes, c'est-à-dire qu'elles ne dépendent pas de la taille de la liste. Au contraire, la routine `rechercher_cellule` nécessite dans le pire des cas de parcourir toute la liste, donc de complexité temporelle linéaire en la taille de la liste.

Exercices et problèmes

Algorithmique

Exercice 1C. ☆ On souhaite mettre en œuvre une structure de donnée linéaire *dynamique* avec un tableau. Pour éviter de recréer un tableau à chaque changement de taille, on considère une construction composite `(tab, n)`, où `tab` est un tableau et {n} est le nombre de cases effectivement occupées par des données $0 \leqslant n \leqslant$ **`nélém(`***`tab`***`)`**.

Écrire et étudier la complexité de chacun des algorithmes suivants, qui requièrent en entrée un tableau, un indice, et un entier qui représente le nombre de cases du tableau effectivement utilisées.

(a) `extraire_case`, qui supprime l'élément du tableau à l'indice passé en entrée, et passe l'élément en sortie.

(b) `insérer_case`, qui accepte en entrée supplémentaire un élément quelconque, et qui l'insère à l'indice passé en entrée.

Exercice 1D. ☆ Écrire un algorithme `longueur` qui calcule la taille d'une liste chaînée.

Exercice 1E. ☆ Écrire un algorithme `indexer_cellule` qui accepte en entrée une liste chaînée et un entier, et passe en sortie une référence sur la cellule dont l'indice dans la liste est l'entier donné en entrée. Étudier sa complexité.

Exercice 1F. ☆ Écrire chacun des algorithmes suivants, qui acceptent en entrée une liste chaînée et un élément quelconque, et modifient la liste si l'élément s'y trouve.

(a) `extraire_cellule`, qui supprime de la liste la première occurrence de l'élément et passe l'élément en sortie.

(b) `insérer_cellule`, qui accepte en entrée supplémentaire un deuxième élément quelconque, et l'insère avant la première occurrence du premier élément donné en entrée.

(c) `modifier_cellule`, qui accepte aussi en entrée un deuxième élément quelconque, et remplace la première occurrence du premier élément par le deuxième élément.

Exercice 20. Écrire un algorithme `concaténer` qui acceptent en entrée deux listes chaînées et lie la deuxième à la suite de la première de sorte à ne former qu'une seule liste. Étudier sa complexité.

Exercice 21. On considère la copie d'une liste passée en entrée d'un algorithme.

(a) Écrire un algorithme `copier_inverser_liste`, qui construit une copie dans laquelle l'ordre des éléments est inversé.

(b) Écrire un algorithme `copier_liste`, qui construit une copie conforme à partir de la liste vide ; la complexité doit être linéaire en la taille de la liste, et chaque élément ne doit être copié qu'une seule fois.

Exercice 22. Le *mélange de* MONGE d'un paquet de cartes consiste à commencer un nouveau paquet avec la première carte, à placer la seconde au-dessous, puis la troisième au-dessus et ainsi de suite en plaçant les cartes d'indice impaire dans le paquet initial au-dessus du nouveau paquet, et les cartes d'indice paire au-dessous. Écrire un algorithme `mélange_de_Monge` qui accepte en entrée une liste chaînée et la réorganise selon le mélange de MONGE. La complexité doit être linéaire en la taille de la liste.

Exercice 23. On souhaite représenter des ensembles mathématiques à l'aide de listes chaînées ; chaque élément d'un ensemble ne doit apparaitre qu'une seule fois dans la liste qui le représente, à un emplacement arbitraire. Écrire et étudier les algorithmes suivants, qui acceptent en entrée deux ensembles.

(a) `intersection`, `union` et `difference_symétrique`, qui calculent en sortie l'ensemble correspondant.

(b) `est_inclus` et `est_égal`, qui vérifient la propriété correspondante.

Problème 10. ★ On considère les listes chaînées circulaires, où le dernier élément donne accès en temps constant au premier élément.

(a) ☆ Décrire un mécanisme très simple qui permet de manipuler une liste chaînée linéaire comme une liste chaînée circulaire

(b) ★ Le pseudo-langage permet aussi d'utiliser encore la construction `(tête, queue)`, mais en s'assurant que la queue du dernier élément donne accès à la première cellule. Écrire les algorithmes correspondant `insérer_tête_circ` et `extraire_tête_circ`.

(c) ★ Il manque un opérateur dans notre pseudo-langage pour faire un parcours d'une liste chaînée circulaire comme proposée au point (b). En proposer un, et écrire l'algorithme `longueur_circ`.

(d) ★★ Observer qu'on peut aussi introduire une boucle qui ne se referme sur une autre cellule que la tête de la liste chaînée. En supposant disponible l'opérateur du point (c), proposer un algorithme qui permet de déterminer si une liste chaînée contient une boucle.

Problème 11. ★ On considère les listes doublement chaînées, où chaque élément donne accès en temps constant à son prédécesseur et à son successeur.

(a) On envisage d'utiliser une construction `(prec, tête, suiv)`, où *prec* et *suiv* sont elles-mêmes de telles listes. Pourquoi notre pseudo-langage n'est-il pas adapté ?

(b) Proposer une autre construction plus adaptée, et des routines `précédent` et `suivant` qui permettent de la parcourir. *Indication : utiliser plusieurs listes simplement chaînées.*

Pseudo-langage et programmation

Exercice 24. ☆ On s'intéresse à la mise en œuvre des listes chaînées dans un module en langage C. On suggère les déclarations suivantes :

```
struct Cellule; typedef struct Cellule * Liste;.
```

(a) Écrire une définition de la structure `Cellule` pour les listes simplement chaînées. Considérer dans un premier temps que les listes contiennent des entiers.

(b) Imaginer comment représenter une liste vide et définir une macro `LISTE_VIDE`.

(c) Contrairement à notre pseudo-langage, le langage C exige une gestion explicite de la mémoire. Écrire les fonctions `inserer_tete` et `extraire_tete` qui permettent respectivement d'insérer et d'extraire une cellule en tête d'une liste. Veiller à gérer le cas d'une liste vide, et à respecter des complexités constantes.

(d) Écrire une fonction `afficher_liste`, qui permet d'afficher une liste chaînée.

(e) ★ Proposer des techniques pour définir des listes chaînées pouvant contenir d'autre types de données que des entiers avec le même code source générique. Attention à l'affichage, à la comparaison des éléments, etc.

(f) Reprendre les points (a) et (c) pour les listes doublement chaînées et circulaires.

© Drew EVANS, *Chain Brain*, 2021.
https://chainbreakerwelding.com/

5 Piles et files

Les *piles* et les *files* sont des structures de données fondamentales en informatique, permettant de manipuler des *séquences de données ou d'opérations*.

