Complément d'informatique
INFO0952

Pierre Geurts

Dernière mise à jour le 23 décembre 2019

E-mail	:	p.geurts@uliege.be
URL	:	http://www.montefiore.ulg.ac.be/~geurts/ci.html
Bureau	:	1.134 (Montefiore)
Téléphone	:	04.366.48.15

Contact

- Chargé de cours :
  - Pierre Geurts, p.geurts@uliege.be, 1.134 Montefiore, 04/3664815
- Assistants :
  - Nicolas Vecoven, nvecoven@uliege.be, 1.103 Montefiore
  - Pascal Leroy, pleroy@uliege.be, 1.136 Montefiore
- Sites web du cours :
  - Cours théorique : http://www.montefiore.ulg.ac.be/~geurts/ci.html
  - Répétitions et projets : http://www.montefiore.ulg.ac.be/~nvecoven/ci/ci.html

Les objectifs du cours

- Consolider et étendre vos connaissances d'un langage de programmation (le C)
- Vous apprendre à écrire des programmes pour résoudre des problèmes réalistes (de taille moyenne)
- Vous initier à l'algorithmique et à l'étude des structures de données
- Vous ouvrir à d'autres paradigmes de programmation
- Et, on ne sait jamais, vous donner le goût pour la programmation (et l'informatique)

L'informatique dans le bachelier ingénieur

Bloc 1 :

- (Obl) INFO2009 - Introduction à l'informatique
- (Obl) INFO0061 - Organisation des ordinateurs

Bloc 2 :

- **(Obl) INFO0952 - Complément d'informatique**
- INFO0902 - Structures de données et algorithmes
- INFO0062 - Object-oriented programming

Bloc 3 :

- INFO0012 - Computation structures
- INFO0004 - Projet de programmation orientée-objet
- INFO0009 - Base de données
- INFO0054 - Programmation fonctionnelle
- INFO0010 - Introduction to computer networking
- INFO8006 - Introduction to artificial intelligence

Approche pédagogique

Apprentissage par la pratique :

- 6 ou 7 séances de travaux dirigés sur papier et/ou ordinateur encadrés par des assistants et élèves-moniteurs
- 2 “petits” devoirs de programmation à réaliser individuellement, lors des séances de travaux dirigés et à la maison.
- 2 projets de plus grande envergure à faire seul ou par groupe de deux, lors de séances de travaux dirigés et à la maison.

Cours théorique :

- Compléments de programmation en C, en particulier sur l’écriture et l’organisation de programmes
- Compléments d’algorithmique (complexité, tri et recherche, diviser pour régner...)
- Introduction aux structures de données (pile, file, liste, arbre, table de hachage...)
- Feedback général sur les devoirs et projets

Organisation pratique

- Cours théoriques :
  - Un mardi sur deux de 13h45 à 16h45 maximum
  - Transparents disponibles sur la page web du cours avant chaque cours
- Travaux dirigés :
  - Un mardi sur deux de 13h45 à 16h45 maximum
  - Instructeurs : assistants et éléves-moniteurs
  - Exercices sur ordinateur ou sur feuille portant sur la matière théorique ou la réalisation des devoirs et projets. Enoncés disponibles sur la page web des projets.
- Projets :
  - Deux projets à réaliser pendant le semestre, seul ou en binôme (idéalement).

Modalités d’évaluation

- Devoirs : 10% (5% par devoir)
- Projets : 60% (30% par projet)
- Examen écrit : 30% (à livre ouvert, portant sur la matière du cours, des répétitions et les projets)
- En deuxième session :
  - les projets non rendus ou ratés devront être refaits
  - si les devoirs ou projets n’ont pas été rendus en première session, la pondération de l’examen écrit passe à 60%.
- Le cours étant fortement basés sur les devoirs et projets, pas de report de cote d’une année à l’autre

Critères d’évaluation des devoirs et projets

Trois critères principaux :

- Exactitude du code : testée automatiquement, en partie lors de la soumission des projets sur la plateforme
- Style et utilisation du langage : vérification rapide, menant à un cotation (presque) binaire : ok ou pas.
- Rapport (projets uniquement) : précision et qualité des réponses

Un feedback global sur les devoirs et projets sera donné lors de certains cours théoriques. Un feedback plus spécifique pourra être obtenu sur demande.

Plagiat

- La collaboration entre étudiants sur le cours théorique, la programmation C, l'utilisation d'outils et les exercices de répétition est fortement encouragée
- Discuter des concepts généraux liés aux devoirs et aux projets entre étudiants est permis.
- **Montrer son code à d'autres étudiants, regarder/copier, même partiellement, le code d'autres étudiants ou du code obtenu via d'autres sources est strictement interdit !**
- Des outils sophistiqués de détection de plagiat, robustes à des modifications de noms de variables ou des réarrangements de code, seront utilisés systématiquement.
- En cas de plagiat avéré (après convocation des étudiants incriminés), des sanctions seront appliquées et le président de jury sera informé.

Notes de cours

Les transparents des cours théoriques disponibles sur la page web du cours (un peu) avant chaque leçon.

Pas de livre de référence obligatoire mais les livres suivants ont été utilisés pour préparer le cours :

- C programming : a modern approach, K.N. King, W. W. Norton & Company, Second edition, 2008.
- Le langage C - Norme ANSI, Kernighan et Ritchie, Dunod, 2ème édition, 2000.
- Computer science : an interdisciplinary approach, Sedgewick et Wayne, Pearson, 2016.
- Introduction to algorithms, Cormen, Leiserson, Rivest, Stein, MIT press, Third edition, 2009.

Contenu du cours

Partie 1:	Rappel de C
Partie 2:	Construction d'algorithmes
Partie 3:	Organisation de programmes
Partie 4:	Complexité
Partie 5:	Tri et recherche
Partie 6:	Structures de données
Partie 7:	Langages de programmation

Partie 1

Rappel de C

23 décembre 2019

Plan

1. Un tour rapide du C via un exemple
2. Rappel sur les pointeurs

Plan

1. Un tour rapide du C via un exemple
2. Rappel sur les pointeurs

Illustration : l'ensemble de Mandelbrot

En mathématique, l'ensemble de Mandelbrot $\mathcal{M}$ est défini comme l'ensemble des points $c \in \mathbb{C}$ du plan complexe pour lesquels la suite de nombres définie par la récurrence suivante :

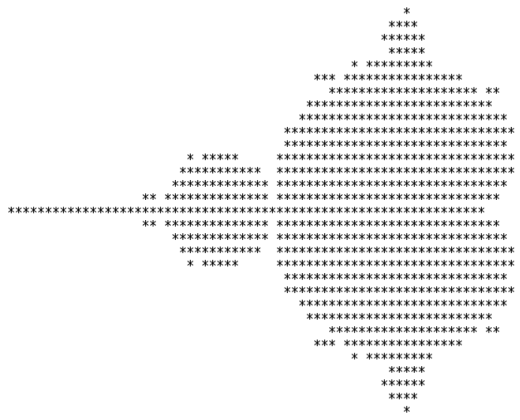
$$\begin{cases} z_0 = 0, \\ z_{n+1} = z_n^2 + c \; (n > 0). \end{cases}$$

est bornée.

Étudié depuis le début du vingtième siècle en dynamique complexe mais représenté pour la première fois à la fin des années septante grâce à l'ordinateur.

On aimerait implémenter un programme en C permettant de visualiser cet ensemble dans le plan complexe.

La première représentation de l'ensemble (en 1978)



https://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot

Vérifier l'appartenance à $\mathcal{M}$

Impossible à vérifier formellement mais on peut néanmoins démontrer que si la suite des modules devient strictement supérieure à 2 pour un certain indice n, alors, la suite est croissante et tend vers l'infini à partir de cet indice.

On peut donc calculer un surensemble de $\mathcal{M}$ en testant la contrainte $|z_n| \leq 2$ pour des valeurs aussi grande que possible de n.

En pratique, on est obligé de se limiter à un nombre maximum N d'itérations, si on veut implémenter ce test empiriquement sur ordinateur.

Notre programme vérifiera donc en fait l'appartenance à l'ensemble $\mathcal{M}' \subseteq \mathcal{M}$ défini comme suit :

$$\mathcal{M}' = \{c \in \mathbb{C} | \forall n, 0 \leq n \leq N : |z_n| \leq 2\}$$

Vérifier l'appartenance à $\mathcal{M}'$

L'idée du programme est donc de calculer z_n pour des valeurs de n allant de 0 à N, en s'arrêtant dès que $|z_n| > 2$.

A priori, on ne peut pas manipuler des nombres complexes directement en C mais la suite peut se réécrire comme suit sur base des parties réelles et imaginaires :

$$\begin{cases} \mathrm{Re}(z_0) = 0, \mathrm{Im}(z_0) = 0, \\ \mathrm{Re}(z_{n+1}) = \mathrm{Re}(z_n)^2 - \mathrm{Im}(z_n)^2 + \mathrm{Re}(c), \\ \mathrm{Im}(z_{n+1}) = 2\mathrm{Re}(z_n)\mathrm{Im}(z_n) + \mathrm{Im}(c), \end{cases}$$

où $\mathrm{Re}(z)$ et $\mathrm{Im}(z)$ désigne resp. les parties réelle et imaginaire d'un complexe z.

Vérifier l'appartenance à $\mathcal{M}'$ (en C) 1/3

En supposant que les variables `cr` et `ci` (double) contiennent les parties imaginaires et réelles de c, le code suivant vérifie la non-appartenance de c à $\mathcal{M}$:

```c
double zr = 0;
double zi = 0;
int n = 0;

while((n < N) && (zr*zr + zi*zi <= 4.0)) {
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
  n++;
}
if (zr*zr + zi*zi <= 4.0)
  printf("c belongs to M'\n");
else
  printf("c does not belong to M'\n");
```

Quel est l'invariant de cette boucle ?

Vérifier l'appartenance à $\mathcal{M}'$ (en C) 2/3

En utilisant un `for` à la place d'un `while`.

```c
double zr = 0;
double zi = 0;

for(int n = 0; (n < N) && (zr*zr + zi*zi <= 4.0); n++) {
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
}
if (zr*zr + zi*zi <= 4.0)
  printf("c belongs to M'\n");
else
  printf("c does not belong to M'\n");
```

Rappel de C : déclaration et types primitifs

```c
int a = 1, b, c;
float e;
```

- Toute variable doit être déclarée en spécifiant son type et peut être initialisée au moment de sa déclaration
- Type primitif :

`bool`	true ou false (ISO-C99, avec stdbool.h)
`char`	caractère signé
`int`	entier signé
`size_t`	entier non-signé représentant une taille ou un indice
`float`	nombre réel (précision simple)
`double`	nombre réel (précision double)

- Le typage du C est statique (le type d'une variable est déterminé à la compilation) et faible (une valeur peut être convertie implicitement vers le type adéquat).

Rappel de C : Opérateurs

(par ordre de précédence)

postfixe	`[] . -> expr++ expr--`
préfixe	`++expr --expr +expr -expr ~ !`
	`&expr *expr sizeof (type)expr`
multiplicatifs	`* / %`
additifs	`+ -`
décalages	`<< >>`
comparaisons	`< > <= >=`
égalité	`== !=`
ET binaire	`&`
OU exclusif binaire	`^`
OU binaire	`|`
ET logique	`&&`
OU logique	`||`
conditionnel	`?:`
affectations	`= += -= *= /= %= <<= >>= &= ^= |=`

Rappel de C : choix conditions

Choix binaire

```c
if (expr) {
    ...
}

if (expr) {
    ...
} else {
    ...
}
```

Choix multiple

```c
switch(expr) {
   case const1 : instr1 break;
   case const2 : instr2 break;
   ...
   default : instr
}
```

Expression conditionnelle

```c
expr1 ? expr2 : expr3;
```


Rappel de C : boucles

```c
for (expr1; expr2; expr3) {
    ...
}

while (expr) {
    ...
}

do {
    ...
} while (expr);
```

Interruption :

- L'instruction `break` permet de quitter la boucle courante.
- L'instruction `continue` permet de passer à l'itération suivante, sans exécuter le restant de l'itération courante.

Rappel de C : entrées-sorties

```
#include <stdio.h>
...
int a, b;

printf("Entrez une première valeur: ");
scanf("%d", &a);
printf("Entrez une seconde valeur: ");
scanf("%d", &b);

printf("%d + %d = %d\n", a, b, a + b);
...
```

- `printf` prend comme premier argument une chaîne formatée. Les arguments suivant sont les valeurs affectées aux spécificateurs de format (c.f. http://en.wikipedia.org/wiki/Printf_format_string#Format_placeholders pour une specification complète).
- `scanf` permet d'entrer une valeur au clavier et de la stocker à l'adresse spécifiée. Attention : la gestion propre des erreurs est difficile !

Vérifier l'appartenance à $\mathcal{M}'$ (en C) 3/3

On peut emballer ce code dans une fonction

```
int mandelbrotSet(double cr, double ci) {
  double zr = 0;
  double zi = 0;
  int n = 0;

  while(n < N && ((zr*zr + zi*zi) <= 4.0)) {
    double temp;
    temp = zr*zr - zi*zi + cr;
    zi = 2*zr*zi + ci;
    zr = temp;
    n++;
  }
  return (zr*zr + zi*zi <= 4.0);
}
```


Rappel de C : fonctions et procédures

```
int fct1(int a, int b) {
   ...
   return 4;
}
int fct2(void) {
   int a,b;
   ...
   return a+b;
}
void fct3(float b) {
   ...
   [return;]
}
```

- Une fonction peut prendre zéro, un ou plusieurs arguments.
- Les arguments sont passés par valeur.
- Chaque fonction renvoie une valeur d'un type donné, ou void.
- Si une fonction renvoie une valeur, elle doit posséder un instruction `return` correspondant au bon type.

Un programme C complet pour visualiser l'ensemble

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

#define N 1000 // Maximum number of iterations
#define W 1000 // Width/height of plot (pixels)

static int mandelbrotSet(double xc, double yc);

int main(int argc, char *argv[]) {
  double x = atof(argv[2]);
  double y = atof(argv[3]);
  double size = atof(argv[4]);

  FILE *fp = fopen(argv[1],"w");

  for (int i = 0; i < W; i++) {
    for (int j = 0; j < W; j++) {
      double cr = x - size/2 + size*j/W;
      double ci = y + size/2 - size*i/W;
      fprintf(fp,"%d ",mandelbrotSet(cr,ci));
    }
    fprintf(fp,"\n");
  }

  fclose(fp);

  return 0;
}
```

```
static int mandelbrotSet(double cr, double ci) {
  double zr = 0;
  double zi = 0;
  int n = 0;

  while(n < N && ((zr*zr + zi*zi) <= 4.0)) {
    double temp;
    temp = zr*zr - zi*zi + cr;
    zi = 2*zr*zi + ci;
    zr = temp;
    n++;
  }
  return n;
}
```

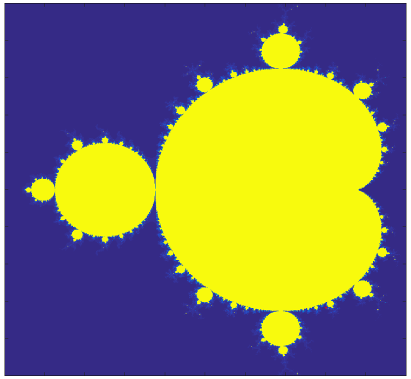
Utilisation :

```
> gcc mandelbrot.c -o mandelbrot
> ./mandelbrot m.amat -0.5 0.0 2.0
```

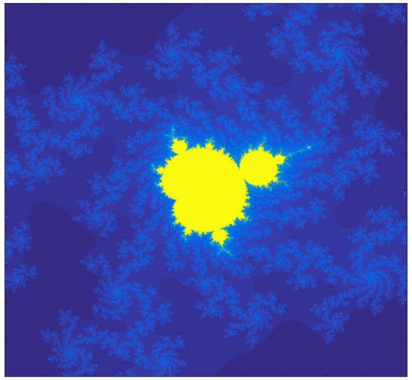
Pour voir l'image avec matlab

```
> M = load('m.amat');
> imagesc(M);
```

L'ensemble de Mandelbrot



./mandelbrot m.amat -0.5 0.0 2.0



./mandelbrot m.amat 0.1015 -0.633 0.001

Pour en voir plus : http://tilde.club/~david/m

Rappel de C : fonction main

```
int main() {
    ...
    return 0;
}
```

```
int main(int argc, char *argv[]) {
    ...
    return 0;
}
```

- Tout programme doit contenir une fonction main.
- La valeur de retour (entière) est rendue au système d'exploitation. Par convention, 0 signifie que tout s'est bien passé.
- On peut récupérer les paramètres passés au programme au moment de l'exécution en ajoutant deux arguments à la fonction :
  - `argc` : le nombre de paramètres passés (y compris le nom de l'exécutable)
  - `argv` : un tableau de chaînes de caractères dont les éléments contiennent ces arguments
- Ces arguments peuvent être analysés à l'aide des fonctions de la libraire `stdlib` (p.ex., `atoi`, `atof`).

Rappel de C : tableaux

```
int[5] a;
a[0] = 1;
a[4] = 42;
a[-1] = 10; // Bug!
a[5] = -5;  // Bug!

char id[] = "texte"; // Chaîne de caractères

int mat[3][4]; // Tableau multidimensionel
```

- Un tableau est un type de données indexable contenant des éléments du même type.
- Les éléments sont indexés à partir de 0 et jusqu'à N-1.
- Les tableaux sont passés aux fonctions par pointeurs (voir plus loin). Leurs modifications sont donc répercutées à l'appelant.
- Une chaîne de caractère est un tableau de `char` terminé par un caractère null `'\0'`.

Rappel de C : organisation d'un programme

Organisation possible d'un programme C en un seul fichier (voir `mandelbrot.c`) :

- Directive d'inclusion (`#include`)
- Définition de constantes et macro (`#define`)
- Définitions de types (`typedef`, voir plus loin)
- Déclarations de variables globales
- Prototypes des fonctions autres que la fonction `main`
- Définition de la fonction `main`
- Définition des autres fonctions

Seul contrainte forte : toute fonction/variable/constante doit être définie avant d'être utilisée.

Autre variante utilisant un nouveau type 1/2

On peut simplifier le code principal et améliorer sa lisibilité en définissant un nouveau type de données pour les nombres complexes.

complex.h

```c
typedef struct {
  double re, im;
} complex;

complex complex_new(double, double);
complex complex_sum(complex, complex);
complex complex_product(complex, complex);
double  complex_modulus(complex);
...
```

complex.c

```c
#include <math.h>
#include "complex.h"

complex complex_new(double re, double im) {
  complex c;
  c.re = re;
  c.im = im;
  return c;
}
complex complex_sum(complex a, complex b) {
  complex c;
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}
complex complex_product(complex a, complex b) {
  complex c;
  c.re = a.re * b.re - a.im * b.im;
  c.im = a.re * b.im + a.im * b.re;
  return c;
}
double complex_modulus(complex c) {
  return sqrt(c.re*c.re + c.im*c.im);
}
...
```


Rappel de C : structures

```c
// définition d'une structure
struct complex_t {
   double re, im;
};
// Définition d'un nouveau type
typedef struct {
   double re, im;
} complex;

// Utilisation
struct complex_t a, c = {1.2, 3.4};
complex b = {.im = 1.0, .re = 2.0};
b.re = 1.0;
a.im = 3.4;
```

- Une structure est un type de données composé, dont les éléments peuvent être de types différents.
- Les éléments de la structure sont accessibles par leurs noms via l'opérateur '.'.

Autre variante utilisant un nouveau type 2/2

```c
#include <stdio.h>
#include <stdlib.h>
#include "complex.h"

#define N 1000 // Maximum number of iterations
#define W 1000 // Width/Height of the plot

static int mandelbrotSet(complex c);

int main(int argc, char *argv[]) {
  double x = atof(argv[2]);
  double y = atof(argv[3]);
  double size = atof(argv[4]);

  FILE *fp = fopen(argv[1],"w");

  for (int i = 0; i < W; i++) {
    for (int j = 0; j < W; j++) {
      double cr = x - size/2 + size*j/W;
      double ci = y + size/2 - size*i/W;
      complex c = complex_new(cr, ci);
      fprintf(fp,"%d ",mandelbrotSet(c));
    }
    fprintf(fp,"\n");
  }

  fclose(fp);

  return 0;
}
```

```c
static int mandelbrotSet(complex c) {
  complex z = complex_new(0,0);
  int n = 0;
  while ((n < N) && (complex_modulus(z) <= 2.0)) {
    z = complex_plus(complex_product(z,z),c);
    n++;
  }
  return n;
}
```

Pour compiler :

```
> gcc mandlebrot.c complex.c -o mandelbrot
```


Autre variante utilisant la librairie `complex` du C99

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

#define N 1000 // Maximum number of iterations
#define W 1000 // Width/Height of the plot

static int mandelbrotSet(double _Complex c);

int main(int argc, char *argv[]) {
  double x = atof(argv[2]);
  double y = atof(argv[3]);
  double size = atof(argv[4]);

  FILE *fp = fopen(argv[1],"w");

  for (int i = 0; i < W; i++) {
    for (int j = 0; j < W; j++) {
      double cr = x - size/2 + size*j/W;
      double ci = y + size/2 - size*i/W;
      fprintf(fp,"%d ",mandelbrotSet(cr+I*ci));
    }
    fprintf(fp,"\n");
  }

  fclose(fp);

  return 0;
}
```

```c
static int mandelbrotSet(double _Complex c) {
  double _Complex z = 0;
  int n = 0;
  while ((n < N) && (cabs(z) <= 2.0)) {
    z = z*z+c;
    n++;
  }
  return n;
}
```

Pour compiler :

```
> gcc mandlebrot.c -o mandelbrot
```

Le langage C

Le C est un langage de bas niveau (très proche du matériel) qui est très permissif (peu de choses sont interdites).

Avantages : efficacité, puissance, flexibilité.

Inconvénients : code souvent sujet aux erreurs (bugs) et parfois difficile à comprendre et à maintenir.

Conseil pour ce cours : éviter les “trucs” de programmation et privilégier la lisibilité à la compacité, voire l'efficacité[1], du code.

1. à complexité constante, cf. Partie 3

Un mauvais exemple

Que fait ce code ?

```c
#include <stdio.h>
main() {
   float C,l,c,o,O,I=-20;char _;for(;I++<20;puts(""))
   for(O=-46;O<14;putchar(_?42:32),O++)for(C=l=_=0;o
   =l*l,c=C*C,l=2*C*l+I/20,C=c-o+O/20,o+c<4&&++_;);
}
```

Source : https://www.codeproject.com/Articles/2228/Obfuscating-your-Mandelbrot-code

Un mauvais exemple

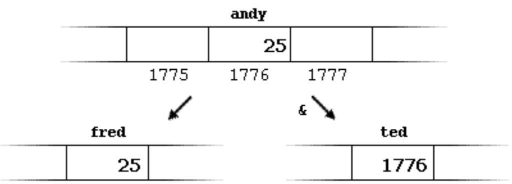
Plan

1. Un tour rapide du C via un exemple
2. Rappel sur les pointeurs

Variables et adresses

- L'identifiant d'une variable correspond à un emplacement mémoire, situé à une certaine adresse, contenant une valeur d'un certain type.
- Un pointeur est une variable dont la valeur est une adresse.
- Le type d'un pointeur est le type de la valeur pointée suivi de `*` (e.g., `int*` pour un pointeur vers un entier).

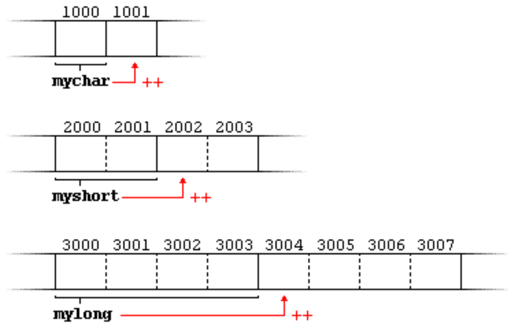
```
int andy = 25;
int fred = andy;
int *ted = &andy; // & dénote l'adresse de la variable andy
```



Arithmétique sur les pointeurs

- L'addition et la soustraction sont autorisées sur des pointeurs.
- $p+1$ correspond à l'emplacement mémoire suivant `p`, du même type.
- $p+n$ correspond au `n`-ème emplacement mémoire après `p`, du même type.

```
char* mychar;
short* myshort;
long* mylong;
mychar = mychar + 1;
myshort++;
mylong++;
```



Tableaux et pointeurs

L'identifiant d'un tableau est équivalent à un pointeur pointant vers le premier élément de ce tableau.

```
int a[5];
int* p;

p = a;
a[0] = 10;
*p = 10;        // Ces deux expressions sont équivalentes
a[2] = 42;
*(p + 2) = 42;  // Ces deux expressions sont aussi équivalentes
```


Où est le bug ?

```
#include <stdio.h>
int main() {
    int s[4], t[4];
    for (int i = 0; i <= 4; i++) {
        s[i] = t[i] = i;
    }
    printf("i:s:t\n");
    for (int i = 0; i <= 4; i++) {
        printf("%d:%d:%d\n", i, s[i], t[i]);
    }
    return 0;
}
```

Sortie :

```
i:s:t
0:4:0
1:1:1
2:2:2
3:3:3
4:4:4
```

Allocation / désallocation de mémoire

- Un bloc mémoire peut être alloué dynamiquement avec la fonction `malloc`.
- Renvoie `NULL` (=0) en cas d'échec, qui représente un pointeur vers rien.
- Tout bloc alloué dynamiquement doit être libéré explicitement avec la fonction `free`.

```
int* p = malloc(sizeof(int)); // Alloue un bloc de la taille d'un int
if (!p) {                      // Toujours vérifier le succès de malloc
    printf("Error");
    return 1;
}
free(p);                       // On libère le bloc

int* q = malloc(10 * sizeof(int)); // Alloue un bloc pour 10 int
q[0] = 42;
free(q);
```


Structures et pointeurs

```
typedef struct {
   double re,im;
} complex;
// cette fonction n'a aucun effet sur la donnée
void test1(complex a) {
   a.re = a.re + 1.0;
   a.im = a.im + 1.0;
}
// cette fonction modifie la donnée reçue en argument
void test1(complex *a) {
   a->re = a->re + 1.0;
   a->im = a->im + 1.0:
}
```

- Les structures sont passées (et renvoyées) par valeurs aux fonctions. En conséquence :
  - Les modifications des champs ne sont pas répercutées vers le code appelant.
  - Les structures sont copiées intégralement lors des appels de fonctions
- On manipule souvent les structures par l'intermédiaire des pointeurs.
- Si `p` est un pointeur vers une donnée possédant un champ `c`, alors la notation `p->c` est une abbréviation pour `(*p).c`.

Une ré-implémentation des complexes par pointeur

`complex.h`

```
typedef struct {
  double re, im;
} complex;

complex *complex_new(double, double);
double  complex_real_part(complex *);
double  complex_imgry_part(complex *);
void complex_sum(complex *, complex *);
void complex_difference(complex *, complex *);
void complex_product(complex *, complex *);
double  complex_modulus(complex *);
double  complex_distance(complex *, complex *);
```

`complex.c`

```
#include <math.h>
#include "complex.h"

complex *complex_new(double re, double im) {
  complex *c = malloc(sizeof(complex));
  c->re = re;
  c->im = im;
  return c;
}

void complex_sum(complex *a, complex *b) {
  a->re = a->re + b->re;
  a->im = a->im + b->im;
}

void complex_product(complex *a, complex *b) {
  double tmp_re, tmp_im;
  tmp_re = a->re * b->re - a->im * b->im;
  tmp_im = a->re * b->im + a->im * b->re;
  a->re = tmp_re;
  a->im = tmp_im;
}

...
```


Partie 2

Construction d'algorithmes

23 décembre 2019

Plan

Construction de programme

Pour résoudre un problème de programmation complexe, on le découpe généralement en sous-problèmes plus simples à appréhender

- *Exemples de sous-problèmes pour afficher l'ensemble de Mandelbrot :*
  - *SP1 : calculer le module d'un nombre complexe,*
  - *SP2 : vérifier l'appartenance d'un point à l'ensemble,*
  - *SP3 : produire l'image en parcourant le plan complexe.*

Ces problèmes dépendent généralement les uns des autres.

- *Dans l'exemple, SP3 dépend de SP2 qui dépend de SP1.*

Avantages d'un découpage :

- Facilite l'implémentation : on peut résoudre chaque sous-problème indépendamment,
- Généralité : on peut partager du code entre différents programmes,
- Lisibilité et facilité de maintenance du code.

Construction de programmes

Résoudre un sous-problème élémentaire :

- Certains sont triviaux et requièrent d'établir une séquence simple d'instructions (p.ex., calculer le module d'un nombre complexe)
- D'autres sont plus complexes et nécessitent de répéter une séquence d'instructions selon un schéma dépendant des données (p.ex., déterminer l'appartenance à l'ensemble de Mandelbrot)

Deux grands types de solutions algorithmiques pour ces derniers cas :

- Solutions **itératives**, basées sur des boucles
- Solutions **récursives**, basées sur des fonctions qui s'invoquent elles-mêmes

On va (re)voir dans cette partie quelques grands principes pour la conception de ces deux types de solutions.

Plan

Construction d'algorithmes itératifs

Concevoir un algorithme itératif (correct) peut être un exercice très compliqué, surtout si on cherche une solution efficace.

Deux difficultés principales :

- Imaginer le schéma itératif permettant de résoudre le problème.
- Générer le code implémentant ce schéma en évitant les bugs.

Le premier problème est de loin le plus compliqué et cette compétence s'acquière quasi uniquement via la pratique (cf. INFO0902 pour des techniques génériques néanmoins).

La technique de l'invariant permet d'aborder formellement le second problème une fois le schéma de la boucle imaginé.

Invariant de boucle

Un **invariant de boucle** est une propriété définie sur les variables du programme qui définit précisément ce qui doit être calculé à chaque itération pour arriver au résultat escompté. Il résume l'état courant des calculs.

Identifier l'invariant revient à imaginer le schéma itératif de résolution du problème et est parfois non trivial.

Une fois l'invariant établi, implémenter la boucle peut par contre se faire de manière relativement automatique.

L'utilisation de l'invariant permet donc d'éviter les erreurs d'implémentation.

Invariant de boucle

Une assertion est une relation entre les variables et les données utilisées par le programme qui est vraie à un moment donné lors de l'exécution du programme.

Deux assertions particulières :

- Pré-condition P : condition que doivent remplir les entrées valides du programme
- Post-condition Q : condition qui exprime que le résultat du programme est celui attendu.

On cherche donc à écrire un programme, noté S, dont l'exécution dans tous les cas où P est vraie mène à ce que Q soit toujours vraie.

Lorsque c'est le cas, on dira que le triplet $\{P\}S\{Q\}$ est correct.

Exemple : Si $P = \{x \geq 0\}$ et $Q = \{y^2 = x\}$, le code $S =$ "$y = \text{sqrt}(x);$" rend le triplet $\{P\}S\{Q\}$ correct.

Invariant de boucle : plus formellement

```
{P}
INIT
while (B)
    CORPS
FIN
{Q}
```

```
{P}
INIT
{I}
while (B)
    {I et B} CORPS {I}
{I et non B}
FIN
{Q}
```

Dans le cas où le programme nécessite une boucle :

- On met en évidence une assertion particulière I, l'invariant de boucle, qui décrit l'état du programme pendant la boucle.
- On détermine ensuite le gardien B et les codes *INIT*, *CORPS* et *FIN* tels que les trois triplets suivants soient corrects :
  - $\{P\}$ INIT $\{I\}$
  - $\{I$ et $B\}$ CORPS $\{I\}$
  - $\{I$ et non $B\}$ FIN $\{Q\}$

(Si on a plusieurs boucles imbriquées, on les traite séparément.)

Ecriture du code sur base de l'invariant

Trois parties de code à écrire en se basant sur l'invariant :

1. Initialisation : $\{P\}$ INIT $\{I\}$
   - INIT doit rendre l'invariant vrai avant de rentrer dans la boucle et en partant de la pré-condition.
   - L'invariant identifie les variables nécessaires et comment les initialiser.
2. Maintenance : $\{I$ et $B\}$ CORPS $\{I\}$
   - CORPS doit faire avancer le problème en maintenant l'invariant en supposant que le gardien soit vrai.
   - L'invariant définit le schéma de la boucle.
3. Terminaison : $\{I$ et non $B\}$ FIN $\{Q\}$
   - FIN doit rendre la post-condition vraie en supposant que l'invariant est vérifié et le gardien est faux.
   - L'invariant définit comment finalement résoudre le problème.

Détermination de l'invariant

Pas de recette miracle pour déterminer l'invariant (ou de manière équivalente le schéma d'une boucle).

Quelques trucs néanmoins :

- Enlever une partie de la post-condition
- Remplacer dans la post-condition une constante par une variable
- Combiner les pré-conditions et post-conditions
- Raisonner par induction (voir plus loin)

Dans tous les cas, l'invariant doit faire apparaître toutes les variables du programme.

Pour un même problème, plusieurs invariants (et/ou gardiens de boucles) sont possibles qui mèneront à différentes implémentations de la boucle.

Illustration 1 : appartenance à l'ensemble de Mandelbrot

Pré et post-conditions :

- $P = \{\mathtt{cr} \in \mathbb{R}, \mathtt{ci} \in \mathbb{R}\}$
- $Q = \{\mathtt{r} = 1 \text{ si } \forall n, 0 \leq n \leq N : |z_n| \leq 2, 0 \text{ sinon}\}$

où cr et ci sont les entrées du programme, r le résultat, N une constante, et z_n est la n-ème valeur de la suite définie précédemment avec $c \in \mathbb{C}$ tel que $c = \mathtt{cr} + i\mathtt{ci}$.

Schéma de la boucle :

- On calcule z_n pour des valeurs croissantes de n allant de 0 à N.
- A chaque itération, on calcule z_n à partir de la valeur z_{n-1} calculée à l'itération précédente.
- On s'arrête dès que $|z_n| > 2$.

Illustration 1 : appartenance à l'ensemble de Mandelbrot

Invariant :

$$I = \{(\forall n' : 0 \leq n' < \mathtt{n} : |z_{n'}| \leq 2) \text{ et } (\mathtt{zr} + i\mathtt{zi} = z_{\mathtt{n}}) \text{ et } (0 \leq \mathtt{n} \leq N)\},$$

où $\mathtt{n}$ est le compteur de boucle et $\mathtt{zr}$ et $\mathtt{zi}$ sont deux variables qui contiendront les résultats intermédiaires.

$$\overbrace{z_0, z_1, \ldots, z_{\mathtt{n}-1},}^{|\cdots|\leq 2} \overbrace{\underbrace{z_{\mathtt{n}}}_{\mathtt{zr}+i\mathtt{zi}=z_{\mathtt{n}}}, z_{\mathtt{n}+1}, \ldots, z_N}^{|\cdots|?}$$

Gardien :

$$B = \{\mathtt{n} < N, \mathtt{zr}^2 + \mathtt{zi}^2 \leq 4\}.$$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \ldots, z_{n-1}}^{|\cdots|\leq 2}, \overbrace{\underbrace{z_n}_{zr+i\,zi=z_n}, z_{n+1}, \ldots, z_N}^{|\cdots|?}$$

$\{P\}$ $\{cr \in \mathbb{R}, ci \in \mathbb{R}\}$

INIT

$\{I\}$

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

$\{I \text{ et } B\}$

CORPS

$\{I\}$

```
}
```

$\{I \text{ et non } B\}$

FIN

$\{Q\}$ $\{r = 1 \text{ si } \exists n : 0 \leq n \leq N : |z_n| > 2, 0 \text{ sinon}\}$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \ldots, z_{n-1}}^{|\cdots|\leq 2}, \overbrace{\underbrace{z_n}_{zr+i\,zi=z_n}, z_{n+1}, \ldots, z_N}^{|\cdots|?}$$

$\{P\}$ $\{cr \in \mathbb{R}, ci \in \mathbb{R}\}$

```
double zr = 0;
double zi = 0;
int n = 0;
```

$\{I\}$

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

$\{I \text{ et } B\}$

CORPS

$\{I\}$

```
}
```

$\{I \text{ et non } B\}$

FIN

$\{Q\}$ $\{r = 1 \text{ si } \exists n : 0 \leq n \leq N : |z_n| > 2, 0 \text{ sinon}\}$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \ldots, z_{n-1}}^{|\cdots|\leq 2}, \overbrace{\underbrace{z_n}_{zr+i\,zi=z_n}, z_{n+1}, \ldots, z_N}^{|\cdots|?}$$

$\{P\}$ $\{cr \in \mathbb{R}, ci \in \mathbb{R}\}$

```
double zr = 0;
double zi = 0;
int n = 0;
```

$\{I\}$

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

$\{I \text{ et } B\}$

```
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
  n++;
```

$\{I\}$

```
}
```

$\{I \text{ et non } B\}$

FIN

$\{Q\}$ $\{r = 1 \text{ si } \exists n : 0 \leq n \leq N : |z_n| > 2, 0 \text{ sinon}\}$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \ldots, z_{n-1}}^{|\cdots|\leq 2}, \overbrace{\underbrace{z_n}_{zr+i\,zi=z_n}, z_{n+1}, \ldots, z_N}^{|\cdots|?}$$

$\{P\}$ $\{cr \in \mathbb{R}, ci \in \mathbb{R}\}$

```
double zr = 0;
double zi = 0;
int n = 0;
```

$\{I\}$

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

$\{I \text{ et } B\}$

```
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
  n++;
```

$\{I\}$

```
}
```

$\{I \text{ et non } B\}$

```
int r = (zr*zr+zi*zi <= 4.0);
```

$\{Q\}$ $\{r = 1 \text{ si } \exists n : 0 \leq n \leq N : |z_n| > 2, 0 \text{ sinon}\}$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

```c
double zr = 0;
double zi = 0;
int n = 0;
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
  n++;
}
int r = (zr*zr+zi*zi <= 4.0);
```

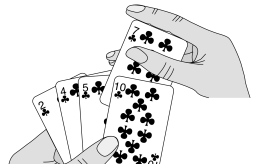

Illustration 2 : tri par insertion

On souhaite écrire une fonction pour trier un tableau A de valeurs entières. Le tri doit être effectué dans le tableau lui-même, via échanges d'éléments.

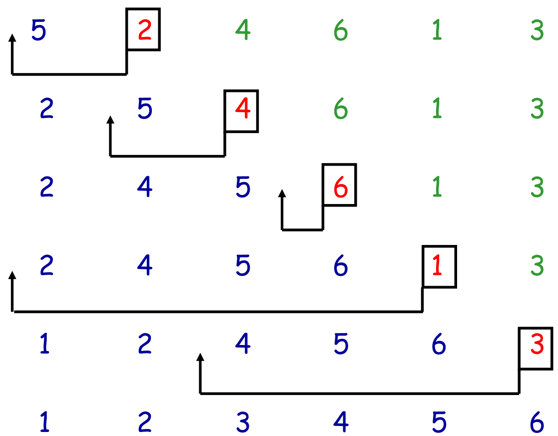
```c
void insertion_sort(int A[], int N);
```

Principe de l'algorithme :

- On parcourt le tableau de gauche à droite en triant successivement les préfixes du tableau de tailles 2, 3, ..., N.
- A chaque itération, on augmente la taille du préfixe trié en insérant le nouvel élément $A[i]$ à sa position dans le sous-tableau $A[0..i-1]$ précédemment ordonné.



Tri par insertion : graphiquement

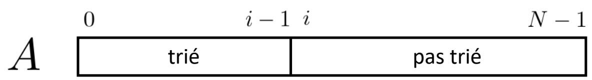


Tri par insertion : boucle externe

Pré-condition P_e : A est un tableau d'entiers de taille N.

Post-condition Q_e : A contient les éléments du tableau de départ triés par ordre croissant.

Invariant I_e (de la boucle externe) : Le sous-tableau $A[0..i-1]$, avec $1 \leq i \leq N$, contient les i premiers éléments du tableau initial triés par ordre croissant.



Gardien B_e : $i < N$.

Tri par insertion : boucle externe

$\{P_e\}$ $\{A$ est un tableau d'entiers de taille $N\}$

INITe

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
while (i<N) {
```

$\{I_e$ et $B_e\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

CORPSe

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
}
```

$\{I_e$ et non $B\}$ $\{A[0\,..\,i-1]$ trié et $i = N\}$

FINe

$\{Q_e\}$ $\{A$ contient les éléments du tableau de départ triés$\}$

Tri par insertion : boucle externe

$\{P_e\}$ $\{A$ est un tableau d'entiers de taille $N\}$

```
int i = 1;
```

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
while (i<N) {
```

$\{I_e$ et $B_e\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

CORPSe

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
}
```

$\{I_e$ et non $B\}$ $\{A[0\,..\,i-1]$ trié et $i = N\}$

FINe

$\{Q_e\}$ $\{A$ contient les éléments du tableau de départ triés$\}$

Tri par insertion : boucle externe

$\{P_e\}$ $\{A$ est un tableau d'entiers de taille $N\}$

```
int i = 1;
```

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
while (i<N) {
```

$\{I_e$ et $B_e\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

CORPSe

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
}
```

$\{I_e$ et non $B\}$ $\{A[0\,..\,i-1]$ trié et $i = N\}$

```
-
```

$\{Q_e\}$ $\{A$ contient les éléments du tableau de départ triés$\}$

Tri par insertion : boucle externe

$\{P_e\}$ $\{A$ est un tableau d'entiers de taille $N\}$

```
int i = 1;
```

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
while (i<N) {
```

$\{I_e$ et $B_e\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

CORPSe'

$\{Q_i\}$ $\{A[0\,..\,i]$ trié$\}$

```
  i++;
```

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
}
```

$\{I_e$ et non $B\}$ $\{A[0\,..\,i-1]$ trié et $i = N\}$

```
-
```

$\{Q_e\}$ $\{A$ contient les éléments du tableau de départ triés$\}$

Tri par insertion : boucle interne (CORPSe')

Pré-condition P_i : $\{I_e$ et $B_e\} = \{A[0..i-1]$ trié et $i < N\}$

Post-condition Q_i : $I_e = \{A[0..i]$ trié$\}$

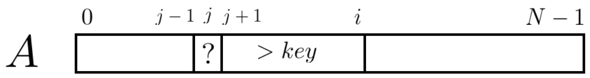
Idée de la boucle : Soit $key = A[i]$, la valeur à déplacer :

- On parcourt le sous-tableau $A[0..i-1]$ de droite à gauche.
- Tant que les éléments parcourus sont supérieurs à key, on les décale d'une position vers la droite.
- On insére key à la position finalement atteinte.

Tri par insertion : boucle interne

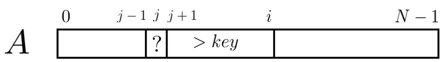
Invariant I_i : Soit j un nouvel indice et $key = A[i]$ la valeur à insérer :

- $A[0..j-1]$ et $A[j+1..i]$ sont triés par ordre croissant et ensemble contiennent tous les éléments du sous-tableau $A[0..i]$ initial, excepté key.
- $key < A[j+1]$



Gardien B_i : $\{j > 0$ et $A[j-1] > key\}$

Tri par insertion : boucle interne



$\{P_i\}$ $\{A[0..i-1]$ trié et $i < N\}$

INITi

$\{I_i\}$

```
while (j > 0 && A[j-1] > key) {
```

  $\{I_i$ et $B_i\}$

  CORPSi

  $\{I_i\}$

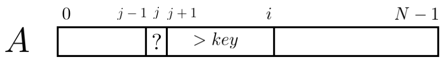
```
}
```

$\{I_i$ et non $B_i\}$

FINi

$\{Q_i\}$ $\{A[0..i-1]$ trié$\}$

Tri par insertion : boucle interne



$\{P_i\}$ $\{A[0..i-1]$ trié et $i < N\}$

```
int key = A[i];
int j = i;
```

$\{I_i\}$

```
while (j > 0 && A[j-1] > key) {
```

  $\{I_i$ et $B_i\}$

  CORPSi

  $\{I_i\}$

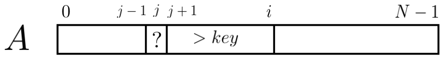
```
}
```

$\{I_i$ et non $B_i\}$

FINi

$\{Q_i\}$ $\{A[0..i-1]$ trié$\}$

Tri par insertion : boucle interne



$\{P_i\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

```
int key = A[i];
int j = i;
```

$\{I_i\}$

```
while (j > 0 && A[j-1] > key) {
```

$\{I_i$ et $B_i\}$

```
  A[j] = A[j-1];
  j--;
```

$\{I_i\}$

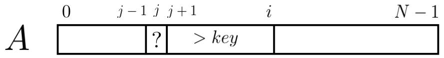
```
}
```

$\{I_i$ et non $B_i\}$

FINi

$\{Q_i\}$ $\{A[0\,..\,i-1]$ trié$\}$

Tri par insertion : boucle interne



$\{P_i\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

```
int key = A[i];
int j = i;
```

$\{I_i\}$

```
while (j > 0 && A[j-1] > key) {
```

$\{I_i$ et $B_i\}$

```
  A[j] = A[j-1];
  j--;
```

$\{I_i\}$

```
}
```

$\{I_i$ et non $B_i\}$

```
A[j] = key;
i++;
```

$\{Q_i\}$ $\{A[0\,..\,i-1]$ trié$\}$

Tri par insertion : code complet

```
void insertion_sort(int A[], int N) {
  int i = 1;
  while (i < N) {
    int key = A[i];
    int j = i;
    while (j > 0 && A[j-1]>key) {
      A[j] = A[j-1];
      j--;
    }
    A[j] = key;
    i++;
  }
}
```


Synthèse

- L'expression d'un invariant permet de limiter les erreurs lors de l'implémentation d'une boucle.
- Vous devez prendre l'habitude d'exprimer l'invariant de toutes vos boucles **avant** leur implémentation, au minimum de manière informelle ou graphique.
- Dans la suite du cours, on fournira ponctuellement les invariants des boucles les plus compliquées.

Plan

Preuve de correction d'algorithmes itératifs

- Dans certains contextes, il est crucial de prouver formellement qu'un programme est correct (p.ex. dans le domaine médical ou en aéronautique).
- On peut toujours tester son programme empiriquement mais en général, il est impossible de considérer tous les cas possibles d'utilisation d'un code.

  Testing can only show the presence of bugs, not their absence
  E.W. Dijkstra

- L'analyse de correction de triplets et la technique de l'invariant de boucle peuvent aussi être utilisées pour prouver formellement qu'un algorithme itératif est correct (voir INFO2009).
- On peut automatiser une grosse partie de ces analyses, mais pas la dérivation des invariants de boucle.

Terminaison de boucle

Prouver qu'un triplet $\{P\}S\{Q\}$ est correct n'est pas suffisant dans le cas d'une boucle.

Il faut encore prouver que la boucle se termine.

Exemple : On peut prouver que le triplet ci-dessous est correct mais la boucle ne se termine pas toujours. On dira que le code est partiellement correct.

$\{\texttt{cr} \in \mathbb{R}, \texttt{ci} \in \mathbb{R}\}$

```
double zr = 0;
double zi = 0;

while(zr*zr + zi*zi <= 4.0)) {
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
}
int r = (zr*zr+zi*zi <= 4.0);
```

$\{\texttt{r} = 1 \text{ si } \forall n \geq 0 : |z_n| \leq 2, 0 \text{ sinon}\}$

Terminaison de boucle

Pour prouver qu'une boucle se termine, on cherche une fonction de terminaison f :

- définie sur base des variables de l'algorithme et à valeur entière naturelle (≥ 0)
- telle que f décroît strictement suite à l'exécution du corps de la boucle
- telle que B implique $f > 0$

Puisque f décroit strictement, elle finira par atteindre 0 et donc à infirmer B.

Exemple :

- Mandelbrot : $f = N - n$
- Tri par insertion (boucle externe) : $f = N - i$

Terminaison de boucle

Il n'est pas toujours trivial de prouver la terminaison d'une boucle.

Personne n'a pu encore prouvé que la boucle suivante se terminait pour tout $n > 1$, bien qu'on l'ait prouvé empiriquement pour toutes les valeurs de $N < 1,25.2^{62}$.

```
void Algo(int n) {
  while(n != 1) {
    if (n % 2) // n est impair
      n = 3*n+1;
    else      // n est pair
      n = n/2;
  }
}
```

https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

Plan

Algorithme récursif

- Un algorithme de résolution d'un problème P sur une donnée a est dit récursif si parmi les opérations utilisées pour le résoudre, on trouve une résolution du même problème P sur une donnée b.
- Dans un algorithme récursif, on nommera appel récursif toute étape résolvant le même problème sur une autre donnée.
- Un algorithme récursif s'implémente généralement via des fonctions récursives.
- Forme générale d'une fonction récursive (directe) :

```
type f(P) {
  ...
  x = f(Pr);
  ...
  return r;
}
```


Exemple 1 : fonction factorielle

La définition mathématique de la factorielle est récursive :

$$n! = \begin{cases} 1 & \text{si } n \leq 1, \\ n * (n-1)! & \text{sinon} \end{cases}$$

et se prête donc naturellement à une implémentation via une fonction récursive :

```
int fact(int n) {
  if (n <= 1)
     return 1;

  return n * fact(n-1);
}
```

Exemple 1 : fonction factorielle

Trace des appels de fonctions pour `fact(5)` :

```
fact(5)
  |fact(4)
  |   |fact(3)
  |   |   |fact(2)
  |   |   |   |fact(1)
  |   |   |   |   |return 1
  |   |   |   |return 2*1 = 2
  |   |   |return 3*2 = 6
  |   |return 4*6 = 24
  |return 5*24 = 120
```


Exemple 2 : algorithme d'Euclide

L'algorithme du calcul du pgcd d'Euclide est basé sur la propriété récursive suivante :

> *Soient deux entiers positifs a et b. Si $a > b$, le pgcd de a et b est égal au pgcd de b et de $(a \mod b)$.*

Suggère l'implémentation récursive suivante :

```
int pgcd(int a, int b) {
  if (b > a)
    return pgcd(b,a);

  if (b == 0)
    return a;

  return pgcd(b, a % b);
}
```

Exemple :

```
pgcd(1440, 408)
  |pgcd(408, 216)
  |   |pgcd(216, 192)
  |   |   |pgcd(192, 24)
  |   |   |   |pgcd(24,0)
  |   |   |   |   |return 24
  |   |   |   |return 24
  |   |   |return 24
  |   |return 24
  |return 24
```


Conception de fonctions récursives

Deux conditions pour qu'une fonction récursive soit bien définie :

- Présence d'un cas de base (condition d'arrêt)
- La "taille" du problème doit être réduite à chaque étape

Forme générale d'une fonction récursive :

```
type f(P) {
  if (cas_de_base) { // cas de base
    ... // pas d'appel récursif
    return [...];
  }

  // étape de réduction
  ...
  [x =] f(Pr); // avec Pr plus "simple" que P
  ...
  [return r;]
}
```


Conception de fonctions récursives

D'autres formes peuvent néanmoins être valides (mais sont plutôt à éviter).

Récursivité non décroissante

(Suite de Syracuse)

```
int Syracuse(int n) {
  if (n == 1)
    return 1;

  if (n % 2) // n est impair
    return Syracuse(3*n+1);
  else       // n est pair
    return Syracuse(N/2);
}
```

Récursivité imbriquée

(fonction d'Ackermann)

```
int ack(int m, int n) {
  if (m==0)
    return n+1;
  else {
    if (n==0)
      return ack(m-1,1);
    else
      return ack(m-1,ack(m,n-1));
  }
}
```

Récursivité croisée

```
int P(int n) {
  if (n==0)
    return 1;
  else
    return I(n-1);
}

int I(int n) {
  if (n==0)
    return 0;
  else
    return P(n-1);
}
```

Que calcule la fonction P ?

Principe de construction d'un algorithme récursif

- Trouver un ou plusieurs paramètres de taille sur lesquels construire la récursion

  Factorielle : n, le nombre dont on veut calculer la factorielle.
- Trouver une solution pour le(s) cas de base, c'est-à-dire des problèmes de petites tailles (la plupart du temps trivial).

  Factorielle : $n! = 1$ si $n \leq 1$.
- Trouver comment réduire le problème à un ou plusieurs sous-problème de tailles strictement plus petites.

  Factorielle : $n! = n * (n-1)!$ si $n > 1$

La dernière étape est la plus délicate. Equivalent à trouver l'invariant dans le cas d'algorithmes itératifs.

Plan

Fonction puissance : solution itérative

On souhaite calculer a^x, avec $a \in \mathbb{R}$ et x entier ≥ 1.

Version itérative

```
float pow_iter(float a, int x) {
    float res = a;
    for (int i = 1; i < x; i++)
        res = res * a;
    return res;
}
```

(Invariant ?)

Nombre de multiplications nécessaires : $x - 1$

Fonction puissance : récursivement (1/2)

Une première formulation récursive :

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

```
float pow_rec1(float a, int x) {
    if (x == 1)
       return a;
    return a*pow_rec1(a, x-1);
}
```

Nombre de multiplications nécessaires : $x - 1$

Cette version est une simple réécriture de la version itérative.

Peut-on faire mieux ?

Fonction puissance : récursivement (2/2)

Une deuxième formulation récursive :

$$a^x = \begin{cases} a & \text{si } x = 1 \\ (a \cdot a)^{x/2} & \text{si } x > 1 \text{ et pair} \\ a \cdot (a \cdot a)^{(x-1)/2} & \text{si } x > 1 \text{ et impair} \end{cases}$$

```
float pow_rec2(float a, int x) {
    if (x == 1)
       return a;
    if (x % 2 == 0) // x pair
       return pow_rec2(a * a, x/2);
    else            // x impair
       return a * pow_rec2(a * a, (x-1)/2);
}
```

Nombre de multiplications nécessaires : entre $\log_2(x)$ et $2\log_2(x)$

- $x = 128 \Rightarrow 7$ multiplications au lieu de 127.

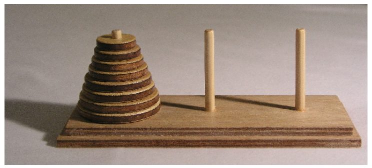
Fonction puissance : remarque

La solution précédente peut aussi s'implémenter de manière itérative.

Mais l'invariant de la boucle est obtenu sur base de la même formulation récursive et l'implémentation serait (un peu) moins immédiate.

Il existe des problèmes pour lesquels une solution itérative serait nettement plus complexe à implémenter qu'une solution récursive.

Le problème des tours de Hanoï cf. INFO2009



Source : wikipedia

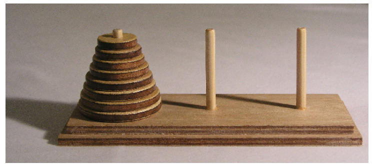
Objectif : transférer les n disques de la tour 1 à la tour 2, sans jamais mettre un disque sur un plus petit.

Solution récursive : Pour déplacer n disques :

- On déplace les $n-1$ disques du dessus vers la tour 3.
- On déplace le n-ème disque de la tour 1 vers la tour 2.
- On déplace les $n-1$ disques de la tour 3 vers la tour 2.

Solution itérative ?

Le problème des tours de Hanoï cf. INFO2009



Source : wikipedia

En fait, une solution itérative simple existe :

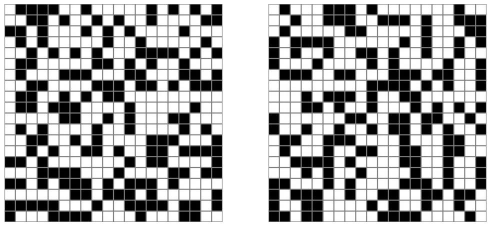
Tant que la pile n'est pas dans sa position finale :

- *Bouger le plus petit disque à droite* ($1 \to 2$, $2 \to 3$ *ou* $3 \to 1$)
- *Bouger le seul autre disque qui peut être bougé*

Cette solution bien que strictement identique à la solution récursive est beaucoup plus compliquée à concevoir à partir de rien.

Son implémentation est aussi plus complexe car elle demande de représenter explicitement l'état des trois tours à chaque itération.

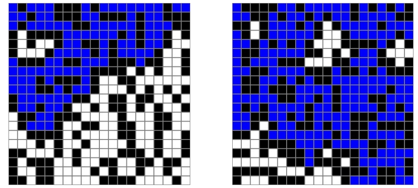
Percolation (Sedgewick et Wayne, 2016)



Problème : déterminer si de l'eau peut s'écouler dans un matériau poreux.

- Entrée : un tableau `grid` de taille $n \times n$, tel que $\texttt{grid}[i][j] = 0$ si le matériau est vide à la position (i, j), 1 si le matériau est plein.
- Sortie : vrai s'il existe un chemin partant de la première ligne, arrivant à la dernière ligne de la matrice et passant uniquement par des cases vides, faux sinon.

Percolation : découpage en sous-problème



Idée de la solution :

- Marquer chaque case de la grille pour laquelle il existe un chemin depuis une case vide de la première ligne.
- Vérifier qu'une case de la dernière ligne au moins a été marquée.

```c
// Valeur 2 utilisée comme marquage
int percolate(int **grid, int n) {
  flow(grid,n);   // fonction qui effectue le marquage des cases
  for (int j = 0; j < n; j++)
    if (grid[n-1][j] == 2)
      return 1;
  return 0;
}
```


Fonction `flow` : découpage en sous-problèmes

On définit une fonction `flowrec` :

```c
void flowrec(int **grid, int n, int i, int j);
```

qui va marquer toutes les cases accessibles à partir de la position (i, j).

La fonction `flow` consiste alors simplement à appliquer `flowrec` à toutes les cases de la première ligne du tableau.

```c
void flow(int **grid, int n) {
  for (int j = 0; j < n; j++)
     flowrec(grid, n, 0, j);
}
```


Une solution récursive pour `flowrec`

```c
void flowrec(int **grid, int n, int i, int j);
```

Cas de base : Ne rien faire si

- $\texttt{i} < 0$, $\texttt{i} \geq \texttt{n}$, $\texttt{j} < 0$, ou $\texttt{j} \geq \texttt{n}$,
  (la case est hors de la grille)
- $\texttt{grid[i][j]} == 1$,
  (la case est pleine)
- $\texttt{grid[i][j]} == 2$
  (la case a déjà été visitée lors d'un précédent appel récursif).

Cas inductif : Si on n'est pas dans un cas de base :

- On marque la case (i, j) qui est accessible.
- On applique récursivement la fonction `flowrec` à toutes les cases adjacentes à la case (i, j).

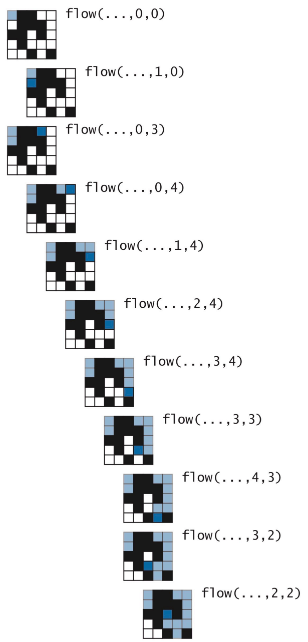
Une solution récursive pour `flowrec`

```c
void flow(int **grid, int n) {
  for (int j = 0; j < n; j++)
     flowrec(grid, n, 0, j);
}

void flowrec(int **grid, int n, int i, int j) {
  // Cas de base
  if (i < 0 || i >= n || j < 0 || j >= n) return;
  if (grid[i][j] == 1 || grid[i][j] == 2) return;
  // Cas inductif
  grid[i][j] = 2;
  flowrec(grid, n, i+1, j); // Bas
  flowrec(grid, n, i, j+1); // Droite
  flowrec(grid, n, i, j-1); // Gauche
  flowrec(grid, n, i-1, j); // Haut
}
```

Correction ? Complexité ?

Note : `flowrec` implémente ce qu'on appelle une recherche en profondeur d'abord (*depth-first search).*

Illustration

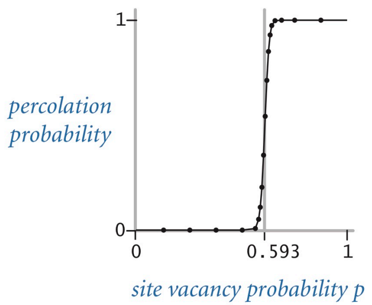


(Sedgewick et Wayne, 2016)

Application : simulations numériques

Cette fonction permet par exemple d'étudier l'impact de la porosité du matériau sur la probabilité de percolation.

Pour des grilles de taille 100x100 (10000 grilles aléatoires générées par point) :



(Sedgewick et Wayne, 2016)

Efficacité de codes récursifs

Dans le cas de la fonction puissance, des tours de Hanoï et de la percolation, la solution récursive donne un code optimal en terme de complexité.

Dans certains cas, l'utilisation naïve de la récursivité peut cependant mener à une solution très sous-optimale.