Représentant des séquences, les piles et les files sont donc des structures de données *linéaires*, au même titre que les tableaux et les listes chaînées décrits au chapitre 4.

Cependant, elles sont plus éloignées des considérations de mémoire que ne le sont les tableaux ou les listes chaînées, car pour les définir on se contentera de décrire les opérations qu'elles supportent : comment et à quel coût accède-t-on aux éléments qui les constituent. Cela s'appelle l'*encapsulation* : on gagne un niveau d'abstraction, et l'utilisateur de la structure peut ignorer la mise en œuvre concrète tant qu'il respecte les spécifications.

Dans un premier temps, nous allons donc décrire les spécifications des piles et des files. Puis, pour permettre de comprendre plus finement leur mécanisme et l'encapsulation, nous allons décrire dans un second temps plusieurs mises en œuvre possibles.

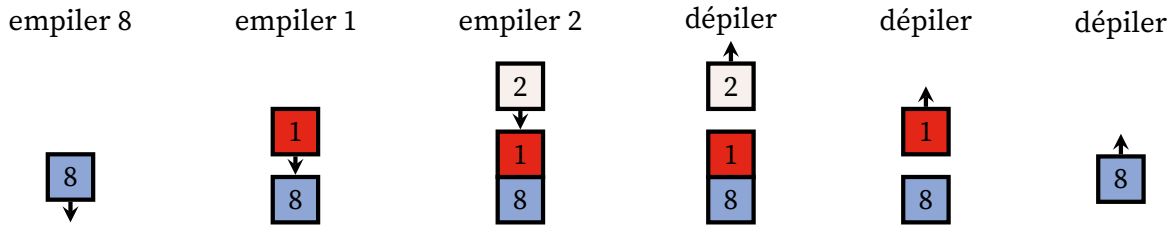
1 Définitions

Les définitions sont intuitives et les noms sont empruntés au vocabulaire courant.

1.1 Piles

Imaginons une pile d'objets quelconque : on ne peut facilement retirer ou ajouter un objet qu'à partir de son *sommet* (qui est le dernier élément), sinon la pile risque de s'écrouler.

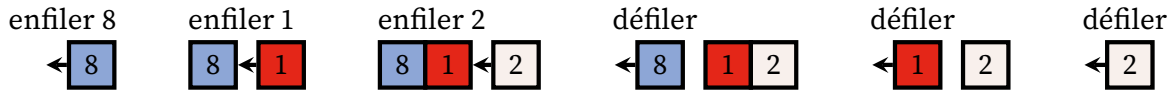
Les seules opérations possibles sont alors d'*empiler* un objet au sommet, ou d'enlever l'objet au sommet (on dira *dépiler*) . De plus, il faut pouvoir identifier si la pile est vide. Toutes ces opérations doivent être de complexité constante.



On retiendra que c'est toujours le dernier objet empilé qui est récupéré quand on dépile ; ce qu'on résume par l'expression « *dernier rentré, premier sorti* ».

1.2 Files

Les files fonctionnent comme les files d'attente : on peut ajouter un objet à la *queue*(qui est le dernier élément, on dira *enfiler*), ou enlever un objet à la *tête*[1] (qui est le premier élément, on dira *défiler*). Comme pour les piles, il faut pouvoir identifier si la file est vide, et toutes ces opérations doivent être de complexité constante.



Cette fois, c'est le premier objet enfilé qui est récupéré quand on défile ; ce qu'on résume par l'expression « *premier rentré, premier sorti* ».

1. Attention, les termes « tête » et « queue » désignent des concepts différents pour les listes chaînés et pour les files.

1.3 Utilisation en informatique

Évidemment, les files sont utiles pour modéliser des files d'attentes : si un service doit traiter plusieurs demandes, il est juste de traiter ces demandes dans l'ordre dans lesquelles elles sont arrivées.

Les piles ont moins d'analogue immédiat dans la vie quotidienne. On peut penser à une pile d'assiettes sur laquelle on viendrait toujours récupérer la dernière empilée ; dans ce cas l'ordre des opérations est imposé par la gravité.

Un exemple d'utilisation directe en informatique concerne les fonctionnalités d'annulation dans les logiciels de bureautique : les modifications successives sont stockées dans une pile, et chaque annulation concerne bien la dernière modification effectuée.

Cependant, ces exemples ne reflètent pas le caractère fondamental qu'ont ces structures en informatique.

Des piles ou des files sont en jeu dès qu'on doit réaliser une séquence d'opérations qui sont dépendantes les unes des autres. Si une première opération *nécessite* le résultat d'une deuxième opération qui elle même nécessite le résultat d'une troisième opération et ainsi de suite, cette séquence sera avantageusement représentée par une pile. Si au contraire une première opération *est nécessaire* pour effectuer une deuxième opération qui elle même est nécessaire pour effectuer une troisième opération et ainsi de suite, cette séquence sera avantageusement représentée par une file.

Ce sont des mécanismes élémentaires qu'on retrouve au sein de structures de données ou d'algorithmes plus complexes. Il est difficile de l'illustrer à notre niveau, mais on pourra s'en rendre compte par exemple à propos du concept de pile d'exécution au chapitre 6, § 2.2.

2 Mise en œuvre

Nous revenons maintenant sur le concept d'encapsulation, en montrant qu'une même structure de donnée abstraite peut être mise en œuvre de plusieurs façons.

2.1 Piles avec des tableaux

Puisque les tableaux sont des structures statiques, cette mise en œuvre est à privilégier quand le nombre maximum d'éléments que peut contenir la pile, qu'on appelle sa *capacité*, est explicitement limitée. En plus de pouvoir vérifier si une pile est vide, il faut maintenant aussi pouvoir vérifier si elle est pleine.

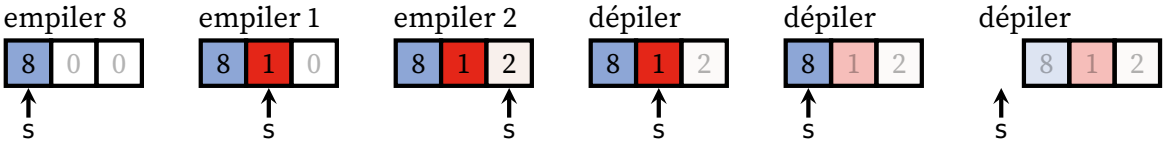
Pour représenter une pile avec un tableau, il suffit de maintenir l'indice du sommet de la pile. Cela se traduit par une construction composite `(`*`tableau`*`, `*`sommet`*`)` ; la taille du *`tableau`* détermine la capacité de la pile, et le *`sommet`* est égal à 0 quand celle-ci est vide, et à la taille du *`tableau`* quand celle-ci est pleine.

```
Algorithme créer_pile : c → ([c], 0) .
Algorithme est_vide : p → b selon
   (tab, s) ← p
   b ← s = 0
.
Algorithme est_pleine : p → b selon
   (tab, s) ← p
   b ← s = nélém(tab)
.
Algorithme empiler : (réf p, x) → () selon
   Si est_pleine(p) alors exception(“pile pleine”) .
   (tab, s) ← réf p
   s ← s + 1, tab(s) ← x
.
```

```
Algorithme dépiler : réf p → x selon
   Si est_vide(p) alors exception("pile vide") .
   (tab, s) ← réf p
   x ← tab(s), s ← s − 1
.
```

Exemple d'utilisation correspondant à l'illustration en § 1.1.

```
p ← créer_pile(3), empiler(p, 8), empiler(p, 1), empiler(p, 2)
x ← dépiler(p), y ← dépiler(p), z ← dépiler(p)
```



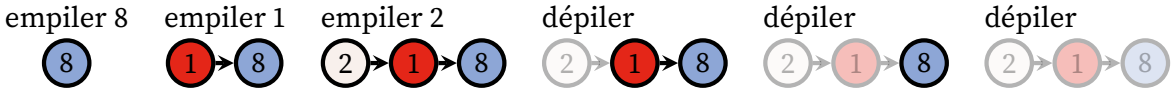
2.2 Piles avec des listes chaînées

Les listes chaînées se comportent déjà comme les piles ; la tête de la liste jouant le rôle du sommet de la pile. Nous pouvons donc écrire les opérations élémentaires en adaptant très légèrement les routines données au chapitre 4, § 2.2. En particulier, la pile vide est représentée par la liste vide.

```
Algorithme créer_pile : () → ∅ .
Algorithme est_vide : p → p = ∅ .
Algorithme empiler : (réf p, x) → () selon insérer_tête(p, x) .
Algorithme dépiler : réf p → x selon
   Si est_vide(p) alors exception("pile vide") .
   x ← extraire_tête(p)
.
```

Exemple d'utilisation correspondant à l'illustration en § 1.1.

```
p ← créer_pile, empiler(p, 8), empiler(p, 1), empiler(p, 2)
x ← dépiler(p), y ← dépiler(p), z ← dépiler(p)
```



2.3 Files avec des tableaux

Pour représenter une file avec un tableau, il faut maintenir les indices de la tête et de la queue. Cela se traduit par une construction composite `(`*`tableau`*`, `*`tête`*`, `*`queue`*`)`.

Notons que les indices délimitant le début et la fin de la file sont tous les deux variables ; la file vide est caractérisée par le fait que la *tête* est égale à la *queue*.

De plus, pour continuer à utiliser les premières cases du tableau, les indices *tête* et *queue* doivent être périodiques, revenant à 1 dès qu'ils dépassent la taille du tableau.

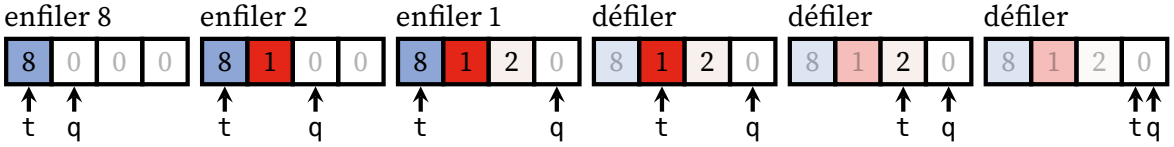
Enfin, pour distinguer la file vide de la file pleine, cette dernière est caractérisée par le fait que la queue se trouve *juste avant* la tête. On en déduit qu'il doit toujours rester au moins une case non utilisée, c'est pourquoi *la taille du tableau est d'un élément de plus que la capacité de la file*.

```
Algorithme créer_file : c → ([c + 1], 1, 1) .
Algorithme est_vide : f → b selon
   (tab, t, q) ← f
   b ← t = q
.
Algorithme est_pleine : f → b selon
   (tab, t, q) ← f
```

```
   b ← t = q \ nélém(tab) + 1
.
Algorithme enfiler : (réf f, x) → () selon
   Si est_pleine(f) alors exception("file pleine") .
   (tab, t, q) ← réf f
   tab(q) ← x, q ← q \ nélém(tab) + 1
.
Algorithme défiler : réf f → x selon
   Si est_vide(f) alors exception("file vide") .
   (tab, t, q) ← réf f
   x ← tab(t), t ← t \ nélém(tab) + 1
.
```

Exemple d'utilisation correspondant à l'illustration en § 1.2.

```
f ← créer_file(3), enfiler(f, 8), enfiler(f, 1), enfiler(f, 2)
x ← défiler(f), y ← défiler(f), z ← défiler(f)
```



2.4 Files avec des listes chaînées

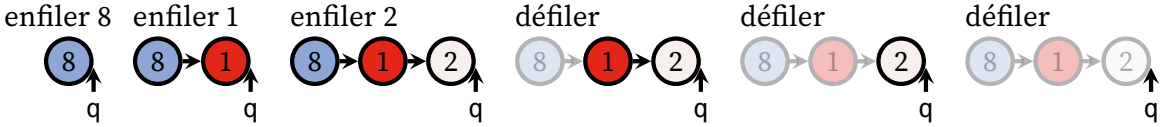
En plus de l'accès à la tête déjà fourni par la liste chaînée, il faut maintenir un accès en temps constant à la queue de la file. Cela se traduit par une construction `(liste, queue)`, où *queue est une référence vers la queue de la dernière cellule* de la `liste`.

Ce principe est facile à comprendre, mais notre pseudo-langage ne permettant pas de manipuler directement des adresses mémoire, cela donne lieu à des subtilités sur lesquelles il est inutile de s'attarder. Nous donnons les routines suivantes par souci d'exhaustivité.

```
Algorithme créer_file : () → file_vide selon
   liste_vide ← ∅, file_vide ← (liste_vide, réf liste_vide)
.
Algorithme est_vide : f → b selon
   (l, q) ← f
   b ← l = ∅
.
Algorithme enfiler : (réf f, x) → () selon
   (l, q) ← réf f          # q est une référence sur la queue de la file
   q ← (x, ∅)              # la queue est modifiée
   f ← (l, réf queue(q)) # référence en sortie de la routine queue
.
Algorithme défiler : réf f → x selon
   Si est_vide(f) alors exception("file vide") .
   (l, q) ← réf f
   x ← extraire_tête(l)
.
```

Exemple d'utilisation correspondant à l'illustration en § 1.2.

```
f ← créer_file, enfiler(f, 8), enfiler(f, 1), enfiler(f, 2)
x ← défiler(f), y ← défiler(f), z ← défiler(f)
```



Exercices et problèmes

Ordre des opérations, analyse de syntaxe

Dans les exercices 25 et 26 et problème 12, **on suppose qu'une expression est donnée sous la forme d'un tableau ou d'une file de caractères** : c'est ce que voit un ordinateur quand il doit analyser du code source pour le traduire en instructions.