Exemple : calcul des nombres de Fibonacci

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ \forall n \geq 2 : F_n &= F_{n-2} + F_{n-1} \end{aligned}$$

Nombres de Fibonacci : implémentation récursive

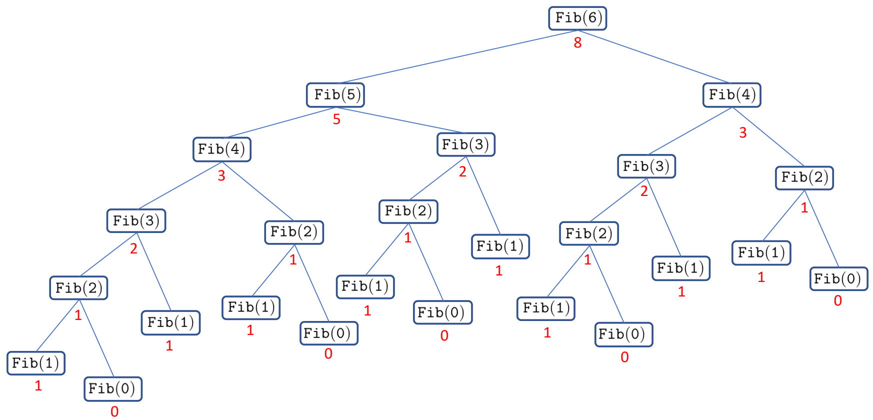
Implémentation récursive directe :

```
int Fib(int n) {
  if (n <= 1)
     return n;
  return Fib(n-1)+Fib(n-2);
}
```

Peut-on calculer Fib(60) ?

n	temps de calcul
10	0s
20	0s
40	1s
45	8s
50	87s
60	? ?

Nombres de Fibonacci : arbres des appels récursifs



Beaucoup de valeurs sont recalculées inutilement.

Nombres de Fibonacci : temps de calcul

Soit $T(n)$ le nombre de noeuds de l'arbre des appels récursifs. On a :

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 1 \\ \forall n \geq 2 : T(n) &= T(n-1) + T(n-2) + 1 \end{aligned}$$

Le nombre d'appels de fonctions est donc plus élevé que le $n$ème nombre de Fibonacci qui vaut approximativement $\frac{\phi^n}{\sqrt{5}}$, avec $\phi \approx 1,618$.

Chaque appel de fonction demandant un temps constant, les temps de calcul vont augmenter exponentiellement avec n. Ils sont multipliés par $1,618$ au moins à chaque incrément de n.

S'il faut 87s pour $n = 50$, il faudra au moins 3 heures pour $n = 60$ et > 77000 années pour $n = 100$.

Nombres de Fibonacci : version itérative

Une version itérative beaucoup plus efficace

```
int Fibiter(int n) {
  if (n <= 1)
     return n;
  else {
     int f;
     int pprev = 0;
     int prev = 1;

     for (int i = 2; i <= n; i++) {
        f = prev + pprev;
        pprev = prev;
        prev = f;
     }
     return f;
  }
}
```

n	Récursive	Itérative
10	0s	0s
20	0s	0s
40	1s	0s
45	8s	0s
50	87s	0s
60	3h	0,001s
100	77k ans	0,001s

Correction formelle d'algorithmes récursifs

```c
int fact(int n) {
  if (n <= 1)
     return 1;
  return n * fact(n-1);
}
```

La correction d'un algorithme récursif se prouve pas induction :

- On montre que le code est correct dans le cas de base.
  - Si $n \leq 1$, fact(n) renvoie 1, ce qui est correct.
- On montre que l'étape de réduction est correcte en supposant que les appels récursifs sont corrects (hypothèse inductive)
  - Si $n > 1$, la fonction renvoie $n * \texttt{fact}(\texttt{n} - 1)$, qui vaut bien $n!$ si fact(n-1) renvoie $(n-1)!$.
- On en conclut, par le *principe d'induction*, que l'algorithme est correct pour toutes les entrées.

Plan

Une expérience

Y-a-t'il une différence lors de l'exécution de ces deux codes ?

```c
int fact_rec(int n) {
   return n*fact_rec(n-1);
}
int main() {
   fact_rec(100);
   return 0;
}
```

```c
int fact_iter(int n) {
   int res = 1;
   for (;; n--)
      res = res*n;
   return res;
}
int main() {
   fact_iter(100);
   return 0;
}
```


Contextes d'appels de fonctions

A chaque appel de fonction, on doit retenir en mémoire (dans une zone appelée le Stack) le contexte de l'appel, c'est-à-dire :

- L'endroit où le code appelant doit continuer son exécution à la fin de l'appel
- La valeur des arguments fournis à la fonction
- La valeur des variables locales à la fonction

Cette information ne peut être supprimée qu'une fois l'appel terminé.

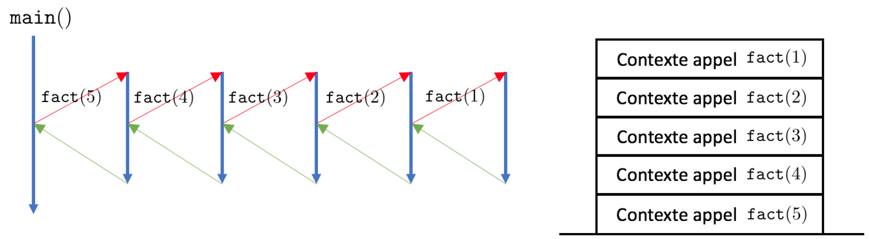
Dans le cas d'une fonction récursive, le nombre d'appels de fonction actifs à un moment donné peut être très important.

La récursivité a donc un coût mémoire dont il faut tenir compte.

Ce coût mémoire est directement proportionnel à la profondeur de l'arbre des appels récursifs.

Contextes d'appels de fonctions : illustration

```c
int fact(int n) {
   if (n == 1)
     return 1;
   return n*fact(n-1);
}
int main() {
   fact(5);
   return 0;
}
```



Une expérience : conclusion

Y-a-t'il une différence lors de l'exécution de ces deux codes ?

```c
int fact_rec(int n) {
   return n*fact_rec(n-1);
}
int main() {
   fact_rec(100);
   return 0;
}
```

⇒ Le programme s'arrête lorsque la mémoire qui lui est allouée est remplie.

```c
int fact_iter(int n) {
   int res = 1;
   for (;; n--)
      res = res*n;
}
int main() {
   fact_iter(100);
   return 0;
}
```

⇒ Le programme ne s'arrête jamais.

Récursivité terminale

Une fonction est récursive terminale s'il n'y a plus de calcul à effectuer une fois l'appel récursif terminé.

Exemple : le calcul du PGCD :

```c
int pgcd(int a, int b) {
  if (b > a)
    return pgcd(b,a);

  if (b == 0)
    return a;

  return pgcd(b, a % b);
}
```

Il est dans ce cas inutile de stocker le contexte des appels.

Les compilateurs modernes sont capables de détecter ce type de récursion (et d'autres plus compliquées) et de ne pas utiliser de mémoire inutile.

Synthèse

- Principal intérêt de la récursivité : élégance, simplicité, et lisibilité du code.
- Moins sujet à des bugs et analyse de correction facilitée par rapport aux boucles.
- Beaucoup d'algorithmes efficaces sont basés sur la récursivité (cf. Partie 5 et INFO0902).

- Mais l'utilisation naïve de la récursivité peut parfois mener à des solutions moins efficaces, voire très inefficaces.
- Il existe néanmoins des techniques systématiques pour améliorer l'efficacité d'une solution récursive (cf. INFO0902 et INFO0540).
- L'implémentation a souvent un coût non négligeable en terme d'espace mémoire dont il faut tenir compte.

Partie 3

Organisation de programmes

23 décembre 2019

Plan

Plan

Programmation modulaire : principes

Un programme (en C ou un autre langage) est souvent découpé en une série de modules indépendants.

Un module est une collection de services, mis à la disposition de clients, c'est-à-dire d'autres modules ou programmes.

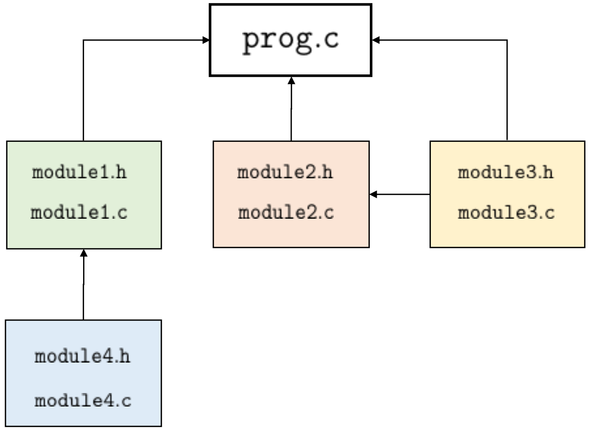
Chaque module a une interface qui décrit précisément les services qu'il fournit.

Les détails du module sont contenus dans son implémentation.

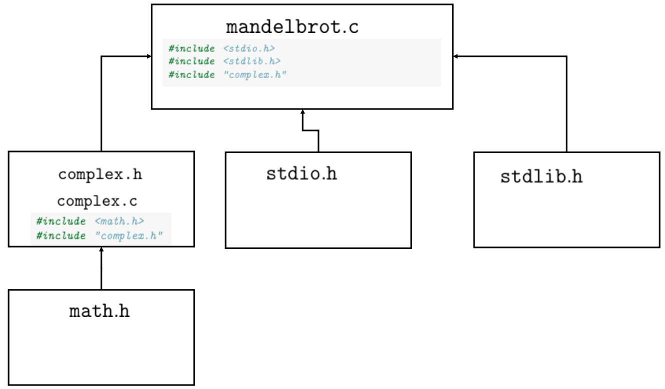
En C :

- Chaque service correspond à une fonction ou procédure.
- L'interface est précisée dans un fichier d'entête (`.h`).
- L'implémentation est fournie dans un fichier source (`.c`).

Programmation modulaire : illustration



Exemple : Mandelbrot



(Code : voir slides 34 et 35)

Avantages de l'approche modulaire

Abstraction :

- Chaque module peut être traité comme une abstraction : on sait ce qu'il fait mais on ne se tracasse pas de comment il le fait.
- Une fois que l'interface du module est bien précisée, on peut séparer les tâches d'implémentation module par module.

Réutilisabilité :

- Tout module est potentiellement réutilisable par d'autres clients.
- Les modules peuvent (devraient) être designés avec cet objectif en tête.

Maintenabilité :

- Rend la maintenance du programme plus aisée.
- On peut corriger, améliorer chaque module séparément.

Découpage en modules

Pas facile de déterminer le découpage optimal. Souvent plusieurs bonnes solutions.

Deux propriétés principales d'un bon module :

- Forte cohésion : les services offerts au sein d'un même module doivent être liés les uns aux autres.
- Couplage faible : les modules doivent être aussi indépendants que possible les uns des autres.

Exemples

- Bons modules : fonctions de manipulations de matrices, fonctions de traitement de chaînes de caractères (`<string.h>`), définition de structures de nombres complexes.
- Mauvais modules : modules 'help', 'truc', ou 'divers' contenant toutes les fonctions auxiliaires d'un programme.

Grands types de modules

Data pool : une collection de variables et de constantes liées entre elles.

- Exemples : <limits.h> et <float.h>.

Librairie (bibliothèque) : une collection de fonctions liées entre elles

- Exemples : ensemble de fonctions mathématiques <math.h>, fonctions de manipulations de chaînes de caractères <string.h>.

Objet abstrait : une collection de fonctions opérant sur une structure de données cachées

- Exemples : un dictionnaire de la langue français, une pile.

Type abstrait de données : définition d'un nouveau type de données avec les opérations associées.

- Contraitement à un objet abstrait, on peut créer plusieurs objets de même type.
- Exemples : type complexe, type dictionnaire, type pile.

Data pool : implémentation typique

Juste un fichier .h avec des définitions de constantes (macro or `const`)

Exemple : `limits.h` (extrait)

```
/* Copyright (C) 1991-2018 Free Software Foundation, Inc.
   This file is part of the GNU C Library. */
...
/* Minimum and maximum values a `signed short int' can hold.  */
#  define SHRT_MIN        (-32768)
#  define SHRT_MAX        32767

/* Maximum value an `unsigned short int' can hold.  (Minimum is 0.)  */
#  define USHRT_MAX        65535

/* Minimum and maximum values a `signed int' can hold.  */
#  define INT_MIN        (-INT_MAX - 1)
#  define INT_MAX        2147483647
/* Maximum value an `unsigned int' can hold.  (Minimum is 0.)  */
#  define UINT_MAX        4294967295U

/* Minimum and maximum values a `signed long int' can hold.  */
#  if __WORDSIZE == 64
#   define LONG_MAX        9223372036854775807L
#  else
#   define LONG_MAX        2147483647L
#  endif
#  define LONG_MIN        (-LONG_MAX - 1L)
...
```


Librairie : implémentation typique

Un fichier d'entête contenant les prototypes des fonctions accessibles au client et un fichier source les implémentant.

Exemple : une librairie de fonctions mathématiques

`fctmath.h`

```
#ifndef _FCTMATH_H
#define _FCTMATH_H

...
double cos(double);
float cosf(float);

double sin(double);
float sinf(float);

double pow(double, double);
float powf(float, float);

double sqrt(double);
float sqrtf(float);
...

#endif
```

`fctmath.c`

```
#include "fctmath.h"

...
double cos(double x) {
     ...
};

float cosf( float x) {
     ...
};

double sin( double )  {
     ...
};

float sinf( float )  {
     ...
};
...
```


Objet abstrait : implémentation typique

Idem qu'une librairie mais les fonctions sont toutes relatives à une structure de données abstraite qui est définie concrètement dans le fichier source.

Exemple : un dictionnaire français

`dicofr.h`

```
#ifndef _DICOFR_H
#define _DICOFR_H

int dicofr_init();
int dicofr_destroy();
int dicofr_search_word(char *word);
char *dicofr_get_definition(char *word);
int dicofr_add_word(char *word,char *definition);
...

#endif
```

`dicofr.c`

```
#include "dicofr.h"

// définition de la structure interne au module
static struct {
  ...
} dicofr;

// définition des fonctions de l'interface

int dicofr_init() {
  ...
}

int dicofr_destroy() {
  ...
};

int dicofr_search_word(char *word) {
  ...
}
...
```

Type abstrait de données : implémentation typique

Le fichier d'entête définit le nouveau type et fournit le prototype des fonctions opérants sur les objets de ce type. Le fichier source définit concrètement les fonctions.

Exemple 1 : type de données complexe

`complex.h`

```c
#ifndef _COMPLEXE_H
#define _COMPLEXE_H

// définition du nouveau type

typedef struct {
  double re, im;
} complex;

// prototypes des fonctions

complex complex_new(double, double);
complex complex_sum(complex, complex);
complex complex_product(complex, complex);
double  complex_modulus(complex);
...

#endif
```

`complex.c`

```c
#include "complex.h"

// Implémentation des fonctions

complex complex_new(double re, double im) {
  ...
}
complex complex_sum(complex a, complex b) {
  ...
}
complex complex_product(complex a, complex b) {
  ...
}
double complex_modulus(complex c) {
  ...
}
...
```


Type abstrait de données : implémentation typique

Exemple 2 : type de données matrice

`matrix.h`

```c
#ifndef _MATRIX_H
#define _MATRIX_H

// définition du nouveau type

typedef struct {
  double **el;
  unsigned n, m;
} matrix;

// prototypes des fonctions

matrix *matrix_create(unsigned, unsigned);
void matrix_free(matrix *);
matrix *matrix_sum(matrix *, matrix *);
matrix *matrix_product(matrix *, matrix *);
double matrix_determinant(matrix *);
...

#endif
```

`matrix.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include "matrix.h"

// Implémentation des fonctions
matrix *matrix_create(unsigned n, unsigned m) {
  ...
}

void matrix_free(matrix *m) {
  ...
}

matrix *matrix_sum(matrix *a, matrix *b) {
  ...
}
...
```


Remarque 1 : documentation

Il est important de documenter précisément toutes les fonctions de l'interface, dans le fichier d'entête et/ou via un document séparé.

Exemple :

```c
/* -------------------------------------------------------------------- *
 * Creates a m by n matrix with all values set to 0. Returns NULL if
 * m <= 0 or n <= 0 and otherwise a pointer to the new matrix.
 *
 * ARGUMENTS
 * m          The number of rows
 * n          The number of columns
 *
 * RETURN
 * matrix     A pointer to the new matrix or NULL
 *
 * NOTE
 * The returned matrix should be cleaned with matrix_free after usage
 * --------------------------------------------------------------------*/

matrix *matrix_create(unsigned m, unsigned n)
```


Remarque 2 : conventions de nommage

Lorsqu'on utilise plusieurs modules, il est possible que des fonctions avec le même nom soient présentes dans différents modules si on n'y prête pas attention.

Particulièrement probable avec des types abstraits de données (exemple : `init`, `create`, `is_full`...).

Pour éviter les conflits de noms, il est utile de mettre le nom de module ou du type de données en préfixe des noms de fonctions du module.

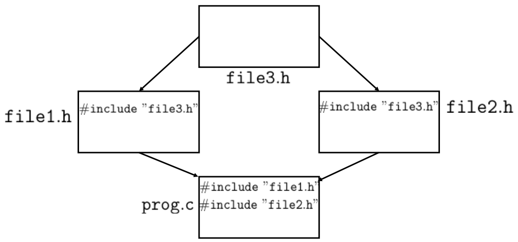
Exemples :

- `complex_new`, `complex_add`, `matrix_create`, `matrix_new`...
- `complexNew`, `complexAdd`, `matrixCreate`, `matrixNew`...

Inclusion gardée

Tout client d'un module doit inclure l'entête de ce module pour pouvoir accéder aux fonctions, variables et types qui y sont définis.

Un fichier d'entête peut inclure un autre fichier d'entête et donc un même fichier d'entête peut être compilé plusieurs fois.



Engendre des erreurs si un fichier d'entête inclus plusieurs fois contient :

- des définitions de types
- des déclarations de variables avec initialisation

Inclusion gardée

Deux solutions :

- S'arranger pour qu'un fichier d'entête ne soit inclus qu'une seul fois (compliqué)
- Utiliser la technique de l'inclusion gardée.

On entoure le fichier d'entête (`module.h`) des instructions suivantes :

```
#ifndef _MODULE_H  // N'inclut ce qui suit que si _MODULE_H n'est pas définie
#define _MODULE_H  // définit la constante _MODULE_H
...
// corps du fichier d'entête
...
#endif
```

Le nom de la constante n'a pas d'importance (`_MODULE_H`, `__MODULE__`...) pour autant qu'il soit le plus éloigné possible de noms communs de constantes (`H`, `N`...).

Pas toujours nécessaire mais autant le faire systématiquement.

Plan

Masquage de l'information

Un bon module garde secret les détails d'implémentation non utiles au client.

Deux avantages :

- Securité : le client ne peut pas corrompre les données. Il est obligé d'utiliser les fonctions de l'interface.
- Flexibilité : le programmeur peut modifier l'implémentation sans devoir en référer au client.

Deux mécanismes sont disponibles en C pour faire ça :

- Déclaration statique des variables globales et fonctions/procédures privées.
- Utilisation d'un type opaque.

Variables globales et fonctions statiques

Le mot-clé `static` permet de rendre les variables globales et les fonctions/procédures inaccessibles en dehors du code source où elles sont définies.

Doit être utilisé pour toutes les fonctions et variables internes au module non destinées à être utilisées par le client.

Exemples :

dicofr.h

```
#ifndef _DICOFR_H
#define _DICOFR_H

int dicofr_init();
...

#endif
```

dicofr.c

```
#include "dicofr.h"

// définition de la structure interne au module
static struct {
  ...
} dicofr;
```

```
// définition des fonctions de l'interface

static int load_data() {
  ...
}

static void terminate(char *message) {
  printf("%s\n", message);
  exit(EXIT_FAILURE);
}

int dicofr_init() {
  ...
  if (!load_data()) {
    terminate("error when loading data");
  }
  ...
}
...
```


Remarque : classe de stockage statique

Le mot-clé `static` a une signification tout à fait différente quand il est utilisé pour des variables locales.

```
int func(int n) {
  static int a;
  ...
}
```

Dans ce contexte, `static` indique que la variable est allouée de manière permanente (statique) et garde sa valeur d'un appel à l'autre de la fonction `func`.

Utilisation très spécifique, à éviter dans le contexte du cours.

Types opaques : motivation

Problème de l'implémentation précédente du type de données 'matrix' : la structure est visible dans l'entête et donc est accessible au client :

matrix.h

```
#ifndef _MATRIX_H
#define _MATRIX_H

// définition du nouveau type

typedef struct {
  double **el;
  unsigned n, m;
} matrix;
...
```

Le client est autorisé à utiliser `mat.m` ou `mat.n` directement s'il le désire et à créer une matrice sans passer par `matrix_new`.

Quid si on veut changer la structure ? Il faudra modifier tous les clients utilisant l'ancienne structure.

La solution est de rendre la structure opaque.

Types opaques : exemple 1

matrix.h

```
#ifndef _MATRIX_H
#define _MATRIX_H

// définition du nouveau type

typedef struct matrix_t matrix;

// prototypes des fonctions

matrix *matrix_create(unsigned, unsigned);
void matrix_free(matrix *);
double matrix_getelement(matrix *,int,int);
void matrix_setelement(matrix *,int,int);
unsigned matrix_getnbrows(matrix *);
unsigned matrix_getnbcols(matrix *);
...

#endif
```

matrix.c

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

// Définition concrète du type
struct matrix_t {
  double **el;
  unsigned n, m;
};

// Implémentation des fonctions
matrix *matrix_create(unsigned n, unsigned m) {
  ...
}

void matrix_free(matrix *m) {
  ...
}
...
```

- '`typedef struct matrix_t matrix`' définit le type matrix comme une structure `matrix_t` mais ne la définit pas.
- La définition concrète est déplacée dans le fichier source.
- Les clients, qui incluent `matrix.h`, n'y ont donc plus accès.

Types opaques

Une fois le type rendu opaque, on peut changer l'implémentation de la structure sans affecter les clients.

Restriction : le client n'ayant pas accès aux détails de la structure, il ne peut y accéder que par pointeur.

Exemple :

```c
#include "matrix.h"
...
matrix m; // Impossible !
matrix *m; // Ok !
...
```

En conséquence, le code du type `complex` du slide 124 ne peut pas être rendu opaque. Il faut plutôt utiliser l'implémentation par pointeur du slide 47.

Types opaques : exemple 2

complex.h

```c
typedef struct complex_t complex;

// constructeur
complex *complex_new(double, double);
// destructeur
void complex_destroy(complex *);
// accesseurs
double  complex_real_part(complex *);
double  complex_imgry_part(complex *);
void complex_set_real_part(complex *,double);
void complex_set_imgry_part(complex *,double);
// opérateurs
void complex_sum(complex *, complex *);
void complex_product(complex *, complex *);
double  complex_modulus(complex *);
...
```

complex.c

```c
#include <math.h>
#include "complex.h"

struct complex_t {
  double re, im;
};

complex *complex_new(double re, double im) {
  complex *c = (complex *)malloc(sizeof(complex));
  c->re = re;
  c->im = im;
  return c;
}

double complex_real_part(complex *a) {
  return a->re;
}

void complex_set_real_part(complex *a, double re) {
  a->re = re;
  return a;
}
...
```


Plan

Généricité

Pour augmenter la réutilisabilité d'un module, on souhaite souvent le rendre le plus générique possible.

Exemple : module de tri (basé sur le tri par insertion)

sort.h

```c
int sort(int array[], int length);
```

sort.c

```c
#include "sort.h"

void sort(int array[], int length) {
  int i = 1;
  while (i < length) {
    int key = array[i];
    int j = i;
    while (j > 0 && array[j-1]>key) {
      array[j] = array[j-1];
      j--;
    }
    array[j] = key;
    i++;
  }
}
```

Inutilisable si on veut trier des tableaux d'autre chose que des entiers.

Généricité en C

Le C n'a pas de mécanisme simple pour rendre le code générique. Il faut un peu bricoler.

Trois techniques néanmoins :

- Utilisation de macros (ou `typedef`)
- Pointeur de fonctions
- Pointeur sur void

Généricité en C via des macros

On peut paramétriser le type du tableau trié en utilisant un `#define` (ou un `typedef`).

`sort.h`

```
#define SORTTYPE int
[Ou bien: typedef int SORTTYPE;]

SORTTYPE sort(SORTTYPE array[], int length);
```

`sort.c`

```
#include "sort.h"

void sort(SORTTYPE array[], int length) {
  int i = 1;
  while (i < length) {
    SORTTYPE key = array[i];
    int j = i;
    while (j > 0 && array[j-1]>key) {
      array[j] = array[j-1];
      j--;
    }
    array[j] = key;
    i++;
  }
}
```

Le client peut définir `SORTTYPE` au type désiré avant d'inclure `"sort.h"`.

On peut utiliser des macros pour faire des choses plus sophistiquées, au détriment de la lisibilité du code.

Pointeurs de fonctions

Soit l'implémentation suivante de la sécante (voir TP1)

```
double secant_method(double approx0, double approx1, double min_error) {
  double xcurrent, xp, xpp;

  xcurrent = approx1;
  xp = approx0;

  while (fabs(xcurrent-xp)) {
    xpp = xp;
    xp = xcurrent;
    xcurrent = ((xpp*f(xp))-((xp)*f(xpp)))/((f(xp)-f(xpp)));
  }

  return xcurrent;
}
```

Le mécanisme de passage de la fonction `f` dont on veut calculer la racine n'est pas satisfaisant.

Solution plus appropriée : passer la fonction `f` en argument. C'est possible en utilisant un pointeur de fonction.

Pointeurs de fonctions

Comme tout élément en C, une fonction dispose d'une adresse en mémoire. Son nom peut être utilisé pour dénoter cette adresse.

Il est possible de stocker ces adresses dans des variables et de les passer en arguments à des fonctions. Les variables et arguments sont alors de type pointeur de fonction.

La déclaration d'une pointeur de fonction se fait comme suit :

```
type (* id) ([type1[, type2] [,...]]]);
```

Note : les parenthèses autour du `*` sont nécessaires pour distinguer le pointeur de fonction d'un fonction renvoyant un pointeur.

Exemples :

- `int (*fonction)(int, int);`
- `void (*procedure)(float);`

Pointeurs de fonctions

Une version générique de la méthode de la sécante :

```
double secant_method(double (* f)(double), double approx0,
                     double approx1, double min_error) {
  ...
   xcurrent = ((xpp*f(xp))-((xp)*f(xpp)))/((f(xp)-f(xpp)));
  ...
}
```

L'appel à `f` peut aussi se faire via `(*f)(.)` ou directement via `f(.)`.

Au niveau du client :

```
double myfunction(double x) {
  return (pow(x,3)-18);
}

int main() {
  double root = secant_method(myfunction, -2.0, 2.0, 0.0005);
}
```


Pointeur sur void

Ce mécanisme n'est pas suffisant pour écrire une fonction de tri générique.

Il faut pour cela qu'on puisse manipuler des données dont le type n'est pas connu à l'avance.

Solution : pointeur sur `void` :

- On passe les données via un pointeur sur void qui peut pointer vers n'importe quoi et est déclaré comme suit :

  $$\texttt{void} \ *\mathrm{p};$$

- Quand on veut manipuler réellement la donnée, on doit néanmoins préciser le type en utilisant une conversion de type. Par exemple :

  $$(\texttt{int} \ *)\mathrm{p}$$

- Idée : on demande au client de fournir les fonctions de manipulation des données en utilisant des pointeurs de fonctions.

Exemple d'algorithme de tri générique

Manipulation de données dans le contexte du tri : comparaison et échange de valeurs.

```
void sort(void *array, int length, int (*compare)(void*, int, int),
          void (*swap)(void *, int, int)) {
  int i = 1;
  while (i < length) {
    int j = i;
    while (j > 0 && !(compare(array, j-1,j))) {
      swap(array,j-1,j);
      j--;
    }
    i++;
  }
}
```

- `swap(array, i, j)` échange les éléments aux positions i et j dans array.
- `compare(array, i, j)` vaut 1 si l'élément i est avant l'élément j en terme d'ordre, 0 sinon.

Tri générique : application 1

Implémentation de `compare` et `swap` pour le tri d'un tableau d'entiers :

```
void swap_int(void *array, int i, int j) {
  int temp = ((int*)array)[i];
  ((int*)array)[i] = ((int*)array)[j];
  ((int*)array)[j] = temp;
}

int compare_int(void *array, int i, int j) {
  return (((int*)array)[i] <= ((int*)array)[j]);
}
```

Utilisation au niveau du client :

```
int A[5]={5, 4, 3, 2, 1};
sort(A, 5, compare_int, swap_int);
```

Tri générique : application 2

Implémentation de compare et swap pour trier un tableau de complexes selon leurs modules :

```
void swap_complex(void *array, int i, int j) {
  complex temp = ((complex*)array)[i];
  ((complex*)array)[i] = ((complex*)array)[j];
  ((complex*)array)[j] = temp;
}

int compare_complex_mod(void *array, int i, int j) {
  return (complex_modulus(((complex*)array)[i]) <= complex_modulus(((complex*)array)[j]));
}
```

Utilisation au niveau du client :

```
complex C[5]={{2,5}, {3,4}, {0,3}, {1,2}, {1,0}};
sort(C, 5, compare_complex_mod, swap_complex);
```


La programmation modulaire en C

Les mécanismes de programmation modulaire en C, tels qu'exposés, sont fonctionnels mais relativement rudimentaires.

Dans d'autres langages de programmation, la modularité est gérée de manière plus naturelle, même si les concepts restent les mêmes.

Par exemple :

- En programmation orientée objet (C++, Java, etc.), il existe de nombreux mécanismes pour générer le masquage de l'information, la définition de nouveau type abstrait et la généricité.
- En programmation fonctionnelle (Scheme, Lisp, Haskell), les fonctions sont des valeurs comme les autres qui peuvent être passées comme arguments sans passer par un mécanisme de pointeurs.

Plan

Compilation d'un seul fichier C

Si tout le code source tient dans un seul fichier (nom_programme.c), on peut le compiler via la commande suivante :

```
gcc -o nom_executable nom_programme.c
```

Si le code est séparé en différents modules, on peut les compiler en une seule ligne :

```
gcc -o nom_executable nom_programme.c module1.c module2.c
```

Cette commande compile en fait séparément les différents fichiers avant de les lier pour créer l'exécutable.

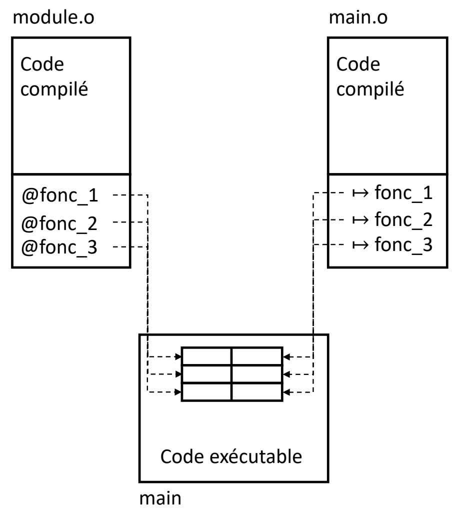
Les différentes étapes de la compilation

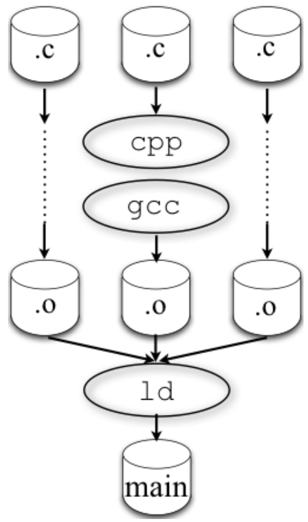
Pour chaque fichier source séparément, on passe par les étapes suivantes :

- Pré-traitement : éliminations des commentaires, remplacement des macros, inclusion des sous-fichiers
- Génération d'un fichier objet (`.o`) contenant :
  - le code compilé en langage d'assemblage
  - La table des liens (les variables/fonctions exportées ou importées par le module)

Ensuite, l'exécutable est obtenu à partir des fichiers objets en utilisant l'éditeur de liens (`ld`). Ce dernier assemble les codes compilés et fait les liens entre les appels de fonctions et de variables.

Les différentes étapes de la compilation





Les différentes étapes de la compilation

Pour faire les étapes manuellement :

```
> gcc -c nom_programme.c
> gcc -c module1.c
> gcc -c module2.c
> gcc -o nom_executable nom_programme.o module1.o module2.o
```

Avec l'approche manuelle, on ne doit pas recompiler la totalité des modules, seulement ceux qui ont été modifiés.

Par contre, il peut être très fastidieux de gérer les recompilations à la main.

Deux solutions :

- Utiliser un IDE (environnement de développement intégré. Par exemple CodeBlocks)
- Utiliser l'outil unix Make en ligne de commande

Make

Make est un utilitaire permettant de gérer la compilation d'un programme réparti en plusieurs fichiers (pas nécessairement en C).

Make est un outil très puissant dont on va juste voir ici le strict minimum.

Principe :

- On place un fichier appelé `Makefile` (ou `makefile`) dans le répertoire où se trouvent les sources.
- On lance la compilation en faisant :

  `> make [cible]`

  où `cible` est défini dans le fichier `Makefile`.

Fichier `Makefile`

Le fichier est constitué d'une suite de règles :

```
cible: dépendances
[tabulation] actions
```

où :

- `cible` est un nom de fichier ou un simple label
- `dépendances` est une liste de fichiers dont dépend la cible
- `actions` est une liste d'actions à effectuer

Suite à un appel à `make cible`, les actions ne sont effectuées que si la date du fichier cible est moins récente qu'au moins l'un des fichiers de dépendances.

Si les fichiers de dépendances apparaissent comme cibles dans une règle, ils sont mis à jour récursivement selon le même principe avant de gérer la cible courante.

Illustration : version 1

Exemple de `Makefile` dans le cas d'un programme `prog.c` basé sur deux modules, `module1.c` et `module2.c`.

```
1 main: prog.o module1.o module2.o
2     gcc -o main prog.o module1.o module2.o
3 prog.o: prog.c module1.h module2.h
4     gcc -c prog.c
5 module1.o: module1.c module1.h
6     gcc -c module1.c
7 module2.o: module2.c module2.h
8     gcc -c module2.c
```

On construit le programme en faisant `make main` ou plus simplement `make` (qui utilise la première cible du fichier par défaut).

Remarque : on peut obtenir toutes les dépendances dans un répertoire (les lignes 3, 5, et 7) en faisant '`gcc -MM *.c`'.

Illustration : version 2

- Make sait comment obtenir un fichier o à partir d'un fichier c. On peut enlever les lignes 4, 6, et 8.
- Il faut néanmoins lui dire quel compilateur utiliser via la définition d'une variable `CC` au début du fichier (ligne 1 ci-dessous).
- On peut fournir les flags de compilation via une variable `CFLAGS`.

```
1 CC = gcc
2 CFLAGS = -Wall -Wextra -Wmissing-prototypes --pedantic\
3          -std=c99
4 main: prog.o module1.o module2.o
5     gcc -o main prog.o module1.o module2.o
6 prog.o: prog.c module1.h module2.h
7 module1.o: module1.c module1.h
8 module2.o: module2.c module2.h
```


Remarque sur les flags de compilation

Pour ce cours (devoirs et projets), on utilise les flags suivants ;

- `--std=c99` : spécifie la norme C99
- `-pedantic` : application stricte de la norme C99
- `-Wall` : affiche (presque) tous les warnings
- `-Wextra` : affiche d'autres warnings
- `-Wmissing-prototypes` : affiche un warning pour les prototypes non définis

Une liste complète des warnings générés par les options `-Wall` et `-Wextra` peut être obtenue ici :

https://gcc.gnu.org/onlinedocs/gcc-6.1.0/gcc/Warning-Options.html

Illustration : version 3

On peut encore simplifier le fichier en utilisant des variables, substituées dans les règles via `$(VAR)`, où `VAR` est le nom de la variable.

```
1  OFILES = prog.o module1.o module2.o
2  TARGET = main
3  CC = gcc
4  CFLAGS = -Wall -Wextra -Wmissing-prototypes --pedantic\
5           -std=c99
6  $(TARGET): $(OFILES)
7      $(CC) $(OFILES) -o $(TARGET)
8  prog.o: prog.c module1.h module2.h
9  module1.o: module1.c module1.h
10 module2.o: module2.c module2.h
```

Les seules lignes à écrire sont les lignes 1 et 2. Les lignes 3-7 sont génériques et les lignes 8-10 sont obtenues directement via '`gcc -MM *.c`'.

Illustration : version finale

- Il peut être utile d'ajouter des règles non liées à des fichiers.
- Par ex. pour supprimer les fichiers compilés (`clean`), exécuter le programme (`run`) et créer une archive avec le code (`archive`)

```
1  OFILES = prog.o module1.o module2.o
2  TARGET = main
3  CC = gcc
4  CFLAGS = -Wall -Wextra -Wmissing-prototypes --pedantic\
5           -std=c99
6  .PHONY: all clean run archive
7
8  all: $(TARGET)
9  clean:
10     rm -f $(OFILES) $(TARGET)
11 run: $(TARGET)
12     ./$(TARGET)
13 archive:
14     tar cvfz illustration_makefile.tar.gz *.h *.c Makefile README
15 $(TARGET): $(OFILES)
16     $(CC) $(OFILES) -o $(TARGET)
17 prog.o: prog.c module1.h module2.h
18 module1.o: module1.c module1.h
19 module2.o: module2.c module2.h
```

- La ligne 6 (`.PHONY: ...`) précise que les cibles qui suivent ne correspondent pas à des fichiers.
- La cible `all` en première position est celle qui est exécutée par défaut.

Synthèse

`make` est un outil très puissant et très complexe. Des livres entiers lui sont consacrés.

C'est l'outil de base pour faciliter la distribution de code source sous environnements de type Unix/Linux.

Beaucoup plus portable que de distribuer des projets produits par un IDE.

Pour plus d'information :
https://www.gnu.org/software/make/manual/make.html.

Plan

Tests

Une fois qu'un programme compile, il faut le tester, c'est-à-dire vérifier que son comportement est conforme au comportement attendu (test de conformité)

Un des avantages de la programmation modulaire est qu'on peut tester chaque module séparément.

Vérifier le fonctionnement de portions de code (p.ex, une fonction, un module) indépendamment des programmes qui les utilisent est ce qu'on appelle un test unitaire.

Il existe aussi des tests d'intégration, qui vérifient les interactions entre modules, et des tests systèmes, qui vérifient le comportement d'un système complet.

Tests unitaires en pratique

Pour chaque module, on crée un fichier source `module_main_test.c` avec une fonction `main` chargée de réaliser les tests.

Ces tests doivent :

- mettre en œuvre l'ensemble des fonctionnalités décrites dans les spécifications du module
- explorer le fonctionnement du module dans des conditions non-spécifiées

Établir ces tests peut constituer un défi en soi.

- Par exemple, comment tester la validité d'un générateur de nombres aléatoires, une fonction mathématique, une générateur d'images ?

Une bonne pratique est d'écrire les tests avant d'implémenter le module.

Exemple 1

module.h

```c
...
int abs(int a);
...
```

module.c

```c
#include "module.h"
...
int abs(int a) {
  if (a < 0) return -a;
  return a;
}
...
```

test_main_module.c

```c
#include "module.c"
#include <assert.h>

int main() {
  ...
  // tests de la fonction abs
  int x = -3;
  int y = abs(x);
  assert (x == y || -x == y);
  assert (y >= 0);
  x = abs(y);
  assert (y == x);
  ...
  return 0;
}
```


La macro assert

```c
void assert(scalar expression)
```

`assert` est une macro définie dans `assert.h`.

Son unique argument est une assertion, c'est-à-dire une expression qu'on suppose être vraie à un moment de l'exécution du programme.

Fonctionnement :

- Si l'expression a une valeur non nulle, `assert` ne fait rien.
- Si l'expression a une valeur nulle, un message est affiché et l'exécution du programme est arrêtée.

Exemple de message :

```
a.out: module.c:9: main: Assertion `y >= 0' failed.
```

Exemple 2

module.h

```
...
int swap(int *x, int *y);
...
```

module.c

```
#include "module.h"
...
int swap(int * x, int * y) {
  if (x == NULL || y == NULL)
     return -1;
  int t = *x;
  *x = *y;
  *y = t;
  return 0;
}
...
```

test_main_module.c

```
#include "module.c"
#include <assert.h>

int main() {
  ...
  // tests de la fonction swap
  int a = 0, b = 1;
  int rv = swap(&a, &b);
  assert (rv == 0);
  assert (a == 1);
  assert (b == 0);
  rv = swap(&a, NULL);
  assert (rv != 0);
  assert (a == 1);
  ...
  return 0;
}
```


Tests unitaires

L'approche informelle manuelle précédente est suffisante dans le cadre de ce cours.

C'est celle que nous utiliserons pour tester vos codes.

Il existe cependant de nombreux outils pour construire ces tests de façon plus systématique. Par exemple : CUnit, cmocka, Seatest...

Débogage

Une fois qu'un test a mis en évidence un problème dans une fonction d'un module, il faut isoler et corriger l'erreur (bogue) dans le programme. C'est ce qu'on appelle le débogage.

Activité assez compliquée en général : le bogue peut être très éloigné du symptôme.

Une programmeur passe en général plus de temps à déboguer du code existant qu'à écrire du nouveau code.

Bogues fréquents en C

- Mauvais cast (ou cast implicite) provoquant une perte de précision
  - `(float)(x/y)` au lieu de `((float) x)/((float y))`
- Accès en dehors des tableaux : en particulier le `'\0'` des chaînes de caractères.
- Variable/mémoire non initialisée
- Confusion entre = et ==
- Ne pas traiter le code de retour d'une fonction pouvant indiquer une erreur
- Boucle infinie

Prévenir les bogues

Pour prévenir les bogues (ou faciliter le débogage) :

- Raisonnez sur papier avant de vous lancer dans l'implémentation (pensez aux invariants)
- Documentez votre code (surtout lorsque vous travaillez à plusieurs)
- Privilégiez toujours la lisibilité et la clarté du code à sa compacité.
  Un mauvais exemple :

```
void my_strcopy(char dest[], char src[]) {
    while ((*dst++ = *src++));
}
```

- Evitez les effets de bord (variables globales, variables locales statiques...)
- Activez (et supprimez) tous les warnings de compilation
- Adoptez une programmation défensive (voir plus loin).

Techniques de débogage

Lecture du code

- En général, inefficace si l'auteur est le lecteur

Exécution instrumentée :

- Ajouts d'assertions
- Ajouts de `printf`

Exécution contrôlée

- Utilisation d'un débogueur, permettant d'exécuter le code pas-à-pas, de mettre des points d'arrêts, de consulter les variables en temps réel...
- Par exemple, `gdb` ou votre IDE préféré (p. ex., CodeBlocks).

Utilisation de `printf` pour le débogage

Approche très simple et facile à mettre en œuvre mais peut être longue et fastidieuse.

Un classique : afficher les indices d'accès à un tableau pour détecter les débordements.

Remarque :

- La commande `printf` est bufferisée : l'appel n'affiche pas tout de suite le résultat à l'écran.
- Le bug pourrait donc suivre un printf avorté plutôt que le précéder.
- Préférez l'instruction `fprintf(stderr,...)` plutôt que `printf(...)`, qui n'est pas bufferisée.

Utilisation de `printf` pour le débogage

Des macros permettent d'afficher de l'information utile lors du débogage :

- `__LINE__` : le numéro de ligne,
- `__FILE__` : le nom de fichier,
- `__FUNCTION__` : le nom de la fonction.

Macro générale d'affichage d'un message de debogage :

```
#define DEBUG(message, indice) \
     fprintf(stderr, "Error (%s%d): ligne %d, fonction \
     [%s], fichier [%s]\n", message, indice, __LINE__,\
       __FUNCTION__,__FILE__)
```

Utilisation :

```
DEBUG("indice i=",i);
```

Programmation défensive

La programmation défensive consiste à penser à tous les cas possibles de mauvaises utilisations de ses fonctions, à ajouter des tests pour détecter ces situations, et à rapporter les erreurs correspondantes.

Elle consiste essentiellement à vérifier la pré-condition des algorithmes mais on peut également tester des invariants.

Elle permet de détecter les symptômes d'un bug au plus tôt.

Deux solutions principales en cas d'erreur :

- Afficher un message et arrêter le programme.
- Prévoir un résultat spécial de la fonction (cfr. le return de la fonction `main()`)

Programmation défensive : un exemple (Bradley, 1998)

```
int reverse(char *in, char *out) {
  if (!in || !out) return -1;

  int i;
  for (i = 0; in[i]; i++);

  assert(in[i] = '\0');

  for (i--; i >= 0; i--) {
    assert(in[i] != '\0');
    *out = in[i];
    out++;
  }
  *out = '\0';
  return 0;
}
```

- La fonction renvoie 0 si tout s'est bien passé, -1 sinon.
- Les `assert` permettent de détecter un problème au niveau de l'algorithme.
- Ce code reste cependant vulnérable à des argument mal formés *(pourquoi ?)*.

Programmation défensive : un exemple (Bradley, 1998)

Une version plus robuste, en ajoutant comme argument la longueur maximale de la chaîne pointée par `in` :

```
int nreverse(char *in, char *out, int maxLength) {
  if (!in || !out) return -1;

  int i;
  for (i = 0; in[i]; i++) {
    if (i > maxLength) return -2:
  }

  assert(in[i] = '\0');

  for (i--; i >= 0; i--) {
    assert(in[i] != '\0');
    *out = in[i];
    out++;
  }
  *out = '\0';
  return 0;
}
```


Programmation défensive : remarques

Faire des tests systématiques des arguments a un coût en terme de temps de calcul.

Généralement seulement utile pour les fonctions de l'interface d'un module auxquels ont accès les utilisateurs. Pas nécessaire pour les fonctions statiques du module.

On réservera l'utilisation de `assert` à des tests du bon fonctionnement de l'algorithme, pas aux tests de vérification des arguments.

La macro `assert` peut être désactivée en faisant.

```
#define NDEBUG
```

Permet de maintenir deux versions du code compilé, une version pour le débogage et une version de production.

Plan

1. Programmation modulaire
2. Masquage de l'information
3. Généricité
4. Compilation
5. Tests et débogage
6. Style

Euh ?

```c
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))

long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
    else switch(m1-=h){
    case f:
        a=(b=(c=(d=g)<<g)<<g)<<g;
        return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
    case h:
        for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
    case g:
        if(n<h)return(g);
        if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
        else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
        if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
        return(o[b-g]%n+k-h);
    default:
        if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
        for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
    }
```


Style

- Le style de programmation est un ensemble de lignes directrices utilisées lors de l'écriture d'un programme informatique.
- Inclut principalement des questions liées à l'aspect visuel du code mais pas uniquement.
- Suivre un (bon) style de programmation permet de rendre le code source plus lisible par soi-même et par d'autres (y compris les correcteurs) et permet d'éviter les erreurs.
- Important dans la mesure où un programme est souvent développé par plusieurs auteurs et où une grande partie de la vie d'un programme est consacrée à sa maintenance.
- La qualité d'un style est assez difficile à apprécier et subjective.
- Pour ce cours, pas de règles strictes, plutôt une série de conseils.

Identifiants

- Utilisés pour la dénomination de variables, fonctions, types, et constantes.
- Suites de lettres, chiffres et de '_'. Doivent commencer par une lettre ou '_'. La casse (majuscule/minuscule) compte.
- Dénominations en anglais (de préférence) ou en français, pour autant que vous soyez cohérent.
- Différents styles :
  - noms composés séparés par des '_' : `add_interest`, `number_of_days`... ("old school")
  - noms composés séparés par des majuscule : `addInterest`, `numberOfDays`... ("camel case").

Identifiants

En général :

- Ne pas utiliser d'abbréviations :
  - `firstName`, `lastName` au lieu de `fname`, `lname`
- Pas de noms trop longs
  - `setField` au lieu de `setTheLengthField`
- Eviter les dénominations trop proches
  - `thisPerishableProduct` au lieu de `perishableProduct` si `perishableProducts` existe.
- Utiliser des noms informatifs pour que le code soit auto-documenté.

```
double tax1;          // sales tax rate
double tax2;          // income tax rate

double salesTaxRate;
double incomeTaxRate;
```


Identifiants

Convention courante (mais non obligatoire) en fonction de la nature de l'identifiant :

- Variables et fonctions commencent par une minuscule
  - `myVar`, `my_var`, `myFunction`, `my_function`
- Types commencent par une majuscule
  - `MyType`, `My_type`
- Constantes en majuscule et utilisation de '_'
  - `MY_CONST`

Identifiants : fonctions

- Utiliser des verbes d'action.
  - `addInterest`, `convertToAustralianDollars`
- Préfixe `get` et `set` pour obtenir et donner une valeur à une variable.
  - `getBalance`, `setBalance` pour des opérations sur la variable `balance`.
- Préfixe `is` et `has` pour des fonctions retournant un booléen.
  - `isOverdrawn`, `hasCreditLeft`

Formatage du code

- Concerne l'indentation, les alignements, l'utilisation des espaces...
- Affecte uniquement la lisibilité du code.
- Une fois un style choisi, le conserver.
- Il existe des outils de formatage automatique
  - Par exemple, `indent` sous unix/linux.

Formatage : indentation et blocs

- Indentation :
  - Ajouter un nombre fixe d'espaces (2 ou 4 par exemple) à chaque rentrée dans un nouveau bloc
  - Éviter l'utilisation de tabulations (qui dépendent du système)
  - Souvent gérée automatiquement par votre éditeur de texte
- Sous-blocs :
  - Découper le code en sous-blocs en laissant une ligne vide entre ces blocs (composés de quelques lignes)
  - Laisser deux lignes vides entre chaque fonction

Formatage : espaces

`for(int i=0;i<n;i++)` vs. `for (int i = 0; i < n; i++)`

Mettre des espaces :

- Avant et après les opérateurs binaires, arithmétiques, et logiques
  `b = 1 + 2;`
  sauf éventuellement pour mettre en évidence la précédence :
  `a*x + b`
- Après les virgules et les points-virgules
  `myfunction(3, 4, 5)`
- Après les mots-clés réservés (`for`, `if`, `else`, `do`...)
- Après le signe de début de commentaires inline :
  `// this is a comment`
- Pour aligner du code si ça améliore la lisibilité

```
int n      = atoi(argv[1]);      //  size of population
int trials = atoi(argv[2]);      //  number of trials
```


Formatage : espaces

Ne pas en mettre :

- Entre les opérateurs unaires et leur variable
  `*p++ = !a;`
- Autour des parenthèses
  `myfunction(5 * (4+5))`
- Autour des opérateurs de sélection
  `member.data`, `node->next`, `vec[i]`...
- Avant la ponctuation
  `myfunction(3, 4, 5)`

Formatage : accolades

- Deux écoles (parmi d'autres) :

```
int add(int a, int b) {
   int result;
   if (a) {
      result = a + b;
      return result;
   } else {
      return result;
   }
}
```

```
int add(int a, int b)
{
   int result;
   if (a)
   {
      result = a + b;
      return result;
   }
   else
   {
      return result;
   }
}
```

- Les deux sont valides. La première est utilisée dans ces slides pour gagner de la place.

Formatage : les accolades

Les accolades ne sont pas obligatoires lorsque le bloc ne contient qu'une instruction :

```c
if (currentHour < AFTERNOON) {
  printf("Morning\n");
} else if (currentHour < EVENING) {
  printf("Afternoon\n");
} else {
  printf("Evening\n");
}
```

```c
if (currentHour < AFTERNOON)
  printf("Morning\n");
else if (currentHour < EVENING)
  printf("Afternoon\n");
else
  printf("Evening\n");
```

Attention cependant aux cas ambigüs :

```c
if (b1)
  if (b2)
     printf("here");
else
  printf("there");
```

⇔

```c
if (b1) {
  if (b2)
     printf("here");
  else
     printf("there");
}
```


Documentation et commentaires

Différents types de commentaires :

- Commentaires d'entête : précise l'auteur du code, la date de création/modification, description succincte du contenu du module, copyright, etc.
- Commentaires de documentation : définit le contrat de chaque fonction/procédure de l'interface du module.
- Commentaires de bloc : résume l'action d'une partie de code
- Commentaires *inline* : précise l'intérêt d'une ligne particulière de code

Documentation

```c
/* -------------------------------------------------------------------- *
 * Creates a m by n matrix with all values set to 0. Returns NULL if
 * m <= 0 or n <= 0 and otherwise a pointer to the new matrix.
 *
 * ARGUMENTS
 * m           The number of rows
 * n           The number of columns
 *
 * RETURN
 * matrix      A pointer to the new matrix or NULL
 *
 * NOTE
 * The returned matrix should be cleaned with matrix_free after usage
 * --------------------------------------------------------------------*/

matrix *matrix_create(unsigned m, unsigned n)
```

Définition non ambigüe des fonctions de l'interface à l'adresse du client.

Un autre programmeur doit être capable de réimplémenter la fonction uniquement sur base des commentaires.

Il existe des outils permettant de générer automatiquement la documentation d'un code source à partir des commentaires et du code.

- Par exemple, doxygen.

Commentaires de bloc et inline

Les commentaires de bloc ou inline doivent décrire principalement le pourquoi d'une portion de code, le comment étant expliqué par le code lui-même.

Ni trop, ni trop peu :

- les réserver aux parties non triviales.
  Un mauvais exemple :

  ```c
  i++          // increment i by one
  ```

- Privilégier l'auto-documentation en utilisant des noms de fonctions et de variables informatifs
- S'il y a besoin d'en mettre trop, c'est probablement que le code manque de clarté. Il vaut mieux alors le réécrire.

Remarque : commentaires imbriqués

En C, on ne peut pas imbriquer des commentaires délimités par `/*...*/`.

Le code suivant génèrera une erreur (pourquoi ?) :

```
/* /* coucou */ */
```

Par contre, cette construction est possible :

```
/*
// coucou
*/
```

Que vaut la variable nest ?

```
int nest = /*/*/0*/**/1;
```


Quelques bonnes pratiques en vrac

- Déclarer des constantes au lieu d'utiliser des nombres/chiffres dans le code :
  Non pas :

```
day = (3 + numberOfDays) % 7;
```

  Mais plutôt :

```
const int WEDNESDAY = 3;
const int DAYS_IN_WEEK = 7;
day = (WEDNESDAY + numberOfDays) % DAYS_IN_WEEK;
```

- Créer des fonctions courtes
  - Si elle ne s'affiche pas sur un écran, il est probablement possible de la découper en plusieurs autres méthodes
- Se limiter à maximum trois niveaux de boucles imbriquées
- Pas plus de 80 caractères par ligne.
- Si nécessaire, aller à la ligne après une virgule ou avant un opérateur.

Quelques bonnes pratiques en vrac

- Éviter les écritures ambiguës même si elles sont autorisées par le langage

```
int b1 = 1;
int b2 = 2;
if (b1 == b2)
   printf("%d\n", b1);
```

```
int b1 = 1;
int b2 = 2;
if (b1 = b2)
   printf("%d\n", b1);
```

- Ne pas déclarer les variables sur la même ligne

```
int a, b, c;
```

$\Rightarrow$

```
int a;
int b;
int c;
```

- Déclarer les variables juste avant leur utilisation et initialiser les en même temps que leur déclaration.

Quelques bonnes pratiques en vrac

- Ne pas réutiliser une même variable dans des situations différentes
  - Si elle est judicieusement nommée, cela ne devrait pas vous effleurer l'esprit
- Éviter les répétitions de code. Mieux vaut créer une fonction que copier dix fois la même suite d'instructions.
- S'intéresser au langage
  - Utiliser les idiomes appropriés
    - `for` au lieu de `while` si gardien pas trop compliqué, `switch` au lieu de `if` imbriqués, `++`, `+=`...
  - Fouiller pour vérifier l'existence d'une fonction plutôt que de réinventer la roue

Sources

Programmation modulaire :

- C programming : a modern approach, K.N. King, W. W. Norton & Company, Second edition, 2008.
- Slides du cours INFO0030, Benoît Donnet

Style :

- A guide to coding style, Justus Piater, 2005.
- Notes du cours Algorithmique I, Renaud Dumont, 2009-2010.

Partie 4

Complexité

23 décembre 2019

Plan

1. Introduction
2. Approche empirique
3. Approche mathématique

Plan

1. Introduction
2. Approche empirique
3. Approche mathématique

Algorithmes et structures de données

- Vous avez maintenant toutes les bases de programmation en C pour pouvoir résoudre n'importe quel problème.
- Pour aborder un nouveau problème, vous ne devez pas partir de rien : Depuis plus de 50 ans, les informaticiens ont étudié et proposé des algorithmes et structures de données standards qui peuvent servir de briques de base pour aborder de nombreux problèmes.
- Dans la suite du cours, on verra les bases de ce domaine qui est un domaine scientifique à part entière [2]
- La résolution de problèmes informatiques requière systématiquement de combiner algorithmes et structures de données (cf. projet 1).
- Le critère de base pour l'analyse et l'évaluation de ces algorithmes et structures de données est leur performance (ou coût) en termes de temps de calcul et en termes d'espace mémoire consommé.

2. Ces notions seront largement approfondies dans le cours INFO0902 (et d'autres).

Pourquoi se soucier des performances ?

L'intérêt pratique d'un programme est très souvent dicté par ses performances :

- Est-il possible d'obtenir les résultats voulus en un temps raisonnable ?
- Peut-on traiter des volumes de données suffisamment importants ?

Étudier la performance d'un programme permet :

- de prédire son comportement
  - Est-ce que mon programme va se terminer ? Après combien de temps ?
- de comparer différents algorithmes et implémentations
  - Puis-je rendre mon programme plus rapide ? Si oui, comment ?

Améliorer les performances

L'amélioration des performances est la dernière étape du cycle de développement d'un programme.

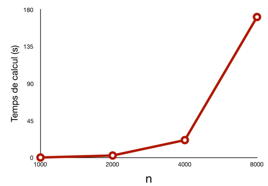
1. Ecriture du programme
2. Compilation ⇒ si erreur (de syntaxe), retour en 1
3. Exécution et vérification du bon fonctionnement (tests unitaires) ⇒ si erreur (de sémantique), retour en 1
4. Test du programme en situation réelle et sur de vraies données ⇒ si erreur (de performance), retour en 1

Comme les autres étapes, on évite les itérations en y réfléchissant d'abord sur papier.

Analyse de performance

Pour analyser les performances d'un programme, on adopte une approche scientifique classique basée soit sur :

- l'expérimentation : on mesure les temps de calcul dans des conditions réelles
- la modélisation mathématique : on dérive une formule mathématique liant les performances aux données d'entrée.



$$T(n) = O(n^3)$$

Contrairement à d'autres sciences (chimie, biologie, physique, sociologie...), en sciences informatiques :

- les expérimentations sont quasi gratuites
- la modélisation mathématique est nettement plus aisée

Illustration : problème 3SUM

Le problème 3SUM :

> *Étant donné n nombres entiers, énumérer tous les triplets de valeurs sommant à 0.*

Trouve des applications théoriques et pratiques (en géométrie et crytographie).

Solution naïve (pour le comptage) : on énumère tous les triplets de valeurs et on teste leur somme.

```c
int count_3sum(int tab[], int n) {
  int count = 0;
  for (int i = 0; i < n-2; i++)
    for (int j = i+1; j < n-1; j++)
      for (int k = j+1; k < n; k++)
        if (tab[i]+tab[j]+tab[k] == 0)
          count++;
  return count;
}
```

Combien de temps prendra ce programme pour un tableau d'un million

Plan

Approche empirique

Principe : On implémente l'algorithme, on l'exécute et on mesure ses performances.

Il faut trouver des données représentatives sur lesquels tester l'algorithme. Deux options :

- On collecte des données réelles
- On écrit un programme pour générer des données

Exemple : un générateur de données pour le problème 3SUM

```c
void generate_3sum_data(int tab[], int n, int m) {
  for (int i = 0; i < n; i++)
     tab[i] = rand() % (2*m) - m;
}
```

Génère un tableau aléatoire de n nombres entiers pris dans $\{-m, \ldots, m-1\}$. La probabilité de trouver des triplets sommant à zéro dépend de n et m.

Comment mesurer les temps de calcul ?

Pour mesurer le temps d'exécution d'un programme, on peut utiliser les fonctions de `time.h`

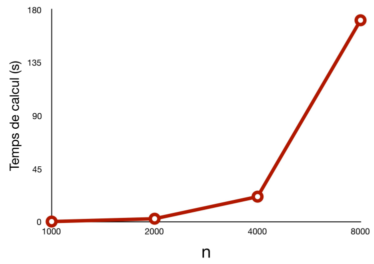
```c
#include <time.h>

....
clock_t begin = clock();

// The code you want to monitor should be here

clock_t end = clock();

double time_spent = (double)(end-begin) / (double)CLOCKS_PER_SEC;
```

La variable `time_spent` contient le temps (en secondes) pris par le code entre les appels à `clock()`.

Illustration sur 3SUM

On double la taille du tableau d'un essai à l'autre avec des entiers compris entre -1000000 et +999999.

n	temps (s)
1000	0.34
2000	2.75
4000	21.45
8000	170.9



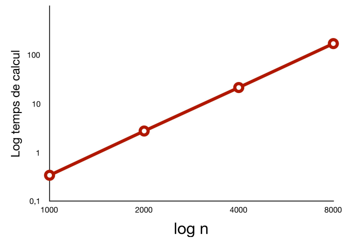
(Sur un Intel Core i7 2.5 GHz)

Analyse des données

Tracer la courbe sur une échelle logarithmique :

- Si les points sont sur une droite, la courbe est de la forme $a.n^b$ (c'est souvent le cas).
- L'exposant b est donné par la pente de la courbe.
- Le facteur a peut s'obtenir à partir des données

n	$T(n)$	$\log_2 n$	$\log_2 T(n)$	$T(n+1)/T(n)$
1000	0,34	10	-1,5	-
2000	2,75	11	1,5	8
4000	21,45	12	4,4	7,8
8000	170,9	13	7,4	8



$$\begin{aligned} T(n) &\approx an^b \Rightarrow b \approx 3 \text{ et } a \approx \frac{1}{3}10^{-9} \\ &\approx \frac{1}{3}10^{-9}n^3 \end{aligned}$$

Prediction et vérification

Hypothèse : le temps de calcul de `count_3sum` est $\approx \frac{1}{3}10^{-9}n^3$.