Exercice 25. ☆ Dans notre pseudo-langage comme dans l'usage commun, le format des opérateurs arithmétiques est *infixe* : les opérandes s'écrivent de part et d'autre de l'opérateur, par exemple `x + y`. Dans le format *préfixe*, [2] l'opérateur apparait avant les opérandes, par exemple `+ x y` ; dans le format *postfixe*, [3] l'opérateur apparait après, `x y +`.

Même s'ils sont moins lisibles par un humain, ces formats présentent plusieurs avantages ; l'un d'eux est de ne pas nécessiter de parenthèses. Par exemple dans le format postfixe, `(x + y) × z` s'écrit `x y + z ×` ; en effet, en lisant de gauche à droite, *on peut évaluer une opération dès qu'on rencontre un opérateur ; le résultat peut alors être réutilisé pour l'opérateur suivant*.

En s'inspirant de cette indication, écrire un algorithme `évaluer_postfixe` qui accepte en entrée une chaîne de caractère représentant une expression arithmétique postfixe, et calcule la valeur résultante. Pour simplifier, on supposera que chaque caractère dans la chaîne représente soit un des opérateurs binaires ' + ' ' − ' ' × ' ' ÷ ', soit une variable ; dans ce dernier cas on accède à sa valeur avec la routine `valeur`.

Exercice 26. ☆ Une expression est *bien parenthésée* si on peut associer deux à deux les parenthèses ouvrantes et fermantes qu'elle contient, de telle sorte que :

— chaque parenthèse ouvrante se trouve avant la parenthèse fermante associée, et

— si une parenthèse se trouve entre deux parenthèses associées alors sa parenthèse associée s'y trouve aussi.

Une telle association est qualifiée de *correcte*.

(a) Donner toutes les configurations bien parenthésées à quatre parenthèses.

(b) ★ Prouver que l'association correcte d'une expression bien parenthésée est unique.
Indication : supposer qu'il existe deux associations correctes différentes, et considérer la première parenthèse fermante *de la liste pour laquelle les associations diffèrent.*

(c) Écrire un algorithme `analyser_parenthèses`, qui accepte en entrée une chaîne de caractères représentant une expression, et qui vérifie si elle est bien parenthésée. Le cas échéant, la valeur calculée est l'association correcte des parenthèses, représentée par une file de couples d'entiers indexant la chaîne de caractères donnée en entrée.

Problème 12. ★ On considère les affectations composites de notre pseudo-langage, décrites en § 3.2 du chapitre 2.

(a) Écrire un algorithme `décomposer` qui accepte en entrée une file de caractères représentant une construction composite, et calcule sa décomposition en éléments constitutifs, représentée par une file, où les éléments sont eux-mêmes des files de caractères.

Une affectation peut être représentée par deux constructions composites : celle de gauche qui contient les variables à définir, et celle de droite qui contient les valeurs à affecter.

(b) Écrire un algorithme `analyser_affectation_composite` qui accepte en entrée deux files de caractères représentant une affectation composite, et qui calcule si elle est syntaxiquement valide. Le cas échéant, la valeur calculée est l'affectation, représentée par une file de couples de files de caractères.

2. Aussi appelé *notation polonaise* en hommage au logicien polonais Jan ŁUKASIEWICZ qui l'a introduite ; utilisée dans le langage Lisp

3. Aussi appelé *notation polonaise inverse*, utilisée par les calculatrices scientifiques Hewlett–Packard, ainsi que dans le langage PostScript.

Mises en œuvre de piles et files

Exercice 27. ☆ On met en œuvre une file avec un tableau, comme expliqué en § 2.3. Représenter le tableau, la tête et la queue de la file, après chacune des opérations suivantes.

```
f ← créer_file(2)
enfiler(f, 'Q'), enfiler(f, 'V'), défiler(f),
enfiler(f, 'L'), défiler(f), enfiler(f, 'C')
```

Exercice 28. ☆ On souhaite représenter deux piles à l'aide d'un seul tableau, qui fonctionne tant que le nombre total d'éléments dans les deux piles n'excède pas la taille du tableau. Écrire les routines `créer_double_pile`, `est_vide`, `est_pleine`, `empiler_double` et `dépiler_double` qui permettent de les mettre en œuvre ; les quatre dernières accepteront une entrée supplémentaire permettant de préciser la pile à utiliser.

Exercice 29. ☆ Une *file à double entrée* est une file où l'on peut ajouter et extraire des éléments à la fois par la queue et par la tête. Écrire les routines `créer_file_double`, `est_vide`, `est_pleine`, `enfiler_double` et `défiler_double` qui permettent de la mettre en œuvre à l'aide d'un tableau ; les deux dernières accepteront une entrée supplémentaire permettant de préciser si l'on veut agir sur la tête ou sur la queue.

Exercice 2A. Dans certaines applications, on manipule une pile de capacité limitée, mais on permet de rajouter un élément au sommet même si la pile est pleine, en supprimant l'élément en bas de la pile. C'est le cas par exemple des fonctionnalités d'annulation dans les logiciels de bureautique évoquées en § 1.3 : on ne peut revenir en arrière que jusqu'à un certain point.

(a) Écrire un algorithme `empiler_écraser` qui permet d'effectuer cette opération sur une pile normale. Analyser sa complexité spatiale et temporelle dans le pire des cas.
Indication : on pourra se servir d'une pile auxiliaire.

(b) Proposer une structure de données émulant une pile de capacité limitée, permettant la fonctionnalité `empiler_écraser` en complexités spatiale et temporelle constantes. On écrira aussi les routines de base `créer_pile`, `est_vide`, `est_pleine` et `dépiler` pour cette structure.
Indication : s'inspirer de la mise en œuvre de files *à l'aide de tableaux.*

Exercice 2B. On souhaite mettre en œuvre une pile à l'aide de deux files, en autorisant que l'une des opérations d'ajout ou de retrait d'élément soit de coût temporel linéaire en la taille de la pile. Écrire les routines `créer_pile`, `est_vide`, `empiler` et `dépiler` qui permettent de la mettre en œuvre.