Vérification :

- Pour $n = 16000$, la fonction devrait prendre 1365 secondes.
- Temps observé : 1375 secondes (23 minutes).

Prédiction : Pour $n = 1$ million, la fonction prendra 333 millions de secondes ($>$ 10 ans).

Temps de calcul moyens

Les temps de calcul peuvent fluctuer fortement en fonction des données.

Exemple :

- vérification de l'occurrence d'un triplet sommant à zéro plutôt que comptage :

```
int has_3sum(int tab[], int n) {
  for (int i = 0; i < n-2; i++)
    for (int j = i+1; j < n-1; j++)
      for (int k = j+1; k < n; k++)
        if (tab[i] + tab[j] + tab[k] == 0)
          return 1;
  return 0;
}
```

- Pour $n = m = 1000000$, les temps de calcul peuvent aller de 0 (le premier triplet testé somme à 0) à plus de 10 ans (aucun triplet ne somme à zéro).

Dans ce cas, il est utile de répéter l'expérience plusieurs fois avec des données aléatoires et de calculer la moyenne (et l'écart-type) des temps de calcul.

Avantages et limitations de l'approche empirique

Avantages :

- Expériences très faciles à réaliser.
- On mesure les temps de calcul de l'implémentation réelle.

Limitations :

- Demande d'implémenter l'algorithme (pour peut-être se rendre compte qu'il est inefficace).
- Temps de calcul dépendent de l'implémentation et de la machine, même à solution algorithmique fixée.
- Ne fournit pas une preuve formelle de l'évolution des temps de calcul avec n. Tirer des conclusions à partir de valeurs expérimentales peut mener à des erreurs.

Plan

Approche mathématique

Principe :

- On fait des hypothèses sur le modèle d'exécution du programme
  - Exécution séquentielle (pas de parallélisme)
  - Les instructions élémentaires prennent une temps constant
  - ...
- On compte le nombre de fois que chaque instruction est exécutée.
- On obtient les temps de calcul en sommant le temps d'exécution (constant) de chaque instruction multiplié par son nombre d'exécutions.

Le temps dépend en général des données d'entrée :

- On exprime le temps de calcul en fonction de la taille des données d'entrée.
- Si les nombres d'exécutions des instructions dépendent de l'entrée, on se place dans le cas le plus défavorable (*worst case*).

Illustration : 2SUM

```
1 int count_2sum(int tab[], int n) {
2   int count = 0;
3   for (int i = 0; i < n-1; i++)
4     for (int j = i+1; j < n; j++)
5       if (tab[i] + tab[j] == 0)
6         count++;
7   return count;
8 }
```

Ligne	Coût	Nombre d'exécutions
2	c_1	1
3	c_2	n
4	c_2	$\sum_{i=0}^{n-2}(n-i)$
5	c_3	$\sum_{i=0}^{n-2}(n-i-1)$
6	c_4	n_4
7	c_5	1

Sachant que

- $\sum_{i=0}^{n-2}(n-i) = \frac{n(n+1)}{2} - 1$
- $\sum_{i=0}^{n-2}(n-i-1) = \frac{(n-1)n}{2}$ (le nombre de paires à tester)
- n_4 dépend du tableau mais vaut au pire cas $\frac{(n-1)n}{2}$ (toutes les paires somment à 0)

on a :

$$T(n) = \frac{c_2 + c_3 + c_4}{2} n^2 + \frac{3c_2 - c_3 - c_4}{2} n + (c_1 - c_2 + c_5)$$

Illustration : 2SUM

$$T(n) = an^2 + bn + c$$

Les constantes a, b, c et d dépendent :

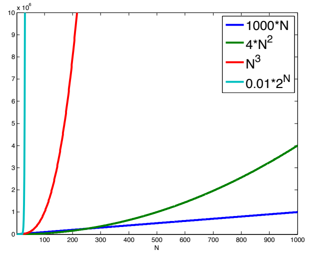
- de l'implémentation exacte (via les nombres d'exécutions)
- de la machine (via les constantes c_i)

Idéalement, on aimerait caractériser les performances d'un algorithme indépendamment de l'implémentation et de la machine.

Solution : on se focalise sur la vitesse de croissance asymptotique des temps de calcul.

Analyse asymptotique

- On s'intéresse à la vitesse de croissance ("order of growth") de $T(n)$ lorsque n est très grand ($n \to \infty$).
  - Tous les algorithmes sont rapides pour des petites valeurs de n
- On simplifie généralement $T(n)$:
  - en ne gardant que le terme dominant
    - Exemple : $T(n) = 10n^3 + n^2 + 40n + 800$
    - T(1000)=100001040800, $10 \cdot 1000^3 = 10000000000$0
  - en ignorant le coefficient du terme dominant
    - Asymptotiquement, ça n'affecte pas l'ordre relatif



- Exemple : 2SUM : $T(n) = an^2 + bn + c \to n^2$.

Pourquoi est-ce important ?

- Supposons qu'on puisse traiter une opération de base en $1\mu s$.
- Temps d'exécution pour différentes valeurs de n

T(n)	$n = 10$	$n = 100$	$n = 1000$	$n = 10000$
n	$10\mu s$	$0.1ms$	$1ms$	$10ms$
$400n$	$4ms$	$40ms$	$0.4s$	$4s$
$2n^2$	$200\mu s$	$20ms$	$2s$	$3.3m$
n^4	$10ms$	$100s$	~ 11.5 jours	317 années
2^n	$1ms$	4×10^{16} années	3.4×10^{287} années	...

(Dupont)

Pourquoi est-ce important ?

- Taille maximale du problème qu'on peut traiter en un temps donné :

T(n)	en 1 seconde	en 1 minute	en 1 heure
n	1×10^6	6×10^7	3.6×10^9
$400n$	2500	150000	9×10^6
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

- Si m est la taille maximale que l'on peut traiter en un temps donné, que devient cette valeur si on reçoit une machine 256 fois plus puissante ?

T(n)	Temps
n	$256m$
$400n$	$256m$
$2n^2$	$16m$
n^4	$4m$
2^n	$m + 8$

(Dupont)

Notation asymptotique “grand-O”

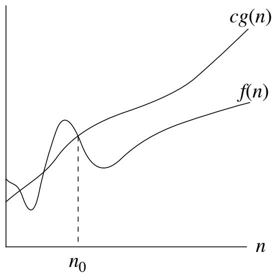
Les vitesses de croissance sont généralement précisées en utilisant la notation asymptotique “grand-O” (ou encore notation de Landau).

Définition : Soient f et g deux fonctions $\mathbb{N} \to \mathbb{R}^+$. On dira que

$$f \in O(g)$$

ssi

$$\exists n_0 \in \mathbb{N}, c \in \mathbb{R}^+ : \forall n > n_0 : f(n) \leq cg(n)$$



Par abus de notation, on écrira aussi : $f(n) \in O(g(n))$ ou $f(n) = O(g(n))$.

Complexité en temps

On dira qu'un algorithme a une complexité en temps $O(f(n))$ (ou plus simplement est $O(f(n))$) si ses temps de calcul dans le pire cas $g(n) \in O(f(n))$.

- Exemple : La complexité de `count_2sum` est $O(n^2)$.

Remarque importante : La notation grand-O sert à exprimer une borne supérieure sur la complexité.

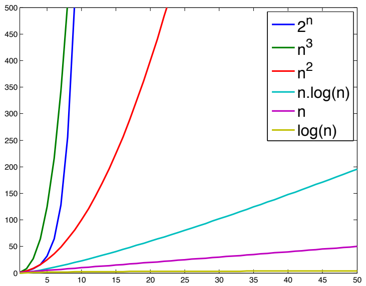
- Généralement, quand on dit qu'un algorithme est $O(f(n))$, on suppose que $O(f(n))$ est le plus petit sous-ensemble contenant la fonction $g(n)$ exprimant les temps de calcul de l'algorithme dans le pire cas.
- Par ex. : on ne dira pas que `count_2sum` est $O(n^3)$ même si $O(n^2) \subset O(n^3)$.

Exemples :

- $n^2 + 2n + 2 \Rightarrow O(n^2)$
- $n^2 + 100000n + 3^{1000} \Rightarrow O(n^2)$
- $\log(n) + n + 4 \Rightarrow O(n)$
- $10^{-4} n \log(n) + 3000n \Rightarrow O(n \log(n))$
- $2n^{30} + 3^n \Rightarrow O(3^n)$

Hiérarchie de classes de complexité

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^{a>1}) \subset O(2^n)$$



Classes de complexité : dénominations et exemples

Complexité constante	$O(1)$	Instructions élémentaires
Complexité logarithmique	$O(\log n)$	Recherche dichotomique (cf. partie 5)
Complexité linéaire	$O(n)$	Parcours d'un tableau 1D (par ex., recherche du maximum)
Complexité linéarithmique	$O(n \log n)$	Tri par fusion (cf. partie 5)
Complexité quadratique	$O(n^2)$	Parcours d'un tableau 2D, 2SUM
Complexité cubique	$O(n^3)$	Multiplication matricielle naïve
Complexité exponentielle	$O(2^n)$	Tour de Hanoï (partie 2), énumération des sous-ensembles d'un ensemble
Complexité factorielle	$O(n!)$	Approche naïve du voyageur de commerce

Analyse de complexité en pratique

En pratique, il n'est (la plupart du temps) pas nécessaire de compter explicitement le nombre d'exécutions de chaque instruction.

On peut calculer la complexité en notation grand-O directement en se basant sur les propriétés suivantes :

- Si $f(n) \in O(g(n))$, alors pour tout $k \in \mathbb{N}$, on a $k \cdot f(n) \in O(g(n))$
  - Exemple : $\log_a(n) \in O(\log_b(n))$, $a^{n+b} \in O(a^n)$
- Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors $f_1(n) + f_2(n) \in O(g_1(n) + g_2(n))$ et $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$
  - Exemple : $\sum_{i=1}^{m} a_i n^i \in O(n^m)$
- Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

Analyse de complexité en pratique

Quelques règles simples :

- Affectation, accès à un tableau, opérations arithmétiques, appel de fonction : $O(1)$
- Instruction If-Then-Else : O(complexité max des deux branches)
- Séquence d'opérations : l'opération la plus couteuse domine (règle de la somme)
- Boucle simple : $O(nf(n))$ si le corps de la boucle est $O(f(n))$

Analyse de complexité en pratique

- Double boucle complète : $O(n^2 f(n))$ où $f(n)$ est la complexité du corps de la boucle
- Boucles incrémentales : $O(n^2)$ (si corps $O(1)$)

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
      ...
```

- Boucles avec un incrément exponentiel : $O(\log n)$ (si corps $O(1)$)

```
for (int i = 1; i <= n; i = 2*i)
   ...
```


Exemple : 3SUM

```
1 int count_3sum(int tab[], int n) {
2   int count = 0;
3   for (int i = 0; i < n-2; i++)
4     for (int j = i+1; j < n-1; j++)
5       for (int k = j+1; k < n; k++)
6         if (tab[i]+tab[j]+tab[k] == 0)
7           count++;
8   return count;
9 }
```

- Ligne 3 exécutée $O(n)$ fois.
- Ligne 4 exécutée $O(n^2)$ fois.
- Lignes 5, 6, et 7 exécutées une fois par triplet de valeurs. Nombre de triplets :

$$C_n^3 = \frac{n(n-1)(n-2)}{6} \in O(n^3).$$

$\Rightarrow$ Complexité en temps : $O(n^3)$.

Limitations de l'analyse asymptotique

- Les facteurs constants ont de l'importance pour des problèmes de petites tailles
  - Si n est petit, il vaut mieux un algorithme s'exécutant en $O(n^2)$ secondes qu'un algorithme s'exécutant en $O(\log n)$ années.
- Deux algorithmes de même complexité (grand-O) peuvent avoir des propriétés très différentes
  - Comme `count_3sum`, les deux algorithmes suivants sont $O(n^3)$
  - `count_3sum_2` est 6 fois plus lent et `has_3sum` peut traiter des tableaux aléatoires de taille 1 million en moins d'un centième de seconde.

```c
int count_3sum_2(int tab[], int n) {
  int count = 0;
  for (int i = 0; i < n-2; i++)
    for (int j = 0; j < n-1; j++)
      for (int k = 0; k < n; k++)
        if (i < j && j < k)
          if (tab[i]+tab[j]+tab[k] == 0)
            count++;
  return count;
}
```

```c
int has_3sum(int tab[], int n) {
  for (int i = 0; i < n-2; i++)
    for (int j = i+1; j < n-1; j++)
      for (int k = j+1; k < n; k++)
        if (tab[i]+tab[j]+tab[k] == 0)
          return 1;
  return 0;
}
```

$\Rightarrow$ Important de tester l'algorithme dans des conditions réelles (ou de réaliser une analyse mathématique plus fine).

Complexité d'algorithmes récursifs

L'analyse de la complexité d'algorithmes récursifs mène généralement à une équation récurrente, dont la résolution n'est pas toujours aisée.

Exemple :

- fonction factorielle :

```c
int fact(int n) {
  if (n <= 1)
     return n;
  return n * fact(n-1);
}
```

$$\begin{aligned} T(0) &= c_0 \\ T(n) &= T(n-1) + c_1 \end{aligned}$$

- Solution : $T(n) = c_1 n + c_0 \in O(n)$

On se contentera de voir quelques cas particuliers dans ce cours.

Complexité en espace

La complexité en espace d'un algorithme mesure l'espace mémoire utilisé par l'algorithme en fonction de la taille de l'entrée.

Comme la complexité en temps :

- On la calcule dans le pire cas.
- On l'exprime en utilisant la notation grand-O.

Dans le cas des algorithmes récursifs, il faut prendre en compte l'espace mémoire nécessaire au stockage du contexte des appels récursifs, qui est proportionnel à la profondeur de l'arbre des appels récursifs.

Par exemple : la complexité en espace de `fact` est $O(n)$.

Exercice

Quelles sont les complexités en temps et en espace de la fonction `pow_rec2` ?

```c
float pow_rec2(float a, int x) {
    if (x == 1)
       return a;
    if (x % 2 == 0) // x pair
       return pow_rec2(a * a, x/2);
    else            // x impair
       return a * pow_rec2(a * a, (x-1)/2);
}
```

Partie 5

Tri et recherche

23 décembre 2019

Plan

Introduction

Les algorithmes de recherche et de tri sont des algorithmes importants en informatique.

Ils sont directement utiles mais aussi à la base de nombreux autres algorithmes.

Objectifs de cette leçon :

- Vous présenter des solutions efficaces à ces deux problèmes : recherche dichotomique et tri par fusion
- Vous convaincre de l'importance de développer des solutions efficaces
- Vous montrer que le tri et la recherche permet de résoudre efficacement d'autres problèmes algorithmiques.
- Vous présenter la technique du "diviser-pour-régner".

Illustration : filtrage d'adresses emails

On gère un serveur d'emails et on aimerait ajouter une fonctionalité de filtrage des adresses, soit :

- Liste noire : on ne veut pas laisser passer les emails des personnes de la liste
- Liste blanche : on ne veut laisser passer que les emails des personnes de la liste.

Combien d'emails peut-on espérer filtrer à la seconde en fonction de la longueur de la liste ?

Recherche linéaire : tableau quelconque

Warm-up : recherche dans un tableau d'entiers :

- Soit un tableau d'entiers, on veut déterminer si une valeur `key` se trouve dans le tableau.

Solution naïve sans faire d'hypothèse sur les valeurs du tableau :

```
int linear_search(int key, int tab[], int n) {
  for (int i = 0; i < n; i++) {
    if (tab[i] == key)
       return i;
  }
  return -1;
}
```

Complexité : $O(n)$ pour un tableau de taille n.

- Pire cas : l'entier recherché n'est pas dans la liste.

Recherche linéaire : tableau trié

Si on suppose que le tableau est trié, on peut s'arrêter plus tôt dans la recherche.

```
int sorted_linear_search(int key, int tab[], int n) {
  int i = 0;
  while (i < n && key > tab[i])
     i++;

  if (tab[i] == key)
     return i;
  else
     return -1;
}
```

Complexité identique : $O(n)$ pour un tableau de taille n.

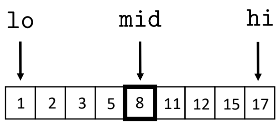
- Pire cas : l'entier recherché est plus grand que toutes les valeurs dans le tableau

Recherche dichotomique (*binary search*)

On peut faire (beaucoup) mieux si on suppose que le tableau est trié.

Idée : On compare la valeur recherchée à la valeur au milieu du tableau :

- Si elle est égale, on la renvoie
- Si elle est plus petite, on recherche récursivement la valeur dans la première moitié du tableau
- Si elle est plus grande, on recherche récursivement la valeur dans la seconde moitié du tableau



Recherche dichotomique : implémentation récursive

```
int binary_search_aux(int key, int tab[], int lo, int hi) {

  if (lo > hi) return -1;

  int mid = lo + (hi - lo) / 2;

  if (key == tab[mid])
    return mid;
  else if (key < tab[mid])
    return binary_search_aux(key, tab, lo, mid-1);
  else
    return binary_search_aux(key, tab, mid+1, hi);
}

int binary_search(int key, int tab[], int n) {
  return binary_search_aux(key, tab, 0, n-1);
}
```

Remarque : `lo+(hi-lo)/2` est préférable à `(lo+hi)/2` pour éviter un dépassement de valeur si `lo` est un entier très grand.

Analyse de complexité

Le nombre d'appels récursifs est maximum lorsque la valeur ne se trouve pas dans le tableau.

Supposons pour simplifier les calculs que la taille du tableau n soit telle que $n = 2^k - 1$ pour un entier $k > 0$ ($\Rightarrow k = \log_2(n+1)$).

Les tailles des sous-tableaux considérés à chaque étape sont :

$$2^k - 1, 2^{k-1} - 1, 2^{k-2} - 1, \ldots, 2^1 - 1, 2^0 - 1$$

Il faudra donc $k+1$ appels récursifs avant d'arriver au cas de base (`lo>hi`).

En dehors de l'appel récursif, le corps de la fonction est $O(1)$.

La complexité en temps est donc $O(\log n)$.

La complexité en espace est aussi $O(\log n)$ (au plus k appels récursifs imbriqués).

Recherche dichotomique : implémentation itérative

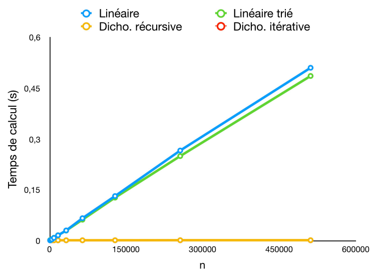
```
int binary_search_iter(int key, int tab[], int n) {
  int lo = 0;
  int hi = n-1;

  while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;
    if (key < tab[mid])
      hi = mid - 1;
    else if (key > tab[mid])
      lo = mid + 1;
    else
      return mid;
  }

  return -1;
}
```

Complexité en temps : $O(\log n)$
Complexité en espace : $O(1)$

Analyse des temps de calcul

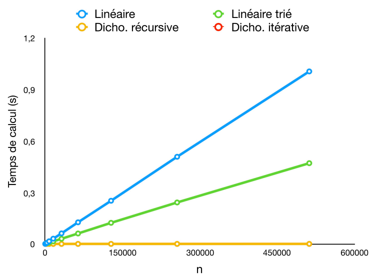
Recherche positive :

- Tableau $[0, 1, \ldots, n-1]$
- 1000 recherches d'une clé : `rand()%n`



Recherche négative :

- Tableau $[0, 2, 4, \ldots, 2n-2]$
- 1000 recherches d'une clé : `rand()%(2*n)+1`



Filtrage d'adresses email : implémentation

Adaptation du code à la recherche d'une chaîne de caractères.

```
int binary_search_iter(char *key, char **tab, int n) {
  int lo = 0;
  int hi = n-1;

  while (lo <= hi) {
    int mid = lo + (hi - lo) / 2;
    int cmp = strcmp(key, tab[mid]);
    if (cmp < 0)
      hi = mid - 1;
    else if (cmp > 0)
      lo = mid + 1;
    else
      return mid;
  }

  return -1;
}
```

Remarques :

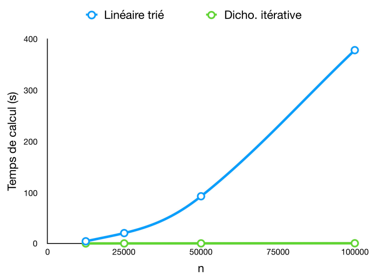
- `strcmp(s1,s2)` renvoie 0 si les deux chaînes sont identiques ou un entier < 0 (resp. > 0) si `s1` est avant (resp. après) `s2` dans l'ordre lexicographique (`string.h`).
- On peut aussi écrire une fonction générique en se basant sur des pointeurs sur void et de fonctions (cf. partie 3).

Filtrage d'adresses email : temps de calcul

Génération de données :

- Liste blanche : n chaînes de caractères (`a-z`) aléatoires de longueur 10.
- Requêtes : $10n$ chaînes prises au hasard dans la liste (que des recherches positives)

n	Rech. lin.(s)	Rech. dicho.(s)
12500	4,30	0,03
25000	20,31	0,08
50000	92,41	0,19
100000	378,42	0,39



Pour une table de $n = 100000$ adresses email :

- Recherche dichotomique : 256000 vérifications par seconde.
- Recherche linéaire : 264 vérifications par seconde.

Recherche en $O(1)$

Supposons que le tableau ne contienne que des valeurs entières positives codées sur 8 bits ($0 \leq$ `key` < 256).

On peut représenter le tableau par une vecteur de taille 256 dont la $i$ème valeur vaut 1 si i appartient au tableau, 0 sinon.

Fonction de recherche sous ces hypothèses :

```
int constant_search(int key, unsigned char tab[]) {
  return tab[key];
}
```

Complexité en temps : $O(1)$

Limitations évidentes :

- Ne marche que lorsque les valeurs du tableau sont des entiers bornés.
- Augmente l'espace mémoire nécessaire si l'ensemble de valeurs est petit.

Plan

1. Recherche
2. Tri
3. Application aux problèmes 2SUM et 3SUM
4. Diviser pour régner

Tri

Un des problèmes algorithmiques les plus fondamentaux.

Applications innombrables : tri des mails selon leur ancienneté, tri des résultats de requêtes sur Google, tri des facettes des objets pour l'affichage 3D, gestion des opérations bancaires...

Sert de brique de base pour de nombreux autres algorithmes

- Recherche dichotomique
- Recherche des éléments dupliqués dans une liste
- Recherche du $k$ème élément le plus grand dans une liste
- 3SUM
- ...

Environ 25% du temps de calcul des ordinateurs est utilisé pour trier.

Deux algorithmes quadratiques : tri par sélection

Idée : on ramène itérativement le minimum du reste du tableau à la position courante.

```
void selectionsort(int tab[], int n) {
  for (int i = 0; i < n-1; i++) {
    int imin = i;
    int j;
    for (j = i+1; j < n; j++) {
      if (tab[j] < tab[imin])
        imin = j;
    }
    if (imin != j) {
      int tmp = tab[i];
      tab[i] = tab[imin];
      tab[imin] = tmp;
    }
  }
}
```

Complexité : $O(n^2)$

- Double boucle complète quel que soit le contenu du tableau

Deux algorithmes quadratiques : tri par insertion

Idée : on insére la valeur à la position i à sa bonne position dans le sous-tableau qui la précéde supposé préalablement trié.

```
void insertionsort(int tab[], int n) {
  int i = 1;
  while (i < n) {
    int key = tab[i];
    int j = i;
    while (j > 0 && tab[j-1]>key) {
      tab[j] = tab[j-1];
      j--;
    }
    tab[j] = key;
    i++;
  }
}
```

Complexité : $O(n^2)$

- Pire cas : la valeur key doit être ramenée au début du tableau à chaque itération de la boucle externe $\Rightarrow$ tableau trié par ordre décroissant.
- Plus efficace que le tri par sélection sur des tableaux presque triés.

Un tri plus efficace

Les tris par sélection ou par insertion sont trop lents pour des applications à grande échelle (voir tests plus loin).

On peut faire (beaucoup) mieux en se basant sur une approche récursive.

Idée du tri par fusion :

- Diviser le tableau en deux.
- Trier les deux sous-tableaux récursivement.
- Fusionner les deux sous-tableaux triés.

Inventé par John von Neumann en 1945, un mathématicien ayant conçu l'architecture des premiers ordinateurs modernes.

Tri par fusion : fonction principale

```
void mergesort(int tab[], int n) {
  mergesort_aux(0, tab, 0, n-1);
}

static void mergesort_aux(int tab[], int lo, int hi) {
  int n = hi - lo + 1;
  if (n <= 1)
    return;
  int mid = lo + (n + 1) / 2;
  mergesort_aux(tab, lo, mid - 1);
  mergesort_aux(tab, mid, hi);
  merge(tab, lo, mid, hi); // fusionne les sous-tableaux triés
                           // tab[lo..mid-1] et tab[mid..hi]
}
```

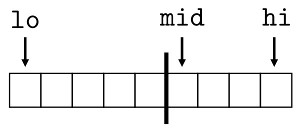
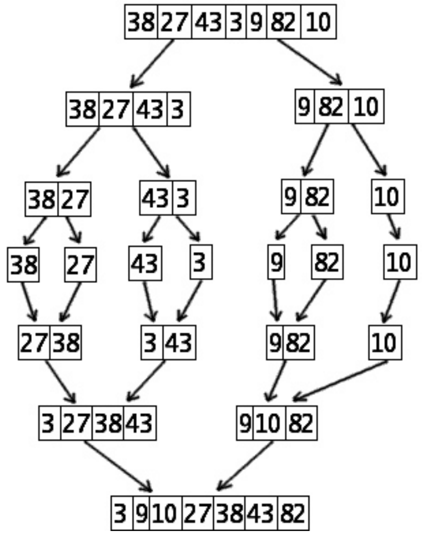


Illustration : trace des appels récursifs

```
mergesort_aux(lo=0, hi=6)
  |mergesort_aux(lo=0, hi=3)
  |  |mergesort_aux(lo=0, hi=1)
  |  |  |mergesort_aux(lo=0, hi=0)
  |  |  |mergesort_aux(lo=1, hi=1)
  |  |  |merge(lo=0, mid=1, hi=1)
  |  |mergesort_aux(lo=2, hi=3)
  |  |  |mergesort_aux(lo=2, hi=2)
  |  |  |mergesort_aux(lo=3, hi=3)
  |  |  |merge(lo=2, mid=3, hi=3)
  |  |merge(lo=0, mid=2, hi=3)
  |mergesort_aux(lo=4, hi=6)
  |  |mergesort_aux(lo=4, hi=5)
  |  |  |mergesort_aux(lo=4, hi=4)
  |  |  |mergesort_aux(lo=5, hi=5)
  |  |  |merge(lo=4, mid=5, hi=5)
  |  |mergesort_aux(lo=6, hi=6)
  |  |merge(lo=4, mid=6, hi=6)
  |merge(lo=0, mid=4, hi=6)
```

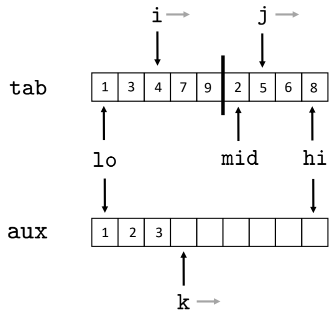


Source : wikipedia

Fusion

Idée (en utilisant un tableau auxiliaire) :

- Un indice pointe vers le début de chacun des deux sous-tableaux.
- On recopie la valeur la plus petite pointée dans le tableau auxiliaire, on incrémente son indice et on recommence jusqu'à ce que toutes les valeurs soient copiées.
- Le tableau auxiliaire est recopié dans le tableau de départ.



Fusion : implémentation

```
static void merge(int tab[], int lo, int mid, int hi, int aux[]) {

  int i = lo, j = mid;

  for (int k = lo; k <= hi; k++)
    if (i == mid)
      aux[k] = tab[j++];
    else if (j == hi + 1)
      aux[k] = tab[i++];
    else if (tab[i] < tab[j])
      aux[k] = tab[i++];
    else
      aux[k] = tab[j++];

  for (int k = lo; k <= hi; k++)
    tab[k] = aux[k];

}
```

Complexité : $O(n)$ avec $n = \mathtt{hi} - \mathtt{lo} + 1$ la taille totale des deux sous-tableaux.

Tri par fusion : code complet

```
void mergesort(int tab[], int n) {
  int aux[n];
  mergesort_aux(0, tab, 0, n-1, aux);
}

static void mergesort_aux(int tab[], int lo, int hi, int aux[]) {
  int n = hi - lo + 1;
  if (n <= 1)
    return;
  int mid = lo + (n + 1) / 2;
  mergesort_aux(tab, lo, mid - 1, aux);
  mergesort_aux(tab, mid, hi, aux);
  merge(tab, lo, mid, hi, aux);
}

static void merge(int tab[], int lo, int mid, int hi, int aux[]) {
  int i = lo, j = mid;
  for (int k = lo; k <= hi; k++)
    if (i == mid)
      aux[k] = tab[j++];
    else if (j == hi + 1)
      aux[k] = tab[i++];
    else if (tab[i] < tab[j])
      aux[k] = tab[i++];
    else
      aux[k] = tab[j++];

  for (int k = lo; k <= hi; k++)
    tab[k] = aux[k];
}
```

Analyse de complexité

Supposons pour simplifier les calculs que la taille du tableau n soit telle que $n = 2^k$ pour un entier $k > 0$ ($\Rightarrow k = \log_2 n$).

On a les appels suivants de la fonction `merge` :

- 1 appel sur un tableau de taille n $\Rightarrow O(n)$
- 2 appels sur des sous-tableaux de tailles $n/2$ $\Rightarrow O(n)$
- 4 appels sur des sous-tableaux de tailles $n/4$ $\Rightarrow O(n)$
- ... ...
- $n/2$ appels sur des sous-tableaux de taille 2 $\Rightarrow O(n)$

On a donc au total $k = \log_2 n$ opérations de complexité $O(n)$.

La complexité en temps est $O(n \log n)$.

La complexité en espace est $O(n + \log n) = O(n)$.

- $O(n)$ pour le tableau auxiliaire, $O(\log n)$ pour les appels récursifs.

Remarques

- Les temps de calcul ne dépendent pas du contenu du tableau, contrairement au tri par insertion.
- Il est possible d'implémenter l'algorithme itérativement et/ou sans utiliser de tableau auxiliaire mais c'est plus compliqué.
- On peut montrer qu'il n'est pas possible d'écrire un algorithme de tri meilleur que $O(n \log n)$, sans faire d'hypothèse supplémentaire sur la nature des valeurs à trier (cf INFO0902).

Filtrage d'adresses email

Adaptation de la fonction merge pour le tri de chaînes de caractères :

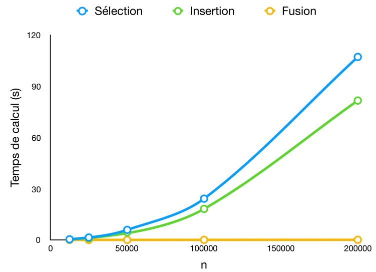
```
static void merge_str(char **tab, int lo, int mid, int hi, char **aux) {
  int i = lo, j = mid;

  for (int k = lo; k <= hi; k++)
    if (i == mid)
      aux[k] = tab[j++];
    else if (j == hi + 1)
      aux[k] = tab[i++];
    else if (strcmp(tab[i], tab[j]) < 0)
      aux[k] = tab[i++];
    else
      aux[k] = tab[j++];

  for (int k = lo; k <= hi; k++)
    tab[k] = aux[k];

}
```

Les autres fonctions peuvent être adaptées trivialement (en changeant `int []` en `char **`).

Analyse empirique

Génération de données : n chaînes de caractères aléatoires de longueur 10.

n	Sélection (s)	Insertion (s)	Fusion (s)
12500	0,36	0,22	<0,01
25000	1,42	0,99	<0,01
50000	5,88	4,48	0,01
100000	24,18	18,18	0,03
200000	107,26	81,69	0,06



Pour 1 millions d'adresses emails :

- Tri par sélection : 44 minutes
- Tri par insertion : 33 minutes
- Tri par fusion : 0,3 secondes

Passer de $O(n^2)$ à $O(n \log n)$ fait une énorme différence.

1. Recherche

2. Tri

3. Application aux problèmes 2SUM et 3SUM

4. Diviser pour régner

Problème 2SUM

Étant donné un tableau de n entiers uniques, trouver (ou compter) les paires d'entiers qui somment à 0.

Solution naïve :

```
int count_2sum(int tab[], int n) {
  int count = 0;
  for (int i = 0; i < n-1; i++)
    for (int j = i+1; j < n; j++)
        if (tab[i] + tab[j] == 0)
          count++;
  return count;
}
```

Complexité en temps : $O(n^2)$

Problème 2SUM : 1ère solution basée sur le tri

Une première solution basée sur le tri et la recherche dichotomique :

- On trie le tableau (en utilisant le tri par fusion).
- Pour chaque élément `tab[i]` (négatif), on recherche `-tab[i]` dans le reste du tableau en utilisant la recherche dichotomique.

```
int count_2sum_bs(int tab[], int n) {
  mergesort(tab, n);
  int count = 0;
  int i = 0;
  while (i < n-1 && tab[i] < 0) {
    if (binary_search(-tab[i],tab+i+1,n-i) != -1)
      count++;
    i++;
  }
  return count;
}
```


Problème 2SUM : 1ère solution basée sur le tri

```
int count_2sum_bs(int tab[], int n) {
  mergesort(tab, n);
  int count = 0;
  int i = 0;
  while (i < n-1 && tab[i] < 0) {
    if (binary_search(-tab[i],tab+i+1,n-i) != -1)
      count++;
    i++;
  }
  return count;
}
```

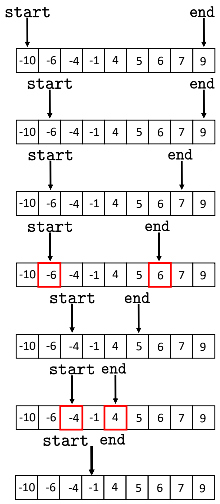
Complexité en temps : $O(n \log n)$

- Tri par fusion : $O(n \log n)$
- Pire cas pour la boucle `for` : toutes les valeurs sont négatives.
  - n recherches dichotomiques en $O(\log n) \Rightarrow O(n \log n)$.

Problème 2SUM : 2ème solution basée sur le tri

On peut se passer de la recherche dichotomique.

```c
int count_2sum_fast(int tab[], int n) {
  mergesort(tab, n);
  int count = 0;
  int start = 0;
  int end = n-1;
  while (start < end) {
    int sum = tab[start] + tab[end];
    if (sum == 0) {
      count++;
      start++;
      end--;
    } else if (sum > 0)
      end--;
    else
      start++;
  }
  return count;
}
```



Problème 2SUM : 2ème solution basée sur le tri

```c
int count_2sum_fast(int tab[], int n) {
  mergesort(tab, n);
  int count = 0;
  int start = 0;
  int end = n-1;
  while (start < end) {
    int sum = tab[start] + tab[end];
    if (sum == 0) {
      count++;
      start++;
      end--;
    } else if (sum > 0)
      end--;
    else
      start++;
  }
  return count;
}
```

Complexité en temps identique à la 1ère solution : $O(n \log n)$

- Tri par fusion : $O(n \log n)$
- Boucle `while` : $O(n)$, négligeable par rapport au tri

Problème 3SUM : 1ère solution basée sur le tri

En se basant sur la recherche dichotomique :

```c
int count_3sum_bs(int tab[], int n) {
  mymergesort(tab,n);
  int count = 0;
  int i = 0;
  while (i < n-2 && tab[i] < 0) {
    int j = i+1;
    while (j < n-1 && tab[i]+tab[j] < 0) {
      if (binary_search(-(tab[i]+tab[j]), tab+j+1, n-j) != -1)
        count++;
      j++;
    }
    i++;
  }
  return count;
}
```

Complexité : $O(n^2 \log n)$

- Double boucle : $O(n^2)$ itérations et recherche dichotomique en $O(\log n) \Rightarrow O(n^2 \log n)$
- Tri par fusion, $O(n \log n)$, négligeable

Problème 3SUM : 2ème solution basée sur le tri

```c
int count_3sum_fast(int tab[], int n) {
  mergesort(tab,n);
  int count = 0;
  int i = 0;
  while (i < n-1 && tab[i] < 0) {
    int start = i + 1;
    int end = n - 1;
    while (start < end) {
      int sum = tab[i] + tab[start] + tab[end];
      if (sum == 0) {
        count++; start++; end--;
      } else if (sum > 0)
        end--;
      else
        start++;
    }
    i++;
  }
  return count;
}
```

Complexité en temps[3] : $O(n^2)$

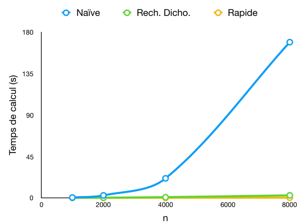
- Double boucle : $O(n^2)$ itérations
- Tri par fusion, $O(n \log n)$, négligeable

3. Longtemps considérée comme la solution optimale mais une solution $O(n^2/(\log n/\log\log n)^{2/3})$ a été proposée en 2014.

Analyse empirique

Tableaux d'entiers compris entre -1000000 et +999999 sans doublons.

n	Naïve	Rech. bin.	Rapide
1000	0,36	0,03	$< 0,01$
2000	2,68	0,14	0,01
4000	21,20	0,60	0,04
8000	169,29	2,75	0,18



(Sur un Intel Core i7 2.5 Ghz)

Prédiction pour $n = 1000000$:

- $>$ 10 ans pour la version naïve $O(n^3)$ (cf. partie 4)
- 17 heures pour la version utilisant la recherche dichotomique $O(n^2 \log n)$
- 1 heure pour la version rapide $O(n^2)$

Synthèse

- Le tri est un composant essentiel dans beaucoup d'applications.
- Le tri par fusion offre une solution optimale au problème.
- Passer de $O(n^2)$ à $O(n \log n)$ ou de $O(n)$ à $O(\log n)$ fait une énorme différence dans les applications pratiques.
- La recherche dichotomique et le tri par fusion sont basés sur l'idée générale du "diviser-pour-régner", qui permet d'obtenir des solutions efficaces à beaucoup de problèmes.

Plan

1. Recherche
2. Tri
3. Application aux problèmes 2SUM et 3SUM
4. Diviser pour régner

Diviser pour régner

Technique générique de résolution de problèmes basée sur la récursivité.

Principe général : Soit un problème algorithmique à résoudre de taille n

- Si le problème est trivial (typiquement, n est petit), on le résout directement
- Sinon :
  - On divise le problème en b sous-problèmes de tailles n/a (avec $b \geq 0$ et $a > 1$ deux entiers).
  - On résout récursivement ces sous-problèmes
  - On fusionne les solutions aux sous-problèmes pour produire une solution au problème original

Exemples : recherche dichotomique, tri par fusion, calcul de puissance récursif.

Diviser pour régner

La complexité d'une solution diviser pour régner s'écrit :

$$T(n) = bT(n/a) + f(n)$$

où $f(n)$ est la complexité de l'opération de fusion.

D'autant plus efficaces que $f(n)$ est faible, a est grand, et b est petit.

Sans intérêt si $f(n)$ n'est pas strictement meilleur que la complexité d'une solution naïve au problème.

Exemples :

- Recherche dichotomique : $a = 2$, $b = 1$, $f(n) = c \Rightarrow T(n) \in O(\log n)$
- Tri par fusion : $a = 2$, $b = 1$, $f(n) = cn \Rightarrow T(n) \in O(n \log n)$
- Calcul d'une puissance entière :
  $a = 2$, $b = 1$, $f(n) = c \Rightarrow T(n) \in O(\log n)$

Un autre exemple : recherche de pics

- Soit un tableau $tab[0\,..\,n+1]$ de $n+2$ valeurs réelles. On supposera que $tab[0] = tab[n+1] = -\infty$.
- Définition : $tab[i]$, avec $i \in \{1, \ldots, n\}$, est un pic s'il n'est pas plus petit que ses voisins :

  $$tab[i-1] \leq tab[i] \geq tab[i+1]$$

  ($tab[i]$ est un maximum local)
- Problème algorithmique : trouver un pic dans le tableau (n'importe lequel)
- Note : il en existe toujours un

Solution naïve

On teste toutes les positions séquentiellement :

```
int findPeak(float *tab, int n) {
  for (int i = 1; i <= n; i++) {
    if ((tab[i-1] <= tab[i]) && (tab[i]<=tab[i+1]))
        return i;
  }
  return -1; // En principe, inutile
}
```

Complexité : $O(n)$

Une solution par diviser-pour-régner

Idée :

- Sonder un élément $tab[i]$ et ses voisins $tab[i-1]$ et $tab[i+1]$
- Si c'est un pic : renvoyer i
- Sinon :
  - les valeurs doivent croître au moins d'un côté

    $$tab[i-1] > tab[i] \text{ ou } tab[i] < tab[i+1]$$

  - Si $A[i-1] > A[i]$, on cherche le pic (récursivement) dans $tab[1\,..\,i-1]$
  - Si $A[i+1] > A[i]$, on cherche le pic (récursivement) dans $tab[i+1\,..\,n]$

(Implémentation en C laissée comme exercice.)

Analyse

Est-ce que l'algorithme est correct ? Oui. Il existera toujours un pic du côté choisi (peut se montrer par l'absurde).

Complexité : $a = 2,\ b = 1,\ f(n) = 1 \Rightarrow O(\log n)$ (comme la recherche dichotomique)

Partie 6

Structures de données

23 décembre 2019

Plan

Plan

Types de données abstraits

Un type de données abstrait définit :

- Un ensemble de données
- Un ensemble d'opérations sur ces données

Types d'opérations standards :

- Création, destruction d'un objet du type donné
- Accès aux données
- Modification des données
  - Insertion et suppression de nouvelle données si l'ensemble est dynamique

Exemples jusqu'ici : nombres complexes, matrices, grilles (cf. projet 1).

Types de données abstraits : implémentation

On implémente concrètement un type de données abstrait en utilisant une structure de données.

Une structure de données consiste en :

- une représentation des données
- une représentation des relations entre ces données

Exemples : tableaux 1D, 2D, liste liée, arbres, graphes...

Pour un même TDA, plusieurs implémentations (structures de données) sont généralement possibles.

On analyse les performances d'une structure particulière selon deux critères :

- Complexité en temps des opérations
- Complexité en espace nécessaire pour la structure

Exprimées dans le pire cas en fonction de la quantité de données présentes dans la structure.

Types de données abstraits : en C (rappel)

Fichier d'entête (`.h`) contient les prototypes des opérations et la définition du type (opaque). Le fichier source (`.c`) contient la définition concrète de la structure et l'implémentation des opérations

```c
// complex.h

#ifndef _COMPLEXE_H
#define _COMPLEXE_H

// définition du nouveau type

typedef struct complex_t complex;

// prototypes des fonctions

complex *complex_new(double, double);
void complex_destroy(complex *);
void complex_sum(complex *, complex *);
void complex_product(complex *, complex *);
...

#endif
```

```c
// complex.c
#include <math.h>
#include "complex.h"

struct complex_t {
  double re, im;
};

complex *complex_new(double re, double im) {
  ...
}
void complex_sum(complex *a, complex *b) {
  ...
}
void complex_product(complex *a, complex *b) {
  ...
}
...
```

On utilise des pointeurs sur void (et éventuellement des pointeurs de fonction) si on veut pouvoir stocker des valeurs arbitraires dans la structure.

TDA standard

Depuis plus de 50 ans, plusieurs TDA standards, utiles dans de nombreuses applications, ont été définis et étudiés dans la littérature.

Principalement des ensemble de données dynamiques.

Quelques exemples :

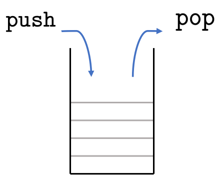
- Pile et file : collection d'objets accessibles selon un politique LIFO/FIFO.
- File à priorité : collection d'objets accessibles selon un ordre de priorité.
- Liste : séquence d'objets accessibles à partir de leur position relative.
- Arbre : collection de valeurs associées au nœuds d'un arbre
- Dictionnaire : collection d'objets accessibles de manière arbitraire via une clé.
- Graphe : collection de valeurs associées aux nœuds d'un graphe et accessible selon ce graphe.

Plan

Plan

Pile

Une pile est une collection de valeurs, accessibles selon une discipline LIFO (Last In First Out)

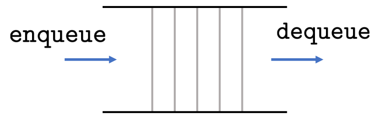


Interface :

- `push(s,v)` : ajoute la valeur `v` au sommet de la pile `s`
- `pop(s)` : retire la valeur au sommet de la pile `s`, et retourne cette valeur. Signale une erreur si la pile est vide.
- `top(s)` : retourne la valeur présente au sommet de la pile `s`, sans la dépiler. Signale une erreur si la pile est vide.
- `size(s)` : retourne le nombre de valeurs présentes dans la pile `s`.
- `isEmpty(s)` : détermine si la pile `s` est vide.
- \+ création de la structure et libération de la mémoire

File

Une file est une collection de valeurs, accessibles selon une discipline FIFO (First In First Out)



Interface :

- `enqueue(q,v)` : ajoute la valeur `v` à la fin de la file `q`
- `dequeue(q)` : retire la valeur au début de la file `q` et la retourne. Signale une erreur si la file est vide.
- `front(q)` : retourne la valeur présente au début de la file `q`, sans la retirer. Signale une erreur si la file est vide.
- `size(q)` : retourne le nombre de valeurs présentes dans la file `q`.
- `isEmpty(q)` : détermine si la file `q` est vide.
- \+ création de la structure et libération de la mémoire

Exemple d'interface en C

stack.h

```c
#ifndef _STACK_H
#define _STACK_H

typedef struct Stack_t Stack;

Stack *stackCreate();
void   stackFree(Stack *);
int    stackPush(Stack *, void *);
void  *stackPop(Stack *);
void  *stackTop(Stack *);
int    stackSize(Stack *);
int    stackIsEmpty(Stack *);

#endif
```

queue.h

```c
#ifndef _QUEUE_H
#define _QUEUE_H

typedef struct Queue_t Queue;

Queue *queueCreate();
void   queueFree(Queue *);
void   queueEnqueue(Queue *, void *);
void  *queueDequeue(Queue *);
void  *queueHead(Queue *);
int    queueSize(Queue *);
int    queueIsEmpty(Queue *);

#endif
```

Les données sont stockées dans la pile/file sous la forme de pointeurs sur `void`

- type de retour des fonctions `stackPop`, `stackTop`, `queueDequeue` et `queueHead` et type du deuxième argument des fonctions `stackPush` et `queueEnqueue`).

Applications

Ces deux structures fondamentales ont de nombreuses applications.

Piles

- Allocation de resources selon un politique "dernier arrivé premier servi"
- Mécanisme d'appels de fonctions dans les langages de programmation
- Implémentation des compilateurs
- Opération undo/back dans certains applications.
- Parcours en profondeur d'abord d'un graphe

Files

- Gestion de resources selon une politique "premier arrivé, premier servi"
- Transfert de données asynchrone (gestion des opérations printf)
- Gestion de requêtes sur un serveur
- Simulation de files d'attente dans la vie réelle
- Parcours en largeur d'abord d'un graphe.

Une application de la file

On aimerait trier les adresses emails d'une liste en vue d'implémenter le filtre de notre serveur d'emails.

Les adresses se trouvent dans un fichier `addresses.txt` (une adresse par ligne) et on aimerait les placer dans un tableau pour le tri.

Solution sur base d'une file :

- Lire les adresses une par une en les stockant dans une file
- Créer un tableau de la même taille que la file
- Retirer les adresses une par une de la file en les stockant dans le tableau

Peut-on utiliser une pile ?

Une application de la file : en C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "queue.h"

const int BUFFER_SIZE = 1000;
int main() {

  char buffer[BUFFER_SIZE];
  // lecture du fichier (sur l'entrée) et stockage dans une file
  Queue *q = queueCreate();
  while (fgets(buffer, BUFFER_SIZE, stdin)) {
    int lenstr = strlen(buffer)-1;
    buffer[lenstr]='\0';   // supprime la fin de ligne

    queueEnqueue(q, strdup(buffer));
  }
  // Creation et remplissage du tableau
  int sizeQ = queueSize(q);
  char **array = malloc(sizeQ * sizeof(char *));
  for (int i = 0; i<sizeQ; i++)
    array[i] = (char *) queueDequeue(q);

  queueFree(q);
  ...
}
```

Une application de la file : en C

Remarques :

- `char *fgets(char *str, int size, FILE *fp)` : lit une ligne dans le fichier `fp` (qui peut être l'entrée standard) et place le résultat dans la chaîne `str`. La chaîne est terminée par le caractère de fin de ligne ('`\n`') et le caractère NUL ('`\0`'). Si la ligne fait plus de `size` caractères, seulement les `size-1` premiers sont lus pour éviter une dépassement de `str`. Renvoie `str` si tout s'est bien passé.
- `size_t strlen(char *str)` : renvoie la longueur de la chaîne (caractère NUL non compris).
- `char *strdup(const chart *src)` : copie la chaîne `src` dans une nouvelle chaîne (en allouant la mémoire nécessaire) et retourne un pointeur vers cette chaîne.

Une application de la pile

La manière standard d'écrire une expression mathématique est la notation infixe, basée sur les parenthèses pour la précédence :

- Exemple : $((3 + 4) * 5) - 8$ $(= 27)$

Notation postfixe (ou polonaise inversée) : les opérateurs suivent les opérandes.

- Exemple : $3\ 4\ +\ 5\ *\ 8\ -$

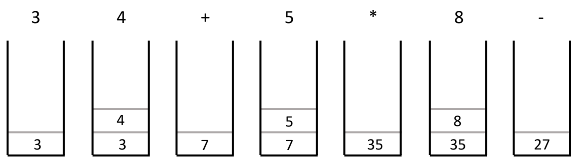
Aucune parenthèse n'est nécessaire dans cette notation : il n'y a qu'une seule manière de mettre des parenthèses.

Les expressions en notation postfixe sont faciles à évaluer en se basant sur une pile.

Evaluation sur base d'une pile

Principe de l'évaluation :

- Si on lit un nombre, on le met (`push`) sur la pile.
- Si on lit un opérateur (binaire) : on retire le second puis le premier opérandes sur la pile (`pop`), on leur applique l'opérateur et on met le résultat sur la pile.



Evaluation d'expression en notation postfixe

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "stack-double.h"

const int BUFFER_SIZE = 1000;

int main() {

  char buffer[BUFFER_SIZE];
  Stack *s = stackCreate();

  while (fgets(buffer, BUFFER_SIZE, stdin)) {
    int lenstr = strlen(buffer)-1;
    buffer[lenstr]='\0';

    if (strcmp(buffer, "+") == 0)
      stackPush(s, stackPop(s)+stackPop(s));
    else if (strcmp(buffer, "-") == 0)
      stackPush(s, -stackPop(s)+stackPop(s));
    else if (strcmp(buffer, "/") == 0)
      stackPush(s, stackPop(s)/stackPop(s));
    else if (strcmp(buffer, "*") == 0)
      stackPush(s, stackPop(s)*stackPop(s));
    else {
      stackPush(s, strtod(buffer, NULL));
    }
  }

  printf("Result = %f\n",stackPop(s));
}
```

```
> gcc -o rpn rpn-evaluation.c stack-double.c
> ./rpn
3
4
+
5
*
8
-
Result = 27.000000
```

Plan

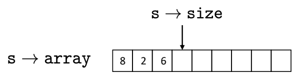
Implémentation par tableau d'une pile : principe

Principes :

- Les valeurs contenues dans la pile sont placées dans les composantes successives d'un tableau
- Un indice `size` contient la taille de la pile, qui est aussi la première position libre dans le tableau

La taille du tableau doit être fixée a priori et une erreur signalée lorsque la taille de la pile excède la taille du tableau.

```c
const int  MAX_STACK_SIZE = 1000;

struct Stack_t {
  void *array[MAX_STACK_SIZE];
  int   size;
};
```



Implémentation par tableau d'une pile : code C

Création et suppression.

```c
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"

const int MAX_STACK_SIZE = 1000;

struct Stack_t {
  void *array[MAX_STACK_SIZE];
  int   size;
};

static void terminate(char *m) {
  printf("%s\n",m);
  exit(EXIT_FAILURE);
}
```

```c
Stack *stackCreate() {
  Stack *s = malloc(sizeof(Stack));
  if (!s)
    terminate("Stack can not"\
              "be created");
  s -> size = 0;
  return s;
}

void stackFree(Stack *s) {
  free(s);
}
```


Implémentation par tableau d'une pile : code C

Accès et insertion.

```c
void stackPush(Stack *s, void *data) {
  if (s -> size >= MAX_STACK_SIZE)
    terminate("Maximum stack size"\
              "reached");

  s -> array[s -> size++] = data;
}

void *stackTop(Stack *s) {

  if (s -> size == 0)
    terminate("Stack is empty");

  return s -> array[s -> size - 1];
}
```

```c
void *stackPop(Stack *s) {
  if (s -> size == 0)
    terminate("Stack is empty");

  return s -> array[--(s -> size)];
}

int stackSize(Stack *s) {
  return s -> size;
}

int stackIsEmpty(Stack *s) {
  return (s -> size == 0);
}
```

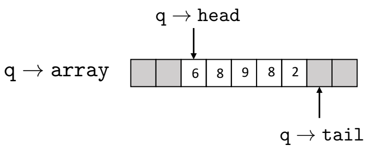
Implémentation par tableau d'une file : principe

Principes :

- Les valeurs contenues dans la file sont placées dans les composantes successives d'un tableau.
- Un indice `head` (resp. `tail`) indique la valeur en tête (resp. en queue) de file.
- La tableau est géré de manière circulaire.

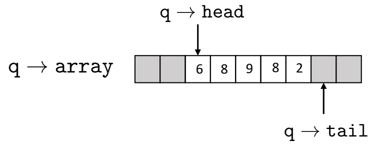
La taille du tableau doit être fixée a priori et une erreur signalée lorsque la taille de la file excède la taille du tableau.

```c
const int  MAX_STACK_SIZE = 1000;

struct Stack_t {
  void *array[MAX_STACK_SIZE];
  int   size;
};
```

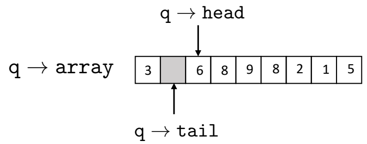


Illustration

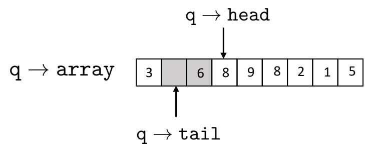
File initiale :



`queueEnqueue(q,1)`, `queueEnqueue(q,5)`, `queueEnqueue(q,3)`



`queueDequeue(q)` ⇒ 6



Implémentation par tableau d'une file : code C

Création et suppression.

```c
#include <stdlib.h>
#include <stdio.h>
#include "queue.h"

const int MAX_QUEUE_SIZE = 1000;

struct Queue_t {
  void *array[MAX_QUEUE_SIZE];
  int head, tail;
};

static void terminate(const char *m) {
  printf("%s\n",m);
  exit(EXIT_FAILURE);
}
```

```c
Queue *queueCreate() {
  Queue *q = malloc(sizeof(Queue));
  if (!q)
    terminate("Queue can not"\
              "be created");
  q -> head = 0;
  q -> tail = 0;
  return q;
}

void queueFree(Queue *q) {
  free(q);
}
```


Implémentation par tableau d'une file : code C

Accès et insertion.

```c
void queueEnqueue(Queue *q, void *data) {
  if (queueSize(q) >=
                  MAX_QUEUE_SIZE - 1)
    terminate("Queue is full");

  q -> array[q -> tail] = data;
  q -> tail = (q -> tail + 1)
              % MAX_QUEUE_SIZE;
}

void *queueHead(Queue *q) {
  if (q -> tail == q -> head)
    terminate("Queue is empty");
  return q -> array[q -> head];
}
```

```c
void *queueDequeue(Queue *q) {
  if (q -> tail == q -> head)
    terminate("Queue is empty");

  void *data = q -> array[q -> head];

  q -> head = (q -> head + 1)
              % MAX_QUEUE_SIZE;

  return data;
}

int queueSize(Queue *q) {
  return (MAX_QUEUE_SIZE + q -> tail
          - q -> head) % MAX_QUEUE_SIZE
}

int queueIsEmpty(Queue *q) {
  return (q -> head == q -> tail);
}
```

Complexité en temps et en espace

La complexité en temps de toutes les opérations est $O(1)$

La complexité en espace est en fait $O(1)$ si la pile/file contient n valeurs, qui exprime que la taille de la structure ne dépend pas de la quantité de données qui y est stockée.

Deux inconvénients :

- Il y a une limite sur le nombre de valeurs qu'on peut stocker dans la structure
- Utiliser un tableau de taille fixe entraîne un gaspillage en terme de mémoire

Pour résoudre ce problème, il faut utiliser une structure de données dynamique $\Rightarrow$ la liste liée

Plan

Liste (simplement) liée

Structure de données composée d'une séquence d'éléments de liste.

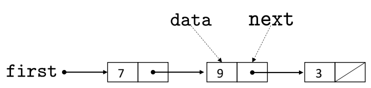
Chaque élément de liste (aussi appelé un nœud) est composé :

- d'un contenu utile de type arbitraire (les valeurs qu'on souhaite stocker dans la structure)
- d'un pointeur vers l'élément suivant dans la séquence (NULL si l'élément est le dernier de la liste)

Une liste liée est un pointeur vers le premier élément de la liste.

Exemple d'une liste liée d'entiers :

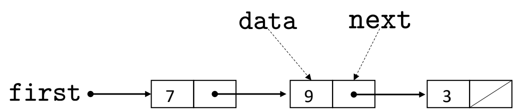
```
typedef struct Node_t {
  int data;
  struct Node_t *next;
} Node;
```



Manipulation d'une liste liée

Construction d'une liste :

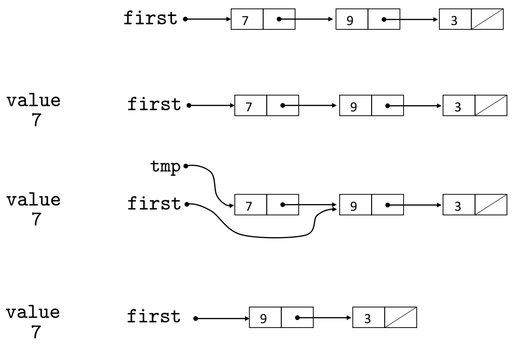
```
Node *n1 = malloc(sizeof(Node));
Node *n2 = malloc(sizeof(Node));
Node *n3 = malloc(sizeof(Node));

n1 -> data = 7;
n1 -> next = n2;
n2 -> data = 9;
n2 -> next = n3;
n3 -> data = 3;
n3 -> next = NULL;

Node *first = n1;
```



Manipulation d'une liste liée

Extraction du premier élément

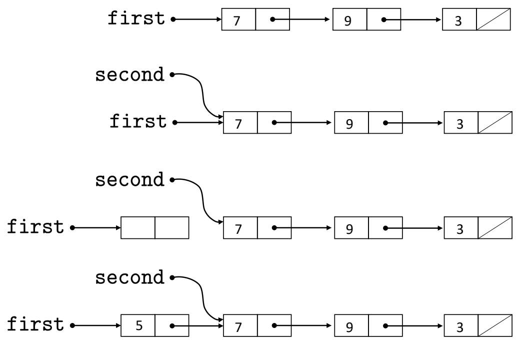
```
int value = first -> data;

Node *tmp = first;
first = first -> next;

free(tmp);
return value;
```



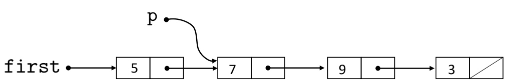
Manipulation d'une liste liée

Ajout d'un élément en début de liste

```
Node *second = first;

first = malloc(sizeof(Node));

first -> data = value;
first -> next = second;
```



Manipulation d'une liste liée : traverser la liste




```
Node *p = first;
while (p != NULL) {
  printf("%d\n", p -> data);
  p = p -> next;
}
```

Sortie :

```
5
7
9
3
```


Liste liée versus tableau

Liste liée et tableau peuvent tous deux représenter une séquence de valeurs.

Liste liée :

- Accès relatif uniquement aux éléments de la séquence (via pointeur `next`)
- Taille dépend directement (linéairement) du nombre d'élements
- Insertion aisée ($O(1)$) de valeurs au milieu de la séquence

Tableau :

- Accès direct aux éléments en fonction de leur rang
- Taille fixée à priori
- Insertion compliquée ($O(n)$) de valeurs au milieu de la séquence.

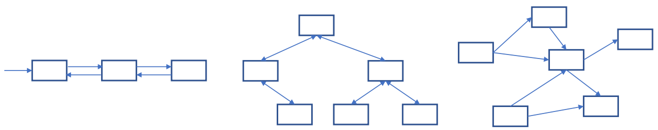
Généralisation : structures liées

Un seul pointeur (`next`) par nœud permet de représenter déjà pas mal de structures, au delà d'une simple séquence.

Le concept peut néanmoins se généraliser facilement en rajoutant d'autres pointeurs.

Exemples :

- Liste doublement liée : pointeur `previous` vers l'élément précédent. Facilite certaines opérations.
- Arbre binaire : pointeurs vers le parent, le fils gauche et le fils droit (voir plus loin).
- Graphe : pointeur vers chaque nœud adjacent.



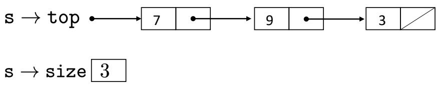
Plan

Implémentation d'une pile par liste liée

Principe :

- Les valeurs contenues dans la pile sont retenues dans une liste liée.
- L'opération `push` place la valeur en tête de liste. Les opérations `pop` et `top` travaillent en tête de liste également.
- Un indice `size` contient la taille de la pile.