Exercice 2C. On souhaite mettre en œuvre une file à l'aide de deux piles, en autorisant que l'une des opérations d'ajout ou de retrait d'élément soit de coût temporel linéaire en la taille de la file. Écrire les routines `créer_file`, `est_vide`, `enfiler` et `défiler` qui permettent de la mettre en œuvre.


© LEVALET, gauche : *Service après vente*, 2014 ; droite : *La rumeur*, 2017. https://www.levalet.xyz/

6 Récursivité

1 Principes

On qualifie de *récursif* tout concept qui intervient dans sa propre définition. Par exemple, on a définit les listes chaînées comme étant la donnée d'un élément et d'une liste chaînée. Remarquons d'ores et déjà que cette définition seule est impossible à mettre en pratique si l'on ne dispose pas d'au moins une liste chaînée à partir de laquelle construire les autres ; dans notre pseudo-langage, c'est le rôle de la liste vide $\emptyset$.

La récursivité peut aussi s'appliquer à des algorithmes. Reprenons le principe de la multiplication égyptienne des chapitres 2 et 3 : $m \times n = \begin{cases} (m/2) \times (2n) & \text{si } m \text{ est paire,} \\ (m-1) \times n + n & \text{sinon.} \end{cases}$.
On peut ramener le problème de la multiplication de deux entiers m et n à celle de $m/2$ et $2n$ ou à celle de $m-1$ et n (suivie d'une addition) ; dans les deux cas, on peut faire un appel récursif. Une fois de plus, il nous faut un cas particulier pour s'assurer que l'algorithme ne se contente pas seulement de faire appel à lui-même indéfiniment, mais qu'il calcule bien une valeur en un temps fini. Ici, c'est quand m est nul : on sait que le résultat est nul.

```
Algorithme multiplication_égyptienne : (m, n) ⟶ p selon
   Si m = 0 alors p ⟵ 0
   sinon
      Si m \ 2 = 0 alors p ⟵ multiplication_égyptienne(m / 2, n × 2)
      sinon p ⟵ multiplication_égyptienne(m − 1, n) + n .
   .
.
```

On peut comparer avec la version proposée au chapitre 2 : d'un côté, la version récursive met bien en évidence la propriété mathématique sur laquelle l'algorithme repose ; en revanche, elle rend moins explicites les variables nécessaires et leur évolution au cours des itérations.

On oppose ainsi la programmation récursive à la programmation *impérative* . La première est souvent plus abstraite et simplifie la compréhension et l'analyse de l'algorithme, tandis que la seconde s'attache plus à décrire les opérations élémentaires et l'état de la mémoire au cours de l'exécution.

On comprend alors que l'informaticien préférera souvent la description récursive, tandis qu'un ordinateur nécessitera moins d'étapes et pourra mieux optimiser l'interprétation d'un code source impératif.

De ce fait, certains langages de programmation n'encouragent pas, voire n'autorisent pas, les appels récursifs ; à l'inverse, d'autres sont spécialisés dans l'expression de structures et processus récursifs. Retenons néanmoins que ce sont deux *styles* différents qui peuvent exprimer plus ou moins facilement les mêmes algorithmes : on peut montrer que tout algorithme récursif possède une version impérative, et que toute boucle peut s'écrire sous une forme récursive.

Bien que nous verrons essentiellement des cas simples, mentionnons ici que la récursivité peut donner lieu à des constructions nettement plus compliquées. Premièrement, la récursion peut bien sûr impliquer plusieurs variables (dans l'exemple ci-dessus, seulement la valeur de `m` décroit à chaque itération et détermine la condition d'arrêt). Ensuite, un algorithme récursif peut faire appel plusieurs fois à lui-même dans sa définition, on parle alors de *récursivité multiple* ; quand de plus la sortie d'un appel récursif sert d'entrée à un autre appel récursif, on parle de *récursivité imbriquée*. Enfin, un algorithme peut utiliser dans sa définition un second algorithme qui utilise lui-même le premier algorithme, on parle alors de *récursivité croisée*.

2 Analyse des algorithmes récursifs

2.1 Terminaison et induction

Comme nous avons déjà pu le mentionner en § 1, il faut s'assurer que la récursion n'est pas infinie, c'est-à-dire dans les termes du chapitre 3, que l'algorithme *termine*.

Pour cela, on montre en fait que les entrées des appels récursifs successifs constituent bien un *convergent* au sens défini en § 1.2 du du chapitre 3 : toute séquence de ces entrées atteint l'ensemble d'arrêt.

Tout algorithme récursif suit alors la forme canonique suivante

```
1 Algorithme récursif : x ⟶ y selon
2    Si est_dans_A(x) alors y ⟵ résultat(x)
3    sinon y ⟵ calcul(x, récursif(phi(x))) .
4 .
```

où `est_dans_A` vérifie si l'entrée est dans l'ensemble d'arrêt, le cas échéant `résultat` calcule directement le résultat, et sinon `calcul` effectue des opérations à partir des entrées et d'un appel récursif dont les nouvelles entrées sont données par `phi`.

Par exemple pour la multiplication égyptienne, les entrées sont à valeur dans $E = \mathbb{N} \times \mathbb{N}$, l'ensemble d'arrêt est $A = \{0\} \times \mathbb{N}$, avec les routines suivantes.

```
Algorithme est_dans_A : (m, n) ⟶ m = 0 .
Algorithme résultat : (m, n) ⟶ 0 .
Algorithme phi : (m, n) ⟶ (m, n) selon
   Si m \ 2 = 0 alors (m, n) ⟵ (m / 2, n × 2)
   sinon (m, n) ⟵ (m − 1, n) .
.
Algorithme calcul : ((m, n), p) ⟶ p selon
   Si m \ 2 ≠ 0 alors p ⟵ n + p .
.
```

Si on veut faire des preuves dans un cadre général, on utilise alors la généralisation suivante du principe de récurrence énoncé en § 1 au chapitre 3.

Théorème (Principe d'induction). Soit E un ensemble bien fondé, A une partie de E, $\varphi : E \setminus A \to E$ une application vérifiant $\forall x \in E \setminus A, \varphi(x) < x$, et enfin un prédicat $\mathcal{P}$ sur E vérifiant $\forall x \in A, \mathcal{P}_x$ ainsi que $\forall x \in E \setminus A, \mathcal{P}_{\varphi(x)} \implies \mathcal{P}_x$. Alors pour tout $x \in E$, $\mathcal{P}_x$.

Remarque. on retrouve le principe de récurrence, en posant $E = \mathbb{N}$, $A = \{0\}$ et $\varphi : n \mapsto n - 1$.

En étudiant le prédicat $\mathcal{P}_x$: *l'algorithme termine pour l'entrée* x, le principe d'induction prouve $\mathcal{P}_x$ pour tout $x \in E$, c'est-à-dire garantit la terminaison de l'algorithme, dès que les ensembles d'entrée, d'arrêt, et la routine `phi` vérifient les propriétés ci-dessus.

Ces considérations s'étendent facilement aux appels récursifs multiples. Par exemple pour deux appels récursifs, il suffit de remplacer la ligne 3 de l'algorithme générique par

```
   sinon y ⟵ calcul(x, récursif(phi1(x)), récursif(phi2(x))) .
```

où `phi1` et `phi2` ont les mêmes propriétés que `phi`.

2.2 Pile d'exécution et récursivité terminale

Examinons ce qui se passe lors d'un appel `y0 ⟵ récursif(x0)`, à supposer que l'exécution nécessite trois appels récursifs.

```
# récursif(x0) requiert calcul(x0, récursif(phi(x0)))
   x1 ⟵ phi(x0) # récursif(x1) requiert calcul(x1, récursif(phi(x1)))
      x2 ⟵ phi(x1) # récursif(x2) requiert calcul(x2, récursif(phi(x2)))
         x3 ⟵ phi(x2) # x3 est dans l'ensemble d'arrêt
```

```
          y3 ← résultat(x3)
       y2 ← calcul(x2, y3)
   y1 ← calcul(x1, y2)
y0 ← calcul(x0, y1)
```

Dans cette description, chaque alinéa correspond à un appel récursif. Pour ne pas confondre les entrées et sorties des appels successifs, nous les avons appelées `(x0, y0)`, `(x1, y1)`, etc. De manière générale, quand une routine est appelée dans un autre algorithme, il faut retenir le *contexte* de l'appel : les entrées et sorties, et toute autre variable temporaire nécessaire.

Les appels récursifs ne font pas exception. Pour exécuter un algorithme récursif, il faut donc en général maintenir une *pile d'exécution*, dans laquelle le contexte de chaque appel récursif sera empilé, et ne pourra être dépilé qu'une fois la sortie correspondante calculée.

Or, la pile d'exécution d'un ordinateur a une capacité limitée. Ce coût en mémoire, en général linéaire en le nombre d'appels récursifs, ne doit pas être ignoré : trop d'appels récursifs risquent de faire *déborder la pile*, ce qui se traduit généralement par un arrêt brutal du programme.

Il existe cependant un cas particulier : la *récursivité terminale*. Un algorithme est récursif terminal lorsque *l'appel récursif est sa dernière instruction* : la sortie de l'algorithme sera directement la sortie de l'appel récursif sans modification.

Dans le schéma algorithmique canonique donné en § 2.1, cela se traduit par l'absence de routine `calcul`, et la ligne 3 devient tout simplement

```
sinon y ← récursif(phi(x)) .
```

Ce cas particulier est important, car il n'est alors plus nécessaire de retenir le contexte d'un algorithme récursif terminal, il est possible de le remplacer par celui de l'appel récursif ; dans notre exemple, on remplace l'information `x` par `phi(x)`. Le coût spatiale dans la pile d'exécution est alors constant.

Il arrive fréquemment qu'on puisse transformer un algorithme récursif en algorithme récursif terminal en utilisant une routine auxiliaire, qui accepte en entrée une variable auxiliaire qui transporte les informations nécessaires, comme dans l'exemple suivant.

```
Algorithme mult_rec_term : (m, n, p) ⟶ p selon
   Si m > 0 alors
      Si m \ 2 = 0 alors p ← mult_rec_term(m / 2, n × 2, p)
      sinon p ← mult_rec_term(m − 1, n, p + n) .
   .
.
Algorithme multiplication_égyptienne : (m, n) ⟶ mult_rec_term(m, n, 0) .
```

2.3 Complexité temporelle

La récursivité permet d'exprimer simplement des algorithmes compliqués, mais il faut se méfier des coûts cachés.

En plus des coûts spatiaux dans la pile d'exécution, un autre écueil à éviter est l'appel récursif multiple naïf : il arrive que certains appels deviennent redondant, et que certaines quantités soient calculées plusieurs fois.

Ce phénomène est particulièrement bien illustré par l'exercice 30.

3 Diviser pour régner

« *Diviser pour régner* » est une technique algorithmique importante qui puise son inspiration dans la récursivité. Elle consiste à *ramener la solution d'un problème à la solution de problèmes similaires de tailles plus petites*.

3.1 Deux exemples classiques

Considérons le calcul d'une puissance entière d'un nombre réel. On peut penser à l'algorithme naïf suivant.

```
Algorithme exponentiation_naïve : (x, p) → y selon
   Si p = 0 alors y ← 1
   sinon y ← y × exponentiation_naïve(x, p − 1)
.
```

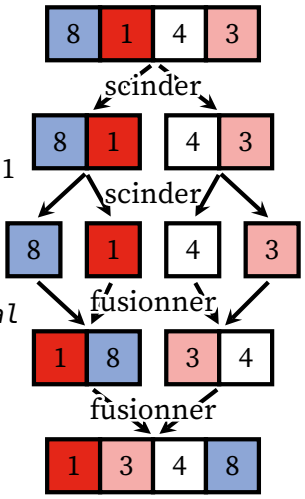
Pour calculer x^p, cet algorithme nécessite à l'évidence p multiplications. Mais la multiplication est associative, donc beaucoup sont redondantes : si l'on a déjà calculé $x^{p/2}$, une seule multiplication supplémentaire suffit pour calculer x^p. En faisant attention à la parité, cela donne l'algorithme suivant.

```
Algorithme exponentiation_rapide : (x, p) → y selon
   Si p = 0 alors y ← 1
   sinon
      tmp ← exponentiation_rapide(x, p / 2)
      Si p \ 2 = 0 alors y ← tmp × tmp
      sinon y ← x × tmp × tmp .
   .
.
```

Un autre exemple un peu plus compliqué est le tri d'un tableau, dont des versions naïves sont traitées aux exercices 09 et 0A du chapitre 3. Pour appliquer la technique diviser pour régner, on partage le tableau en deux parties égales que l'on trie chacune de son côté. Il reste alors à fusionner les deux sous-tableaux, en tirant partie du fait qu'ils sont triés.