```
struct Node_t {
  void          *data;
  struct Node_t  *next;
};

struct Stack_t {
  Node *top;
  int   size;
};
```



Implémentation d'une pile par liste liée

Création et suppression.

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

typedef struct Node_t {
  void           *data;
  struct Node_t *next;
} Node;

struct Stack_t {
  Node *top;
  int   size;
};

static void terminate(char *m) {
  printf("%s\n",m);
  exit(EXIT_FAILURE);
}
```

```
Stack *stackCreate() {
  Stack *s = malloc(sizeof(Stack));
  if (!s)
    terminate("Stack can not"\
              "be created");
  s -> top = NULL;
  s -> size = 0;
  return s;
}

void stackFree(Stack *s) {
  Node *n = s -> top;
  while (n) {
    Node *nNext = n -> next;
    free(n);
    n = nNext;
  }
  free(s);
}
```

Implémentation d'une pile par liste liée

Accès et insertion.

```
void stackPush(Stack *s,
               void *data) {
  Node *n = malloc(sizeof(Node));

  if (!n)
    terminate("Stack node can "\
              "not be created");

  n -> data = data;
  n -> next = s -> top;
  s -> top = n;
  s -> size++;
}

void *stackTop(Stack *s) {

  if (!(s -> top))
    terminate("Stack is empty");

  return s -> top -> data;
}
```

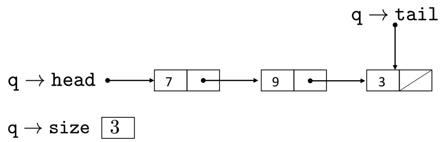
```
void *stackPop(Stack *s) {
  if (!(s -> top))
    terminate("Stack is empty");

  Node *n = s -> top;
  void *data = n -> data;

  s -> top = n -> next;
  s -> size--;

  free(n);

  return data;
}

int stackSize(Stack *s) {
  return s -> size;
}

int stackIsEmpty(Stack *s) {
  return (s -> size == 0);
}
```


Implémentation d'une file par liste liée

Principe :

- Les valeurs contenues dans la file sont retenues dans une liste liée.
- L'opération enqueue place les nouvelles valeurs en fin de liste, l'opération dequeue retire les valeurs en début de liste.
- Des pointeurs head et tail indiquent resp. le début et la fin de la liste.
- Un indice size contient la taille de la pile.

```
typedef struct Node_t {
  void          *data;
  struct Node_t *next;
} Node;

struct Queue_t {
  Node *head;
  Node *tail;
  int   size;
};
```



Implémentation d'une file par liste liée

Création et suppression.

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

typedef struct Node_t {
  void          *data;
  struct Node_t *next;
} Node;

struct Queue_t {
  Node *head;
  Node *tail;
  int   size;
};

static void terminate(char *m) {
  printf("%s\n", m);
  exit(EXIT_FAILURE);
}
```

```
Queue *queueCreate() {
  Queue *q = malloc(sizeof(Queue));
  if (!q)
    terminate("Queue can not"\
              "be created");
  q -> head = NULL;
  q -> tail = NULL;
  q -> size = 0;
  return q;
}

void queueFree(Queue *q) {
  Node *n = q -> head;
  while (n) {
    Node *nNext = n -> next;
    free(n);
    n = nNext;
  }
  free(q);
}
```


Implémentation d'une file par liste liée

Accès et insertion.

```
void queueEnqueue(Queue *q,
                  void *data) {
  Node *n = malloc(sizeof(Node));
  if (!n)
    terminate("Queue node can"\
              "not be created");
  n -> data = data;
  n -> next = NULL;

  if (q -> tail)
    q -> tail -> next = n;
  else
    q -> head = n;
  q -> tail = n;

  q -> size++;
}

void *queueHead(Queue *q) {
  if (!(q -> head))
    terminate("Queue is empty");
  return q -> head -> data;
}
```

```
void *queueDequeue(Queue *q) {
  if (!(q -> head))
    terminate("Queue is empty");

  Node *n = q -> head;
  void *data = n -> data;

  q -> head = n -> next;
  q -> size--;
  if (q -> size == 0)
    q -> tail = NULL;
  free(n);
  return data;
}

int queueSize(Queue *q) {
  return q -> size;
}

int queueIsEmpty(Queue *q) {
  return (q -> size == 0);
}
```

Implémentation par liste liée : complexité

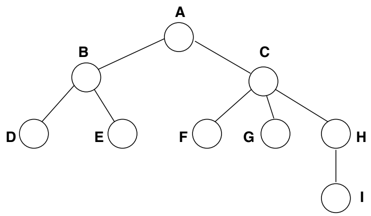
Complexité en temps est $O(1)$ pour toutes les opérations, comme pour la représentation par tableau.

Complexité en espace est $O(n)$ si la pile/file contient n éléments.

Il n'y a plus de limite a priori sur la taille de la pile/file.

Plan

Arbre : définitions



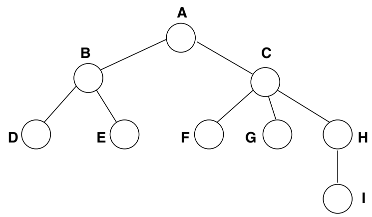
Un arbre (enraciné) T est un graphe dirigé (N, E), où :

- N est un ensemble de nœuds, et
- $E \subset N \times N$ est un ensemble d'arêtes,

possédant les propriétés suivantes :

- T est connexe et acyclique
- Si T n'est pas vide, alors il possède un nœud distingué appelé racine (*root node*). Cette racine est unique.
- Pour toute arête $(n_1, n_2) \in E$, le nœud n_1 est le parent de n_2.
  - La racine de T ne possède pas de parent.
  - Les autres nœuds de T possèdent un et un seul parent.

Arbres : définitions



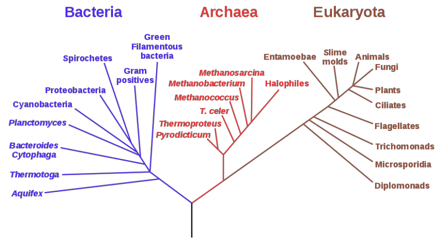
- Si n_2 est le parent de n_1, alors n_1 est le fils (*child*) de n_2.
- Deux nœuds n_1 et n_2 qui possèdent le même parent sont des frères (*siblings*).
- Un nœud qui possède au moins un fils est un nœud interne.
- Un nœud externe (c'est-à-dire, non interne) est une feuille (*leaf*) de l'arbre.

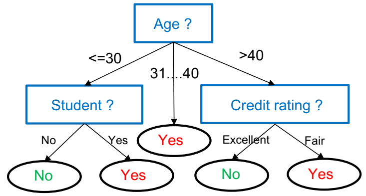
Arbre : structure de données

- Principe :
  - Des données sont associées aux nœuds d'un arbre
  - Les nœuds sont accessibles les uns par rapport aux autres selon leur position dans l'arbre
- Interface : Pour un arbre T et un nœud n
  - $\textsc{Parent}(T,n)$: renvoie le parent d'un nœud n (signale une erreur si n est la racine)
  - $\textsc{isEmpty}(T)$: renvoie vrai si l'arbre est vide
  - $\textsc{Children}(T,n)$: renvoie une structure de données (ordonnée ou non) contenant les fils du nœud n (liste, tableau, etc.)
  - $\textsc{isRoot}(T,n)$: renvoie vrai si n est la racine de l'arbre
  - $\textsc{isInternal}(T,n)$: renvoie vrai si n est un nœud interne
  - $\textsc{isExternal}(T,n)$: renvoie vrai si n est un nœud externe
  - $\textsc{GetData}(T,n)$: renvoie les données associées au nœud n
  - $\textsc{Root}(T)$: renvoie le nœud racine de l'arbre
  - Pour un arbre binaire (ordonné) :
    - $\textsc{Left}(T,n)$, $\textsc{Right}(T,n)$: renvoie les fils gauche et droit de n
    - $\textsc{hasLeft}(T,n)$, $\textsc{hasRight}(T,n)$: détermine si le nœud n possède un fils respectivement gauche et droit.

Arbre : applications

Beaucoup de données sont représentées de manière hiérarchique : arbres généalogique, arbres de décision, arbre phylogénétique, etc.





Les arbres sont utilisé comme structures de données pour de nombreux algorithmes : tri par tas, files à propriété, arbres binaires de recherche, arbre de couverture minimale, etc (voir INFO0902).

Exemple d'interface en C (arbre binaire)

BTree.h

```c
#ifndef _BTREE_H
#define _BTREE_H

typedef struct BTNode_t BTNode;
typedef struct BTree_t BTree;

BTree  *btCreate();
void    btFree(BTree *tree);
BTNode *btCreateRoot(BTree *tree, int data);
BTNode *btInsertLeft(BTree *tree, BTNode *n1, int data);
BTNode *btInsertRight(BTree *tree, BTNode *n2, int data);

BTNode *btRoot(BTree *tree);
BTNode *btLeft(BTree *tree, BTNode *n);
BTNode *btRight(BTree *tree, BTNode *n);
BTNode *btParent(BTree *tree, BTNode *n);
int     btGetData(BTree *tree, BTNode *n);
int     btSize(BTree *tree);

int     btIsRoot(BTree *tree, BTNode *n);
int     btIsInternal(BTree *tree, BTNode *n);
int     btIsExternal(BTree *tree, BTNode *n);
int     btHasLeft(BTree *tree, BTNode *n);
int     btHasRight(BTree *tree, BTNode *n);
#endif
```


Exemples d'opération sur un arbre

Calcul de la profondeur d'un nœud (sa distance, en nombre d'arêtes, à la racine)

Récursivement :

```c
int btNodeDepth(BTree *tree, BTNode *n) {
  if (btIsRoot(tree,n))
     return 0;
  else
     return 1+btNodeDepth(tree, btParent(tree, n));
}
```

Itérativement :

```c
int btNodeDepth(BTree *tree, BTNode *n) {
  int depth = 0;
  while (!btIsRoot(tree,n)) {
     n = btParent(tree, n);
     depth++;
  }
  return depth;
}
```

Complexité en temps : $O(n)$, où n est la taille de l'arbre (si les opérations de l'interface sont $O(1)$)

Exemples d'opération sur un arbre

Calcul de la hauteur de l'arbre (la plus longue distance, en nombre d'arêtes, d'un nœud à la racine)

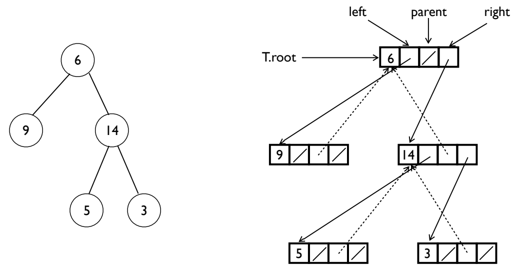
```
int btHeight(BTree *tree) {
   return btHeightAux(tree, btRoot(tree));
}

int btHeightAux(BTree *tree, BTNode *n) {
  if (btIsExternal(tree, n))
     return 0;
  else {
    int hl = 0;
    int hr = 0;
    if (btHasLeft(tree, n))
      hl = btHeightAux(tree, btLeft(tree, n));
    if (btHasRight(tree, n))
      hr = btHeightAux(tree, btRight(tree, n));
    return 1 + (hl>hr? hl: hr);
  }
}
```

Complexité en temps : $O(n)$, où n est la taille de l'arbre (si les opérations de l'interface sont $O(1)$)

Implémentation en C (arbre binaire)

En utilisant une structure liée :

- BTNode : une structure contenant pour chaque nœud `n` :
  - Un champ de données (`data`)
  - Un pointeur vers son nœud parent (`parent`)
  - Un pointeur vers ses fils gauche et droit (`left` et `right`)
- BTree : une structure contenant un pointeur vers la racine (`root`)
- Complexité en temps des opérations : $O(1)$
- Complexité en espace : $O(n)$ pour n nœuds



(généralise la notion de liste liée)

Implémentation par structure liée : création et libération

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

struct BTNode_t {
  BTNode *parent;
  BTNode *left;
  BTNode *right;
  int data;
};

struct BTree_t {
  BTNode *root;
  int size;
};

static void terminate(char *m) {
  printf("%s\n",m);
  exit(EXIT_FAILURE);
}
```

```
BTree  *btCreate() {
  BTree *tree = malloc(sizeof(BTree));
  if (!tree)
    terminate("Tree can not be created");
  tree->root = NULL;
  tree->size = 0;
  return tree;
}

static void FreeNodesRec(BTNode *n) {
  if (!n)
    return;
  FreeNodesRec(n->left);
  FreeNodesRec(n->right);
  free(n);
}

void btFree(BTree *tree) {
  FreeNodesRec(tree->root);
  free(tree);
}
```


Implémentation par structure liée : création de nœuds

```
static BTNode *createNode(int data) {
  BTNode *n = malloc(sizeof(BTNode));
  if (!n)
    terminate("Node can not be created");
  n->data   = data;
  n->left   = NULL;
  n->right  = NULL;
  n->parent = NULL;
  return n;
}

BTNode *btCreateRoot(BTree *tree,
                     int data) {
  BTNode *root = createNode(data);
  tree->root = root;
  tree->size = 1;
  return root;
}
```

```
BTNode *btInsertLeft(BTree *tree,
                     BTNode *n,
                     int data) {
  BTNode *nleft = createNode(data);
  n->left = nleft;
  nleft->parent = n;
  tree->size++;
  return nleft;
}

BTNode *btInsertRight(BTree *tree,
                      BTNode *n,
                      int data) {
  BTNode *nright = createNode(data);
  n->right = nright;
  nright->parent = n;
  tree->size++;
  return nright;
}
```

Implémentation par structure liée : accesseurs

```c
BTNode *btRoot(BTree *tree) {
  return tree->root;
}
BTNode *btLeft(BTree *tree, BTNode *n) {
  return n->left;
}
BTNode *btRight(BTree *tree, BTNode *n) {
  return n->right;
}
BTNode *btParent(BTree *tree, BTNode *n) {
  return n->parent;
}
int btGetData(BTree *tree, BTNode *n) {
  return n->data;
}
int btSize(BTree *tree) {
  return tree->size;
}
```

```c
int btIsRoot(BTree *tree, BTNode *n) {
  return (n->parent==NULL);
}
int btIsInternal(BTree *tree, BTNode *n) {
  return (n->left!=NULL || n->right!=NULL);
}
int btIsExternal(BTree *tree, BTNode *n) {
  return (n->left==NULL && n->right==NULL);
}
int btHasLeft(BTree *tree, BTNode *n) {
  return (n->left!=NULL);
}
int btHasRight(BTree *tree, BTNode *n) {
  return (n->right!=NULL);
}
```

Remarques :

- Pour plusieurs fonctions, l'argument `tree` ne sert à rien. Ca pourrait être le cas cependant pour une autre implémentation.
- On pourrait ajouter une gestion d'erreur (pour empêcher l'accès à un nœud inexistant par exemple)

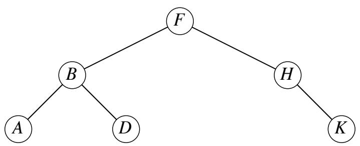
Parcours d'arbres (binaire)

Un parcours d'arbre est une façon d'ordonner les nœuds d'un arbre afin de les parcourir

Différents types de parcours :

- Parcours en profondeur :
  - Infixe (en ordre)
  - Préfixe (en préordre)
  - Suffixe (en postordre)
- Parcours en largeur

Parcours infixe



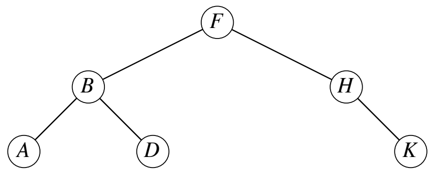
$$\Rightarrow \langle A, B, D, F, H, K \rangle$$

Parcours infixe (en ordre) : Chaque nœud est visité après son fils gauche et avant son fils droit

```c
static void btInOrderTreeWalk(BTree *tree, BTNode *n) {
  if (btHasLeft(tree, n))
    btInOrderTreeWalk(tree, btLeft(tree, n));
  printf(" %d", btGetData(tree, n));
  if (btHasRight(tree, n))
    btInOrderTreeWalk(tree, btRight(tree, n));
}
```

Appel : `btInOrderTreeWalk(tree, btRoot(tree))`

Parcours préfixe



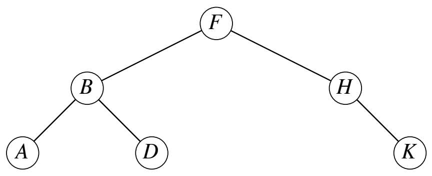
$$\Rightarrow \langle F, B, A, D, H, K \rangle$$

Parcours préfixe (en préordre) : chaque nœud est visité avant ses fils

```c
static void btPreOrderTreeWalk(BTree *tree, BTNode *n) {
  printf(" %d", btGetData(tree, n));
  if (btHasLeft(tree, n))
    btPreOrderTreeWalk(tree, btLeft(tree, n));
  if (btHasRight(tree, n))
    btPreOrderTreeWalk(tree, btRight(tree, n));
}
```

Appel : `btPreOrderTreeWalk(tree, btRoot(tree))`

Parcours postfixe



$$\Rightarrow \langle A, D, B, K, H, F \rangle$$

Parcours postfixe (en postordre) : chaque nœud est visité après ses fils

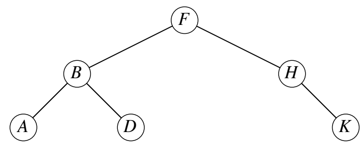
```
static void btPostOrderTreeWalk(BTree *tree, BTNode *n) {
  if (btHasLeft(tree, n))
    btPostOrderTreeWalk(tree, btLeft(tree, n));
  if (btHasRight(tree, n))
    btPostOrderTreeWalk(tree, btRight(tree, n));
  printf(" %d", btGetData(tree, n));
}
```

Appel : `btPostOrderTreeWalk(tree, btRoot(tree))`

Parcours en largeur

Parcours en largeur : on visite le nœud le plus proche de la racine qui n'a pas déjà été visité. Correspond à une visite des nœuds de profondeur 1, puis 2, ...

Implémentation à l'aide d'une file :



$$\Rightarrow \langle F, B, H, A, D, K \rangle$$

```
void btBreadthFirstTreeWalk(BTree *tree) {
  Queue *q = queueCreate();
  if (btSize(tree)>0)
    queueEnqueue(q, btRoot(tree));
  while (!queueIsEmpty(q)) {
    BTNode *n = queueDequeue(q);
    printf(" %d ", btGetData(tree, n));
    if (btHasLeft(tree, n))
      queueEnqueue(q, btLeft(tree, n));
    if (btHasRight(tree, n))
      queueEnqueue(q, btRight(tree, n));
  }
  queueFree(q);
}
```

(Exercice : Implémenter les parcours en profondeur de manière non récursive)

Complexité des parcours

Tous les parcours sont :

- $O(n)$ en temps : dans tous les cas, on ne passe qu'une et une seule fois sur chaque nœud.
- $O(n)$ en mémoire dans le pire cas *(à quoi correspondent-ils ?)*

Plan

Plan

Dictionnaire : principe

Un dictionnaire est une collection de paires (clé, valeur) où

- clé est une valeur permettant d'identifier de manière unique un élément du dictionnaire. Par exemple : un entier, une chaîne de caractères, etc.
- valeur est une valeur qu'on souhaite associer à cet élément.

Interface :

- `Insert(d, key, value)` : insère la paire (`key`,`value`) dans le dictionnaire `d`. Si la clé s'y trouve déjà, sa valeur est mise à jour.
- `Search(d, key)` : cherche la clé `key` dans le dictionnaire `d`. Si la valeur est trouvée, la valeur associée est renvoyée, sinon `NULL`.
- `Remove(d, key)` : supprime la clé `key` (et sa valeur) du dictionnaire.
- + création d'un dictionnaire vide et libération de la mémoire.
- + parcours de toutes les clés

Dictionnaire : principe

Un dictionnaire s'appelle aussi un tableau associatif ou une table de symboles.

Un dictionnaire peut être vu comme un généralisation d'un tableau dont les indices sont remplacés par n'importe quel type de clé.

`Insert(d, "Pierre",4)` $\Leftrightarrow$ `d["Pierre"] = 4`

Les clés sont souvent supposées pouvoir être ordonnées mais ce n'est pas toujours nécessaire.

Contraintes de performance implicites :

- Les opérations d'insertion et de recherche doivent être les plus efficaces possibles.
- L'espace mémoire consommé doit être minimal et idéalement s'adapter au nombre de clés.

Applications

Les applications sont très nombreuses :

- Liste de contacts : clé = nom, valeur = numéro de téléphone, adresse
- Dictionnaire : clé = mot, valeur = définition
- Recherche internet : clé = mot clé, valeur = liste de pages web
- domain name service (DNS) : clé = nom de domaine (`www.uliege.be`), valeur = adresse IP (`139.165.X.Y`)
- Compilateur : clé = nom de variable, valeur = valeur et type de la variable.
- Système de fichier : clé = nom de fichier, valeur = localisation du fichier sur le disque
- ...

Ensemble : un dictionnaire simplifié

Un ensemble (set) est une collection de clés uniques.

Équivalent à un dictionnaire, sans valeurs associées aux clés.

Interface :

- `Insert(s, key)` : ajoute la clé `key` dans l'ensemble `s` si elle ne s'y trouve pas déjà.
- `Contains(s, key)` : renvoie 1 si la clé `key` est contenue dans l'ensemble `s`, 0 sinon.
- `Remove(s, key)` : supprime la clé `key` de l'ensemble `s`.
- + création et libération de la structure.
- + parcours des clés de l'ensemble.

Les techniques d'implémentation d'un dictionnaire peuvent s'adapter trivialement à l'ensemble.

Exemple d'interface en c

En supposant des clés de type `int` et des valeurs de type `void *`.

`dict.h`

```
#ifndef _DICT_H
#define _DICT_H

typedef struct Dict_t Dict;

Dict *dictCreate();
void   dictFree(Dict *);
void   dictInsert(Dict *, int, void *)
void  *dictSearch(Dict *, int);
int    dictContains(Dict *, int);
void  *dictRemove(Dict *, int);

#endif
```

`set.h`

```
#ifndef _SET_H
#define _SET_H

typedef struct Set_t Set;

Set *setCreate();
void   setFree(Stack *);
void   setInsert(Set *, int);
int    setContains(Set *, int);
void  *setRemove(Set *, int);

#endif
```

Remarque : on ne peut pas remplacer le type de la clé par un type `void *`, sauf à rajouter des pointeurs de fonctions en argument aux fonctions `dicoCreate` et `setCreate`. Par exemple pour passer une fonction de comparaison des clés.

Application 1 : filtrage d'adresses emails

(voir Partie 5, slide 239)

```
...
#include "set.h"

int main() {
  char buffer[10000];

  // construction de l'ensemble
  Set *s = setCreate(100);
  while (fgets(buffer, 1000, stdin)) {
    int lenstr = strlen(buffer) - 1;
    buffer[lenstr]='\0';
    setInsert(s, strdup(newstring));
  }

  // filtrage
  if (setContains(s, "p.geurts@uliege.be")) {
    ...
  }
  setFree(s);
}
```

Utilisation : `./filter < adresses.txt`

Application 2 : suppression des doublons dans un fichier

```
...
#include "set.h"

int main() {
  char buffer[1000];
  Set *s = setCreate(100);
  while (fgets(buffer, 1000, stdin)) {
    int lenstr = strlen(buffer)-1;
    buffer[lenstr]='\0';

    if (!setContains(s, buffer)) {
      printf("%s\n", buffer);
      setInsert(s, strdup(newstring));
    }
  }
  setFree(s);
}
```

Utilisation : `./removeduplicates < list.txt > listunique.txt`

Application 3 : compter la fréquence des mots

```c
...
#include "dict.h"

int main() {
  char buffer[1000];
  Dict *d = dictCreate(25000);

  while (fgets(buffer, 1000, stdin)) {
    char *string = buffer;
    char *token;
    while ((token = strsep(&string," \t\n,;.:?\"\r*-_()[]'")) != NULL) {
      if (strlen(token)>0) {
        int c = dictSearch(d, token);
        if (c == -1) {
          dictInsert(d, strdup(token), 1);
        } else {
          dictInsert(d, token, c+1);
        }
      }
    }
  }
  dictPrint(d); // affiche le contenu du dictionnaire
  dictFree(d);
}
```

Utilisation : `./countwords < le_rouge_et_le_noir.txt > countings.csv`

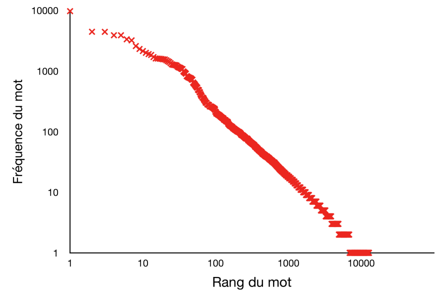
Application 3 : compter la fréquence des mots

Application en linguistique, compression de données, etc.

La loi de Zipf[4] dit que la fréquence du $n$ème mot le plus fréquent dans un texte est $f(n) = \frac{K}{n}$ où K est une constante.

Exemple : "le rouge et le noir" de Stendhal[5] (180000 mots dont 13000 uniques)

Mot	Rang	Freq.
de	1	9738
il	2	4454
la	3	4439
le	4	3902
à	5	3898
l	6	3344
et	7	3212
un	8	2587
que	9	2315
d	10	2140



4. https://fr.wikipedia.org/wiki/Loi_de_Zipf
5. Téléchargé de https://www.gutenberg.org

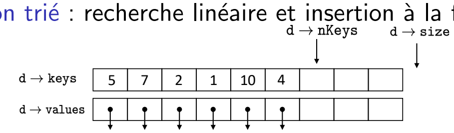
Plan

Implémentation par tableau

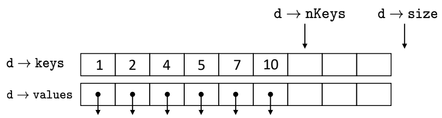
```c
struct Dict_t {
  int *keys;            // clés à valeurs entières
  void **values;        // valeurs de type (void *)
  unsigned int size;    // taille du tableau
  unsigned int nKeys;   // nombre de clés
};
```

Deux options :

- Tableau de clés non trié : recherche linéaire et insertion à la fin



- Tableau de clés trié : recherche dichotomique et insertion en décalant vers la droite.



(Implémentation laissée comme exercice)

Complexité

En temps : *(en fonction du nombre de clés n dans le dictionnaire)*

- Tableau non trié :
  - Recherche : $O(n)$
    - On parcourt le tableau jusqu'à trouver la clé recherchée ($O(n)$).
  - Insertion : $O(n)$
    - On cherche la clé dans le tableau ($O(n)$) : si présente, on écrase sa valeur ($O(1)$), sinon on la place à la fin du tableau ($O(1)$).
- Tableau trié :
  - Recherche : $O(\log n)$
    - On utilise la recherche dichotomique.
  - Insertion : $O(n)$
    - On insére la clé en fin de tableau et on la fait remonter vers la gauche jusqu'à sa position dans l'ordre (cf. tri par insertion)

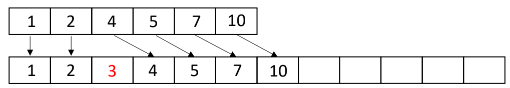
En espace : $O(1)$, car la taille du tableau ne dépend pas du nombre de paires (clé, valeur) stockées.

Tableau extensible

Problème de l'approche précédente : la taille du tableau est fixée et n'évolue pas en fonction des données (complexité en espace $O(1)$).

Solution :

- Initialisez les tableaux `keys` et `values` à une certaine taille.
- Quand les tableaux deviennent trop petits :
  - On alloue des nouveaux tableaux, deux fois plus grands.
  - On recopie les anciennes clés et valeurs dans ces nouveaux tableaux.
  - On libére les anciens tableaux.



Complexités :

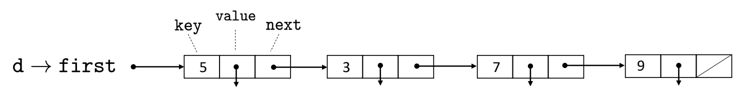
- en temps : $O(n)$ (inchangée)
- en espace : $O(n)$ (au pire, on gaspille $O(n)$).

Plan

Implémentation par liste liée

```
typedef struct Node_t {
  int key;              // clés à valeurs entières
  void *value;          // valeurs de type (void *)
  struct Node_t *next;
} Node;

struct Dict_t {
  Node *first;
  unsigned int nKeys;  // nombre de clés (optionnel)
};
```

Recherche en parcourant la liste, insertion en début de liste.



(Implémentation laissée comme exercice)

Complexités

En temps : *(en fonction du nombre de clés n dans le dictionnaire)*

- Recherche : $O(n)$
  - On parcourt la liste depuis le début et on s'arrête quand on trouve la clé recherchée (ou quand la fin de liste est atteinte).
- Insertion : $O(n)$
  - On recherche la clé dans la liste ($O(n)$). Si présente, on écrase sa valeur ($O(1)$), sinon on l'ajoute en début de liste ($O(1)$).

En espace : $O(n)$

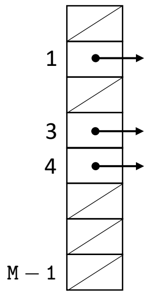
- L'espace mémoire nécessaire croît linéairement avec le nombre de paires stockées

Plan

Implémentation par tableau : deuxième version

Si on suppose que les clés prennent des valeurs entre 0 et $M-1$ avec M petit, on peut obtenir une implémentation beaucoup plus efficace.

```
struct Dict_t {
  void *values[M];
}

void dictInsert(Dict d, int key, void *value) {
  d -> values[key] = value;
}

void *dictSearch(Dict d, int key) {
  return values[key];
}
```



Complexités :

- en temps : $O(1)$ pour la recherche et l'insertion.
- en espace : $O(1)$, car l'espace mémoire ne dépend pas du nombre de clés.

Table de hachage : fonction de hachage

Problème de l'approche précédente : les clés doivent être entières et prendre des valeurs pas trop grandes.

Idée 1 : Soit U l'ensemble des valeurs possibles de clés, même non entières. On définit une fonction de hachage h :

$$h : U \rightarrow \{0, \ldots, M-1\}$$

envoyant chaque clé $k \in U$ vers une position $h(k)$ dans la table.

Insertion et recherche modifiées (toujours $O(1)$ si h est $O(1)$) :

```
void dictInsert(Dict d, int key, void *value) {
  d -> values[h(key)] = value;
}

void *dictSearch(Dict d, int key) {
  return values[h(key)];
}
```

Table de hachage : gestion des collisions

Problème : Sans connaître les clés, il est difficile de garantir que $h(k_1) \neq h(k_2)$ lorsque $k_1 \neq k_2$. Et c'est impossible dès que $|U| > M \Rightarrow$ le code précédent est incorrect.

Idée 2 : Chaque case i de la table contient une liste liée retenant toutes les clés k insérées telles que $h(k) = i$.

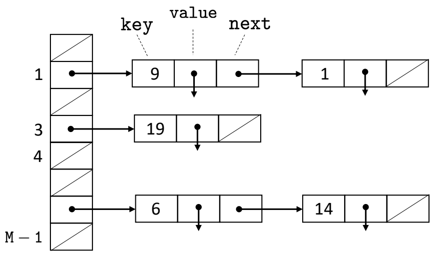


Table de hachage : implémentation (structure et création)

```
typedef struct Node_t {
  int key;
  void *value;
  struct Node_t *next;
} Node;

struct Dict_t {
  Node **array;
  unsigned int arraySize; // taille du tableau
  unsigned int nKeys;     // nombre de clés (optionnel)
};

Dict *dictCreate(int m) {
  Dict *d = malloc(sizeof(Dict));
  if (d == NULL) exit(-1);
  d -> array = calloc(m, sizeof(Node));
  if (d -> array == NULL) exit(-1);
  d -> arraySize = m;
  d -> nKeys = 0;
  return d;
}
```

Remarque : Contrairement à `malloc`, `calloc` initialise la mémoire à 0 (ou NULL) en plus de faire l'allocation. Nécessaire ici !

Table de hachage : implémentation (recherche)

```
void *dictSearch(Dict d, int key, void *value) {
  Node *p = d -> array[h(d, key)];
  while (p != NULL && p -> key != key)
    p = p -> next;

  if (p != NULL)
    return p -> value;
  else
    return NULL;
}
```

Remarque : La fonction de hachage `h` prend la table en argument (voir plus loin).

Table de hachage : implémentation (insertion)

```
void *dictInsert(Dict d, int key, void *value) {
  int hashval = h(d,key);
  Node *p = d -> array[hashval];
  while (p != NULL && p -> key != key)
    p = p -> next;
  if (p != NULL)
    p -> value = value;
  else {
    Node *newNode = malloc(sizeof(Node));
    newNode -> key = key;
    newNode -> value = value;
    newNode -> next = d -> array[hashval];
    d -> array[hashval] = newNode;
    d -> nKeys++;
  }
}
```

Complexité : éléments d'analyse

La complexité dépend de la taille M de la table et de la fonction de hachage (supposée $O(1)$).

Dans le pire cas :

- Toutes les clés sont envoyées dans la même case.
- $\Rightarrow$ Insertion et recherche en $O(n)$.

Dans le meilleur cas :

- Les clés sont réparties uniformément dans toutes les cases de la table.
- $\Rightarrow$ Insertion et recherche en $O(n/M)$.
- $\Rightarrow$ Proche de $O(1)$ si M est $O(n)$.

Complexité en espace : $O(M+n)$ (pour la table et pour les éléments de liste liée).

Fonction de hachage

Le choix de la fonction de hachage détermine l'efficacité des opérations :

- Elle doit être facile à calculer (c'est-à-dire $O(1)$).
- Elle doit répartir les clés aussi uniformément que possible dans la table (très difficile à assurer).

Lorsque les clés sont à valeurs entières, une approche simple est la méthode de division :

$$h(k) = k \mod M.$$

En C :

```
static unsigned int h(Dict *d, int key) {
  return key % d->arraySize;
}
```


Fonction de hachage : chaînes de caractères

Lorsque la clé n'est pas à valeur entière, il faut d'abord passer par une fonction d'encodage.

Exemples pour les chaînes de caractères :

- On additionne les caractères de la chaîne (naïf) :

```
static unsigned int h(Dict *d, char *key) {
  unsigned int hash = 0;
  while (*key != '\0') {
    hash += *key;
    key++;
  }
  return hash % d->arraySize;
}
```

- Une meilleure approche (`djb2`) :

```
static unsigned int h(Dict *d, char *key) {
  unsigned long hash = 5381;
  while (*key != '\0') {
    hash = hash * 33 + *key;
    key++;
  }
  return hash % d->arraySize;
}
```


Implémentation générique

On peut définir une table de hachage plus générique en stockant dans la structure une fonction de comparaison et une fonction d'encodage de clés (passées en arguments à `dictCreate`) :

```
typedef struct Node_t {
  void *key;
  void *value;
  struct Node_t *next;
} Node;

struct Dict_t {
  Node **array;
  unsigned int arraySize;
  unsigned int nKeys;

  int (*compareFunction)(const void *key1, const void *key2);
  unsigned long (*encodeKey)(const void *key);
};

static unsigned int h(Dict *d, void *key) {
  return d -> encodeKey(key) % d->arraySize;
}
```

Plan

Application au filtrage d'adresses email

Quelle est la complexité du code du slide 350 en fonction de l'implémentation de l'ensemble ?

Création et remplissage de l'ensemble :

- $O(n^2)$ dans le pire cas pour toutes les implémentations
- $O(n \log n)$ avec l'implémentation tableau si on trie le tableau seulement quand toutes les adresses ont été lues (possible seulement si toutes les adresses sont connues à l'avance).
- $O(n)$ avec la table de hachage si fonction de hachage uniforme et table suffisamment grande.

Recherche d'une adresse :

- $O(n)$ dans le pire cas pour la liste liée et la table de hachage.
- $O(\log n)$ pour le tableau trié.
- $O(1)$ avec la table de hachage si fonction de hachage uniforme et table suffisamment grande.

(Complexité de la suppression de doublons et du comptage de mots ?)

Application au filtrage d'adresses email

Tests empiriques :

- Génération de données : n chaînes de caractères (`a-z`) aléatoires de longueur 10.
- Requêtes : $10n$ chaînes prises au hasard dans la liste (recherches positives) ou en dehors (recherches négatives).

Tableau trié (par fusion) et recherche dicho. versus table de hachage ($M = 2n$, `djb2`) :

Création de la structure :

n	Tri (s)	Table (s)
12500	0,003	0,002
25000	0,005	0,002
50000	0,012	0,007
100000	0,026	0,013
200000	0,057	0,036
400000	0,128	0,085

Recherches positives

n	Rech. dico. (s)	Table (s)
500000	3,09	1,31
1000000	7,95	2,89

Recherches négatives

n	Rech. dico. (s)	Table (s)
500000	3,76	1,23
1000000	9,21	2,72

La table de hachage permet de traiter trois fois plus d'adresses à la seconde.

Conclusion

L'implémentation par table de hachage est en pratique plus efficace que les implémentations par liste liée et par tableau, malgré une complexité dans le pire cas identique, voire moins bonne.

Les performances de la recherche dans un tableau trié sont plus stables mais l'insertion est beaucoup plus couteuse, à cause du décalage.

D'autres structures existent qui permettent de combiner la facilité d'insertion de la liste liée avec la rapidité et la stabilité de la recherche dichotomique. Par exemple les arbres binaires de recherche (INFO0902).

Il existe aussi des structures spécifiques pour les chaînes de caractères. Par exemple les tries.

Partie 7

Langages de programmation

23 décembre 2019

Plan

1. Langages de programmation
2. Catégorisation des langages
3. Comparaison des langages

Plan

1. Langages de programmation
2. Catégorisation des langages
3. Comparaison des langages

Langages de programmation

Un langage de programmation est une notation permettant de décrire des calculs d'une manière compréhensible à la fois pour l'ordinateur et pour le programmeur.

Les premiers langages ont été inventés à la fin de seconde guerre mondiale (Plankalkül, 1945).

Depuis lors, une multitude de langages ont été inventé. Wikipedia liste ±680 langages différents.

Source

Pourquoi autant de langages ?

- Beaucoup de langages sont spécifiques à une machine
- De nouveaux langages sont constamment introduits pour résoudre les limitations (supposées ou non) des langages existants.
- Les langages doivent suivre les évolutions des machines et les nouveaux développements en génie logiciel
- Pour des questions de hype et de marketing
- Le nombre de degrés de liberté pour le design d'un langage est très important.
- Beaucoup de langages sont spécialisés :
  web (JS, PHP, HTML), bases de données (SQL), texte (PERL, TeX), banques (COBOL), maths (Matlab, Mathematica), GPU (Cuda, OpenCL), etc.

Pourquoi apprendre plusieurs langages ?

- Il n'existe pas de langage universel ultime qui conviendrait pour tout.
- Pour étoffer sa boîte à outils et aborder plus efficacement de nouveaux problèmes.
- Pour s'adapter à un domaine/environnement de travail, pour pouvoir réutiliser/améliorer les outils de ce domaine.
- Pour apprendre d'autre paradigmes de programmation.
  - Le langage utilisé façonne le processus de raisonnement
  - Différents problèmes sont résolus de manière efficace par différentes approches.
- Tant qu'on n'a pas testé un langage, on ne sait pas ce qu'on manque.
- Pour le challenge intellectuel.

Plan

Catégorisation des langages

Les langages sont généralement caractérisés selon un ensemble de propriétés, par exemple :

- Expressivité
- Niveau d'abstraction
- Paradigme de programmation
- Processus de compilation/exécution des programmes
- Approche de gestion de la mémoire
- Système de typage des variables/valeurs
- ...

Le **C** est un langage turing complet, de bas niveau, à typage statique faible, de type impératif procédural, compilé et dont la mémoire est gérée manuellement.

Expressivité

Le C est un langage...**Turing-complet**...

La théorie de l'informatique étudie la classe des fonctions qui sont calculables par une procédure effective, c'est-à-dire un suite finie d'instructions qui renvoie toujours une réponse correcte.

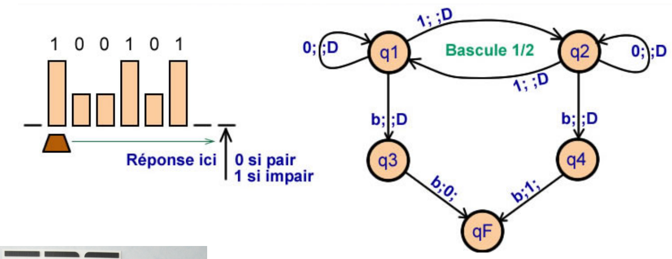
Thèse de Turing-Church : une fonction est calculable par une procédure effective si et seulement si elle est calculable par une machine de Turing (voir le slide suivant)

Un langage de programmation est Turing-complet s'il est au moins aussi expressif qu'une machine de Turing.

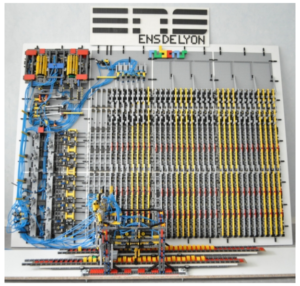
Pas déterminant dans le choix d'un langage car la plupart des langages pratiques sont Turing-complet.

Machine de Turing

Programme calculant la parité du nombre de 1 dans une séquence :



Source : Le monde, 2017



Source : Wikipedia

Niveau d'un langage

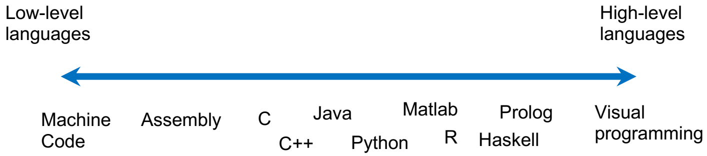
C est un langage...**de (relativement) bas niveau**...

Un langage est d'autant plus de **bas niveau** qu'il est proche du langage machine

- Contrôle fin de l'exécution et efficacité mais programmation plus laborieuse.

Un langage est d'autant plus de **haut niveau** qu'il est proche du langage humain

- Programmation plus facile mais contrôle et efficacité moindre.



Paradigmes de programmation

Le C est un langage...**impératif procédural**...

Il existe de multiple styles (= paradigmes) de programmation

Principaux paradigmes :

- Impératif :
  - Procédural : C, fortran...
  - Orienté objet : C++, Java...
- Déclaratif :
  - Fonctionnel : Lisp, Scheme, OCaml, Haskell...
  - Logique : Prolog, Oz, Coq...

Historiquement, les langages étaient mono-paradigmes. De nos jours, la plupart des langages sont multi-paradigmes, même si tous les paradigmes ne sont pas implémentés avec la même efficacité.

Programmation impérative procédurale *C, fortran...*

La programmation impérative consiste à décrire les calculs comme une série de commandes/instructions modifiant directement l'état du programme (les valeurs stockées dans la mémoire).

Procédural signifie que les séquences d'instructions sont organisées en procédures, fonctions ou sous-routines qui peuvent s'appeler les unes les autres.

Paradigme très populaire, généralement des langages simples de bas niveau. Pas adapté pour des projets de grandes envergures.

Programmation impérative orientée objet *C++, Java...*

Impérative : le programme modifie directement l'état du programme (les champs associés aux objets).

Orientée objet : la programmation est centrée sur les (types de) données.

- On identifie les "choses" qui font partie du problème ou de la solution.
- On décrit ces choses par des variables (champs).
- Ces choses interagissent avec le monde via des méthodes.

Programmation orientée "nom", plutôt que "verbe", contrairement au procédural.

Basé sur l'encapsulation : les données (objets) et les méthodes associées sont définies conjointement (=classes) et l'accès direct aux données est interdit (= types opaques).

Très bien adaptée à de gros projets collaboratifs. Très utilisée dans l'industrie.

Exemple en C

L'utilisation de types de données abstraits et de types opaques en C permet d'imiter certains mécanismes de la programmation orientée objet.

`complex.h`

```
typedef struct Complex_t Complex;

Complex *complex_new(double, double);
void     complex_free(Complex *);

double   complex_re(Complex *);
double   complex_im(Complex *);

Complex *complex_plus(Complex *, Complex *);
Complex *complex_times(Complex *, Complex *);
double   complex_abs(Complex *);
...
```

`complex.c`

```
#include <math.h>
#include "complex.h"

struct Complex_t {
  double re, im;
};

Complex *complex_new(double re, double im) {
  Complex *c = (Complex *)malloc(sizeof(Complex));
  c->re = re;
  c->im = im;
  return c;
}

double complex_re(Complex *a) {
  return a->re;
}

Complex *complex_plus(Complex *a, Complex *b) {
  double real = a->re + b->re;
  double imag = a->im + b->im;
  return complex_new(real, imag);
}
...
```


Exemple en Java

`Complex.java`

```
public class Complex
{
  private final double re; // type opaque
  private final double im;
  public Complex(double real, double imag)
  {  re = real; im = imag;  }
  public Complex plus(Complex b)
  {  // Return the sum of this number and b.
    double real = re + b.re;
    double imag = im + b.im;
    return new Complex(real, imag);
  }
  public Complex times(Complex b)
  {  // Return the product of this number and b.
    double real = re * b.re - im * b.im;
    double imag = re * b.im + im * b.re;
    return new Complex(real, imag);
  }
  public double abs()
  {  return Math.sqrt(re*re + im*im);  }
  public double re()  { return re; }
  public double im()  { return im; }
  public String toString()
  {  return re + " + " + im + "i";  }
}
```

Exemple d'utilisation

```
Complex z0 = new Complex(1.0, 1.0);
Complex z = z0;
z = z.times(z).plus(z0);
z = z.times(z).plus(z0);
StdOut.println(z);
```

Remarques :

- `new` replace le `malloc`. Pas besoin de `free` (voir plus loin)
- Méthodes directement associées aux objets ($\neq$ procédures appliquées aux objets).
- En Java, pas de pointeurs.

Programmation fonctionnelle *Lisp, Scheme, OCaml, Haskell...*

Approche plus abstraite de la programmation. on décrit quoi calculer plutôt que comment le calculer (= déclaratif).

Un programme calcule en évaluant des fonctions, au sens mathématique du terme.

Caractéristiques principales :

- Fonctions d'ordre supérieur : fonctions pouvant prendre comme arguments des fonctions et/ou en renvoyer en sortie.
- La récursion est utilisée pour la construction de boucles.
- Les listes sont la structures de base (map/reduce/filter)
- Pas d'effets de bord et d'état partagé ($\neq$ impératif)

Avantages : idempotence des fonctions, optimisation, test et déboguage plus aisés, permet le calcul concurrent, concision, etc.

Pas "mainstream" mais des touches de fonctionnelles sont présentes dans beaucoup de langages multi-paradigmes (Python, Rust, Swift, Javascript...).

Fonctions d'ordre supérieur en Python

Python n'est pas un langage purement fonctionnel mais il en a quelques caractéristiques.

Fonctions d'ordre supérieur :

```
def maxVal(f, g, x):
   return max(f(x), g(x))

def f(x):
   return x+2

g = lambda x: 6

maxVal(f,g,3) #=> 6
```

```
def maxFun(f, g):
   def maximumFunction(x):
      return max(f(x), g(x))
   return maximumFunction

maxFun(f,g)(3)    #=> 6
mv = maxFun(f,g)
mv(3)             #=> 6
```

- `lambda` permet de définir une fonction non nommée.
- MaxFun renvoie une fermeture (*closure*), une fonction avec des variables liées (ici, `f` et `g`).

Listes, map, filter et reduce

```
l = [2, 18, 9, 22, 17, 24, 8, 12, 27]
print(list(filter(lambda x: x % 3 == 0, l)))
#[18, 9, 24, 12, 27]
print(list(map(lambda x: x * 2 + 10, l)))
# [14, 46, 28, 54, 44, 58, 26, 34, 64]
print(reduce(lambda x,y: x + y, foo))
# 139
```

- `map(f,l)` applique `f` à tous les éléments de `l`, `filter(f,l)` renvoie la liste des éléments tels que `f(x)==True`.
- `reduce(f,[x1,x2,...,xN])` renvoie

$$f(f(...f(f(f(x1,x2),x3),x4),...),xN)$$

- `map` et `filter` sont évalués paresseusement, c'est-à-dire seulement quand on y accède (via `list` par exemple).

Test de primalité en Python : approche impérative

Un nombre naturel n est premier si et seulement si :

$$\nexists k \in [2, n[\colon n \bmod k = 0.$$

Solution impérative procédurale en python :

```
def is_prime(n):
   k = 2
   while k < n:
      if n % k == 0:
         return False
      k += 1
   return True
```

Liste d'instructions à exécuter pas à pas, effets de bords locaux (incrémentation de `k`).

Test de primalité en Python : approche fonctionnelle

```
def is_prime(n):
   return not any(filter(lambda k: n%k==0, range(2,n)))

def primes(m):
   return list(filter(is_prime, range(1,m)))

print(primes(20))
#[1, 2, 3, 5, 7, 11, 13, 17, 19]
```

Remarques :

- `range(n,m)` renvoie la liste `[n,n+1,...,m-1]`.
- `any` renvoie `True` si au moins une valeur (non `False`) dans la liste, `False` si la liste est vide

Solution particulièrement compacte et lisible (car proche de la définition).

Une autre solution possible en Python, encore plus lisible :

```
def is_prime(n):
   return not any(n%k==0 for k in range(2,n))
```


Programmation logique

Encore plus déclaratif que le fonctionnel (Rappel : on décrit le *quoi* plutôt que le *comment*).

Inclut plusieurs paradigmes :

- Programmation logique : principalement logique booléenne *Prolog*
- Programmation sous contraintes : contraintes numériques *Oz, AMPL*
- Assistants de preuve ("theorem provers") *Coq, Agda, Isabelle*

Langages généralement assez inefficaces pour des tâches classiques, basés sur un algorithme de recherche/optimisation.

Plus spécialisés que les autres paradigmes.

Exemples en Prolog : 3SUM et test de primalité

3SUM problem :

```
triplet0([A,B,C],L) :-
                member(A,L),
                member(B,L),
                member(C,L),
                all_distinct([A,B,C]),
                A+B+C =:= 0.

?- triplet0([A,B,C],[1,2,3,-2,-3,5,6,-1]).
```

Test de primalité :

```
is_prime(P) :- P < 4.
is_prime(P) :- integer(P), P > 3, not(has_factor(P,3)).
has_factor(N,L) :- N mod L =:= 0.
has_factor(N,L) :- L < N-1, L2 is L + 1, has_factor(N,L2).
```


Exemples en Prolog : puzzle

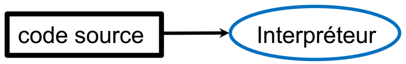
```
/* Houses logical puzzle: who owns the zebra and who drinks water?
   1) Five colored houses in a row, each with an owner, a pet,
   cigarettes, and a drink.
   2) The English lives in the red house.
   3) The Spanish has a dog.
   4) They drink coffee in the green house.
   ...
   15) The Norwegian lives near the blue house.
Who owns the zebra and who drinks water?*/
houses(Hs) :- length(Hs, 5),                                 %  1
        member(h(english,_,_,_,red), Hs),                    %  2
        member(h(spanish,dog,_,_,_), Hs),                    %  3
        member(h(_,_,_,coffee,green), Hs),                   %  4
        member(h(ukrainian,_,_,tea,_), Hs),                  %  5
        next(h(_,_,_,_,green), h(_,_,_,_,white), Hs),        %  6
        ...
        next(h(norwegian,_,_,_,_), h(_,_,_,_,blue), Hs),  % 15
        member(h(_,_,_,water,_), Hs),    % one of them drinks water
        member(h(_,zebra,_,_,_), Hs).       % one of them owns a zebra
?- zebra_owner(Owner).
?- water_drinker(Drinker).
```

Langages compilés vs interprétés

Le C est un langage...**compilé**...

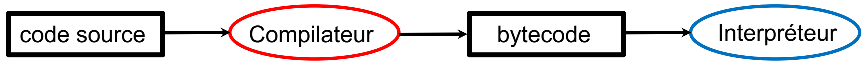
Langage compilé : un compilateur transforme le code source en un code machine qui peut être exécuté efficacement. P.ex., C, C++, fortran.

Langage interprété : un interpréteur lit et exécute (plus inefficacement) à la volée le code source. P.ex., Matlab, R, Python, Prolog.



Approche hybride (bytecode)

Pour certains langages, un compilateur transforme le code source en un code intermédiaire, le bytecode, qui peut être exécuté plus efficacement par un interpréteur.



Exemples : Java, Python.

Avantage principal : portabilité du bytecode, plus efficace que l'interpréteur direct.

Interprétation peut être interactive (Python) ou pas (Java).

Typage

Le C est un langage...**à typage statique faible**...

Le typage d'un langage définit comment sont gérés les types des variables :

- Statique (p.ex., C, Fortran, Java)
  - Toutes les variables ont un type.
  - Le système contrôle les erreurs de type lors de la compilation.
  - Plus contraignant mais plus efficace et prévient les erreurs.
- Dynamique (p.ex., Python)
  - Les valeurs, pas les variables, ont un type.
  - Le système contrôle les erreurs de types lors de l'exécution du code.
  - Plus souple mais moins efficace et plus sujet aux erreurs.

Le typage est fort si toutes les erreurs de type sont détectées (p.ex, Java), faible si des opérations peuvent être réalisées sur des valeurs de mauvais types (p.ex, C).

Typage : exemple

En C :

```c
char *a = "coucou";
a = 4;                // Génère un warning à la compilation
int c = b * a;        // Génère une erreur à la compilation
int d = b * (int)a;   // Pas de soucis
short e = d;          // Pas de soucis
```

En Java :

```java
int b = 4
short d;
d = b;                // Génère une erreur à la compilation
```

En Python :

```python
a = 'CI'
b = 4
c = b * a            # c contient 'CICICICI'
d = 4 + 'coucou'     # génère une erreur à l'exécution
```

Inférence de type

Certains langages à typage statique inférent le type des variables lors de la compilation. Le type des variables/arguments ne doit pas être précisé explicitement dans le code.

Exemple en OCaml :

```
let rec length = function
[] -> 0
| h::t -> 1+(length t);;

let myf() = length 4;;
```

Malgré le fait que les arguments des fonctions ne soient pas typés, ce code génèrera une erreur à la *compilation*.

Gestion de la mémoire

Deux approches :

- Manuelle : l'utilisateur a la responsabilité d'allouer et de désallouer la mémoire
  - P.ex., `malloc` et `free` en C
  - Laborieux pour le programmeur mais il a un contrôle total
- Automatique : la mémoire doit toujours être allouée mais la désallocation est automatique.
  - L'algorithme qui s'occupe de désallouer la mémoire s'appelle le ramasse-miettes ("garbage collector").
  - Pas de fuite mémoire possible mais moins efficace et pas de contrôle de la mémoire par le programmeur

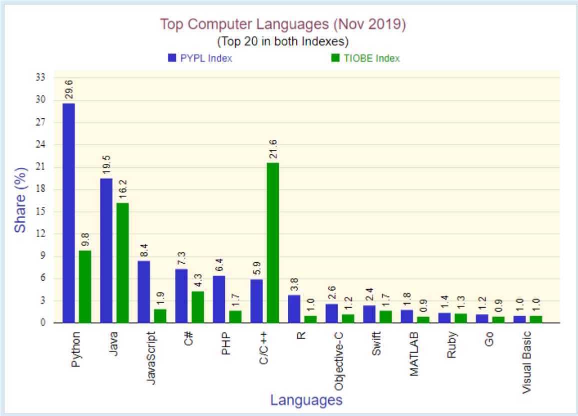
Exemple : réutilisation d'un nom de tableau, C versus Java :

```
double arr[] = malloc(5, sizeof(double));
...
free(arr);
arr = malloc(10, sizeof(double));
```

```
double[] arr = new double[5];
...
arr = new double[10];
```


Plan

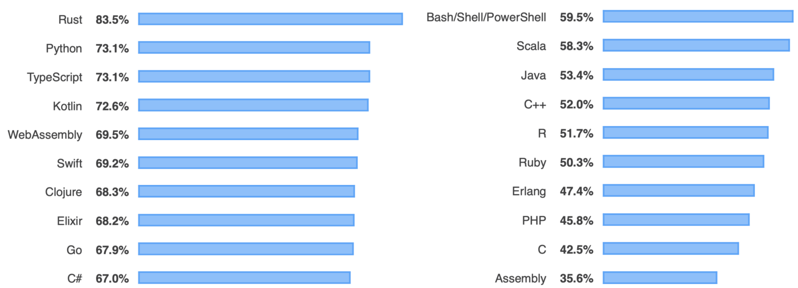
Les langages les plus populaires



Source

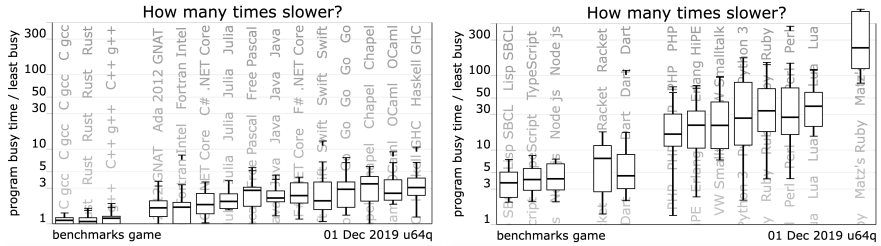
Deux indices de popularité : PYPL (recherche d'aide sur google) et TIOBE (basé sur plus de sources).

Les langages les plus aimés



Sondage sur stackoverflow (en 2019) : pourcentage de développeurs utilisant le langage et ayant exprimé avoir l'intention de continuer de devélopper dans ce langage. (Seulement le top 10 et le bottom 10 du classement).

Les langages les plus performants



Source : The computer langage benchmark games

- L'ordonnée montre combien de fois plus lente est la meilleure solution à chaque tâche en moyenne par rapport à la meilleur solution parmis tous les langages.
- Pas toujours le critère le plus pertinent.

Quel langage et pour quoi ?

Parmi les langages généralistes, orientés applications scientifiques :

- C/C++ : Valeurs sûres. Efficace. C++ mieux adapté pour des gros projets.
- Java : Efficace. Très utilisé dans les entreprises de développement logiciel.
- Python : Très utilisé en recherche. Prototypage très facile, multi-paradigmes, interprété, énormément de bibliothèques externes...
- Matlab : Utilisé dans le monde académique et dans les sciences de l'ingénieur. Beaucoup de bibliothèques dédiées à l'ingénieurie. Gros défaut : payant (le seul de cette liste).
- R : Orienté sciences des données et statistiques. Interprété, beaucoup de bibliothèques et outils de visualisation.
- Julia : Orienté calcul scientifique (comme Matlab). Très efficace, gratuit. Manque de bibliothèques, en comparaison avec Python ou Matlab.

Pour les plus aventureux :

- Rust : Langage qui monte. Multi-paradigme, sécurisé, concurrent, très efficace et peu gourmand en mémoire. Manque pour l'instant de bibliothèques.
- Haskell, OCaml : Pour ceux qui veulent tester la programmation fonctionnelle.

Programmations et langages dans les cours d'informatique

Bloc 2 :

- INFO0902 - Structures de données et algorithmes : Impérative procédurale en C.
- INFO0062 - Object-oriented programming : Orientée objet (en Java).

Bloc 3 :

- INFO0012 - Computation structures : langage d'assemblage, parallèle.
- INFO0004 - Projet de programmation orientée-objet : Orientée objet (en C++).
- INFO0054 - Programmation fonctionnelle : Fonctionnelle (en Scheme).
- INFO8006 - Introduction to artificial intelligence : Logique (un peu), projets en Python.
- INFO009 - Base de données : Langage d'interrogation de base de données (SQL).

Dans les cours d'informatique

En master :

- INFO0016 - Introduction to the theory of computation : Théorie des langages et de la calculabilité.
- INFO0085 - Compilers : fonctionnement des compilateurs.
- INFO0049 - Knowledge representation : Programmation logique (en Prolog).
- INFO0050 - Constraint programming projects : Programmation logiques et sous contraintes.
- INFO0060 - Concurrent system verification and temporal logic : Assistants de preuves.
- ELEN0062/INFO8010 Machine an deep learning : Programmation automatique sur base d'exemples.
- ...

Remerciements

Sources d'inspiration pour cette leçon : Cyril Soldani (présentation aux geeks anonymes), John R. Woodward, Adam Byrtek, Robert Sedgewick/Kevin Wayne.