Pour scinder un tableau en plusieurs parties sans avoir à créer un nouveau tableau, on repère les sous-tableaux avec des indices de début `i`, `j` et une taille totale `n`. Pour simplifier la fusion, on l'effectue dans un tableau temporaire qu'on recopie ensuite dans le tableau initial.

```
Algorithme fusionner : (réf tab, i, j, n) → () selon
   tmp ← [n]       # tableau temporaire
   ii ← i, ji ← j # indices initiaux
   Pour k parcourant (1, ..., n) répéter
      Si i = ji alors tmp(k) ← tab(j), j ← j + 1
      sinon si j = ii + n alors tmp(k) ← tab(i), i ← i + 1
      sinon si tab(i) < tab(j) alors
         tmp(k) ← tab(i), i ← i + 1
      sinon tmp(k) ← tab(j), j ← j + 1 .
   . # fusion terminée ; recopie dans le tableau initial
   Pour k parcourant (1, ..., n) répéter
      tab(ii + k − 1) ← tmp(k)
   .
.
```




```
Algorithme tri_fusion_rec : (réf tab, i, n) → () selon
   Si n > 1 alors
      tri_fusion_rec(tab, i, n / 2)       # tri de la première partie
      tri_fusion_rec(tab, i + n / 2, n − n / 2) # tri de la seconde partie
      fusionner(tab, i, i + n / 2, n)
   .
.

Algorithme tri_fusion : réf t → () selon tri_fusion_rec(t, 1, nélém(t)) .
```

3.2 Analyse de complexité

La complexité temporelle d'un algorithme diviser pour régner suit toujours une relation de récurrence, dans laquelle apparait le coût de chacun des appels récursifs sur les entrées de tailles réduites, et le coût des opérations nécessaires pour combiner le résultat des appels récursifs.

En notant $(c_n)_{n\in\mathbb{N}}$ la complexité pour une taille de problème n, si à chaque étape on réduit de moitié cette taille on obtient une relation de récurrence de la forme

$$\text{pour tout } n > 0, \quad c_n = ac_{\lceil n/2\rceil} + bc_{\lfloor n/2\rfloor} + d_n \ . \tag{6.1}$$

La plupart du temps, a et b valent 0 ou 1, selon s'il faut résoudre les problèmes de tailles réduites pour chaque moitié : pour le cas de l'exponentiation rapide, on a $a = 0$ et $b = 1$; pour le cas du tri par fusion, $a = 1$ et $b = 1$. La suite $(d_n)_{n\in\mathbb{N}}$ représente les opérations autres que les appels récursifs.

L'analyse de la complexité en fonction des valeurs de a, b et du comportement de la suite $(d_n)_{n\in\mathbb{N}}$ est connue sous le nom de théorème « maître ». Nous donnons ici les cas particuliers les plus intéressants à retenir, accompagnés d'une démonstration détaillée qui illustre des mécanismes classiques d'analyse de complexité.

Théorème (dit « maître », cas particuliers). Soit $(c_n)_{n\in\mathbb{N}}$ une suite réelle positive vérifiant la relation de récurrence 6.1.

(i) Si $a = 0$ et $b = 1$ ou si $a = 1$ et $b = 0$, et $d_n = \Theta(1)$, alors $c_n = \Theta(\log n)$;

(ii) Si $a = 1$ et $b = 1$, et $d_n = \Theta(n)$, alors $c_n = \Theta(n \log n)$.

Démonstration. La méthode est toujours la même : pour se débarasser des termes compliqués en partie entière $\lfloor n/2\rfloor$, $\lceil n/2\rceil$, on étudie le comportement des termes de rangs en puissances de 2, $n = 2^p$, et on simplifie le comportement de la suite u_n. On conclut sur le cas général par croissance et encadrement.

(i). Considérons le cas $a = 1$ et $b = 0$ et $d_n = \Theta(1)$.

Si on suppose $d_n = M$ constant, alors on a $c_n = c_{\lceil n/2\rceil} + M$, et pour tout $p \geqslant 1$

$$c_{2^p} = c_{2^{p-1}} + M \ .$$

En posant $v_p = c_{2^p}$, on s'aperçoit que $v_p = v_{p-1} + M$, c'est une suite arithmétique. On en déduit $v_p = v_0 + Mp$ c'est-à-dire $c_{2^p} = c_1 + Mp$.

Pour les termes qui ne sont pas des puissances de 2, on montre (par récurrence) que c est croissante, et donc avec $p = \lfloor \log n\rfloor$, $c_{2^p} \leqslant c_n \leqslant c_{2^{p+1}}$ soit encore $c_1 + M(\log n - 1) \leqslant c_n \leqslant c_1 + M(\log n + 1)$ et finalement $c_n = \Theta(\log n)$.

Finalement, si d_n n'est pas constant, on sait qu'à partir d'un certain rang, $M \leqslant d_n \leqslant M'$. On peut alors définir deux suites u et u' par récurrence $u_n = u_{\lceil n/2\rceil} + M$ et $u'_n = u'_{\lceil n/2\rceil} + M'$, qui encadrent c_n. D'après ce qui précède $u_n = \Theta(\log n)$ et $u'_n = \Theta(\log n)$, et donc $c_n = \Theta(\log n)$.

Le cas $a = 0$ et $b = 1$ se traite de la même façon.

(ii) On considère maintenant $a = b = 1$ et $d_n = \Theta(n)$.

Si on suppose $d_n = Mn$, on a $c_n = c_{\lceil n/2\rceil} + c_{\lfloor n/2\rfloor} + Mn$, et pour tout $p \geqslant 1$,

$$c_{2^p} = 2c_{2^{p-1}} + M2^p \ ,$$

donc en posant cette fois $v_p = c_{2^p}/2^p$, on a encore $v_p = v_{p-1} + M$. On en déduit $v_p = v_0 + Mp$ c'est-à-dire pour tout n tel que $n = 2^p$, $c_n = 2^p v_0 + Mp2^p = nv_0 + n\log n$.

On montre encore que c est croissante et on encadre les termes de rangs qui ne sont pas des puissances de 2 pour obtenir $c_n = \Theta(n \log n)$.

Finalement, si d_n n'est pas exactement Mn, on sait qu'à partir d'un certain rang, $Mn \leqslant d_n \leqslant M'n$. On peut encore définir deux suites u et u' par récurrence $u_n = u_{\lceil n/2\rceil} + u_{\lfloor n/2\rfloor} + M$ et $u'_n = u'_{\lceil n/2\rceil} + u'_{\lfloor n/2\rfloor} + M'$, qui encadrent c_n, et conclure avec ce qui précède. ■

Exercices et problèmes

Théorie et mise en œuvre classique

Exercice 2D. Prouver le principe d'induction énoncé en § 2.1.

Exercice 2E. ☆ Traduire le schéma algorithmique récursif donné en § 2.1 en un algorithme impératif, en simulant les appels récursifs à l'aide d'une pile.

Exercice 2F. ☆ Écrire des versions récursives des solutions proposées pour les exercices 06, 07 et 09–0B au chapitre 2, et pour les exercices 1D–22 au chapitre 4.

Exercice 30. On s'intéresse à la suite de FIBONACCI, définie par $u_0 = u_1 = 1$, et pour tout $n \geqslant 2$, $u_n = u_{n-1} + u_{n-2}$.

(a) Donner un ordre de grandeur asymptotique de la suite de FIBONACCI.

(b) Écrire un algorithme récursif qui accepte en entrée un entier naturel et calcule le terme de rang correspondant. Analyser sa complexité.

(c) Écrire une version impérative. Analyser sa complexité.

(d) Écrire une version récursive ayant même complexité temporelle que la version impérative.

(e) Écrire une version récursive ayant même complexité que la version impérative.

Exercice 31. L'algorithme d'exponentiation rapide présenté en § 3 divise par 2 l'exposant à chaque étape. Écrire une version qui divise par 3 l'exposant, et analyser sa complexité.

Exercice 32. L'algorithme de tri par fusion présenté en § 3 se prête bien au tri de listes chaînées. On considère donc des listes dont les éléments sont ordonnés par la relation $\leqslant$.

(a) ★ Écrire un algorithme `scinder` qui accepte en entrée une liste chaînée et la scinde en deux listes indépendantes de longueurs ne différant pas plus de un, et passant en sortie une référence sur chacune d'elles.

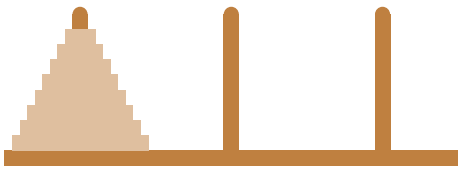
(b) Écrire un algorithme récursif `fusionner` qui accepte en entrée deux listes chaînées chacune triée dans l'ordre croissant, et qui les fusionne en une seule liste triée dans l'ordre croissant, passée en sortie par référence.

(c) Écrire un algorithme `tri_fusion` qui accepte en entrée une liste chaînée et la trie par ordre croissant.

Exercice 33 (Tours de Hanoï). Les *tours de Hanoï* est un casse-tête inventé par le mathématicien Édouard LUCAS. On dispose de trois tiges, sur lesquelles peuvent être enfilés n disques de diamètres différents. Au début du jeu, ces disques sont tous enfilés sur la même tige, et le but est de déplacer tous les disques sur une autre tige, en respectant les règles suivantes :

— un seul disque peut être déplacé à la fois ;

— on ne peut jamais poser un disque sur un disque de diamètre inférieur.



On modélise les tours de Hanoï par trois piles `(pile1, pile2, pile3)` ; initialement, la première contient les entiers de n à 1 dans cet ordre, et les deux autres sont vides.

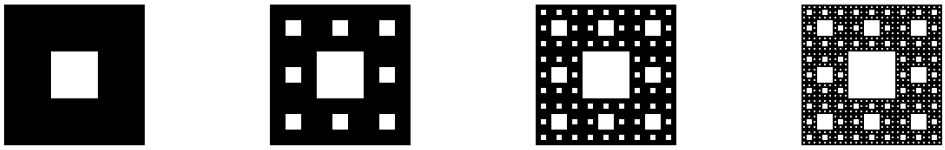
(a) ☆ Écrire un algorithme récursif qui accepte en entrée des tours de Hanoï, et qui simule la résolution du problème en effectuant les opérations élémentaires nécessaires.
Indication : que faut-il faire avant de pouvoir déplacer le plus grand disque ?

(b) ☆ Déterminer le nombre minimal d'opérations élémentaires nécessaires.

(c) On rajoute une règle supplémentaire : les déplacements directs entre la première et la troisième pile sont interdits, tous les déplacements doivent dépiler depuis, ou empiler sur, la deuxième pile. Traiter les points (a) et (b) dans ce cas.

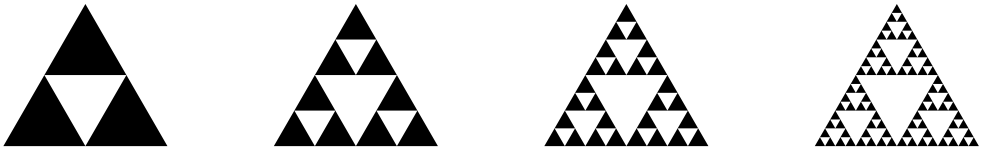
Figures géométriques récursives

Exercice 34 (Tapis de Sierpiński). À l'ordre zéro, le *tapis de Sierpiński* est un carré. À l'ordre un, on le divise en neuf carrés égaux et on évide le carré central. Aux ordres suivant, on répète récursivement l'opération sur chacun des carrés non évidés.



Si le carré initial est entre les points du plan $(0, 0)$ et $(1, 1)$, écrire un algorithme qui accepte en entrée un point du plan et un entier naturel, et vérifie si le point fait partie du tapis de Sierpiński à l'ordre correspondant.

Exercice 35 (Triangle de Sierpiński). Voici les *triangles de Sierpiński* d'ordres un à quatre ; tous les triangles sont équilatéraux.

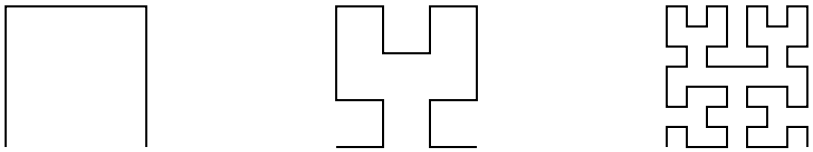


On suppose disposer d'un algorithme `triangle`, qui accepte en entrée trois points du plan, et qui trace le triangle plein dont les sommets sont les trois points donnés. Écrire un algorithme qui accepte en entrée un entier naturel et qui trace le triangle de Sierpiński à l'ordre correspondant, dont la base est le segment $[(0, 0), (1, 0)]$.

Problème 13 (Tortue graphique et courbe de Hilbert). ★★ Apparu en 1967, le langage de programmation Logo est connu pour sa « *tortue graphique* », vision imagée d'un robot qui se déplacerait sur l'écran en laissant derrière lui une trace de son passage.

Dotons notre pseudo-langage d'une version très simplifiée : l'instruction `avancer` fait avancer la tortue d'une certaine quantité prédéfinie en traçant un segment de droite ; les instructions `droite` et `gauche` font changer la direction de la tortue d'un quart de tour, respectivement vers la droite et vers la gauche.

Voici les *courbes de Hilbert* d'ordres un à trois.



Écrire un algorithme qui accepte en entrée un entier naturel qui trace la courbe de Hilbert d'ordre correspondant.

Jon Sullivan, http://pdphoto.org/