Algorithmique et Programmation 1 : introduction

1re année Ensimag

https://algo-prog-1.pages.ensimag.fr/web/

Dans le cours d'aujourd'hui

GO AP1 !

- Présentation de la matière
- Introduction au langage GO
- Le nécessaire pour faire ses premiers pas en GO

Petit sondage

- Qui a déjà programmé ?

Quel langage ?

- Python
- C
- Caml
- Java
- ...

Objectifs du cours

Trois grands objectifs

- Apprendre les concepts de la programmation (impérative)
  - notions de base (variables, fonctions...)
  - structures de contrôle (conditionnelles, boucles...)
  - structures de données (tableaux, piles, files, listes...)
  - notions plus « avancées » (invariants, coûts...)
- Apprendre à programmer
  - cours orienté vers la pratique
  - « comprendre en pratiquant »
  - beaucoup de séances machines
  - évaluation sur machine à la fin
- Apprendre des langages de programmation vraiment utilisés
  - majorité des séances en Go
  - courte introduction au C à la fin du semestre

Public attendu

Un cours accessible aux (quasi-)débutants

- Toutes les notions sont présentées sans nécessiter de pré-requis
- Exercices de difficulté croissante en commençant par du facile
- Séances machines encadrées pour guider les premiers pas
- Des séances supplémentaires pour ceux qui en auraient besoin

Un cours intéressant aussi pour les confirmés

- Choix d'un langage récent pour atténuer l'hétérogénéité
  - qui a déjà fait du Go ?
- Il est important de pratiquer pour ne pas perdre la main

Et bien sûr, pour tous ceux entre les deux !

Différents types de séances

- 3 × 2h de Cours-TP (comme aujourd'hui)
  - pour favoriser l'interactivité et la mise en œuvre des concepts
- 2 × 2h de Travaux Dirigés
  - notions « théoriques » (présentation informelle intuitive)
- 18 × 2h de Travaux Pratiques
  - apprentissage par la pratique
  - les TP sont longs, ils doivent être finis en temps-libre
- 1 × 2h d'examen blanc (pas noté)
  - pour s'entraîner et faire le point à mi-semestre

Attention

- 4 ECTS = 48h encadrées + **48h de travail personnel**
- Tout au long du semestre, pas juste avant l'examen !

Évaluation

Examen en fin de semestre

- Examen de 2h en salles machines
- Environnement spécifique
  - réseau coupé entre les machines
  - accès limité à internet (site du cours et documentation)
    - mais pas ChatGPT & Co. !
  - compte vide et pas d'accès au compte informatique personnel
- Une feuille de notes **manuscrite** A4 recto verso
- Contenu : des exercices similaires à ceux traités en TP

Attention

2h ça passe très vite : il faut pratiquer la mise au point en TP !

Permanences d'algorithmique et programmation

Des séances facultatives pour ceux qui en ont besoin

- 32 h de TP supplémentaires tout au long du semestre
- Pas d'inscription, vous venez si et quand vous voulez
- Pas de programme spécifique, on travaille AP1
- Des séances en petit groupe pour faciliter les questions
- Plusieurs créneaux prévus dans l'emploi du temps
  - jeudi de 13h30 à 17h30
  - vendredi de 16h15 à 18h15
    (pas toutes les semaines, vérifiez dans ADE)

$\Rightarrow$ choisissez les créneaux qui collent avec vos autres cours

Conseil

Du temps encadré en plus pour finir vos TP d'AP1
C'est fait pour vous aider, profitez-en !

Équipe pédagogique

Vos enseignants

- G1 : Christophe Rippert
- G2 : Julie Dumas
- G3 : Claire Maiza
- G4 : Manuel Selva
- G5 : Sylvain Bouveret
- G6 : Christophe Rippert
- Permanences : Christophe Rippert

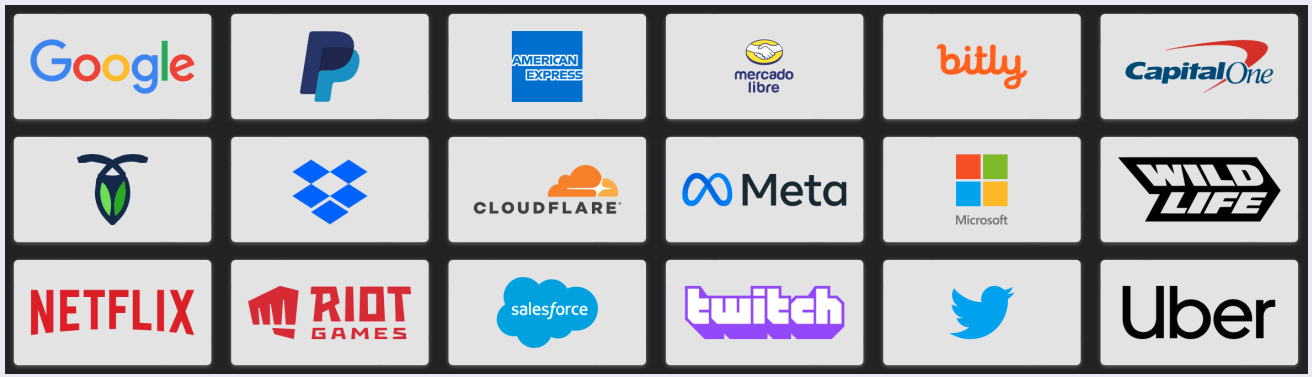
Contact

Contact : `Prenom.Nom@Grenoble-INP.fr` (← pas ~~`.org`~~)

Historique

- Créé en 2009 par Robert Griesemer, Rob Pike, et **Ken Thompson**
- Développé par Google (v1.0 en 2012, v1.25 en 2025)
- Indice TIOBE mars 2025 : 7^{e} langage le plus recherché, en progression régulière

Utilisateurs dans la vraie vie



Un peu d'histoire

Kenneth Lane Thompson, aka « Ken »

1943 naissance à la Nouvelle-Orléans

1966 diplôme d'ingénieur de UC Berkeley

1969 invente le système Unix

- Android, BSD, iOS, Linux, macOS, ...

1970 invente le langage B, précurseur direct du C

1980 invente Belle, 1re machine de niveau Maître aux échecs

1983 obtient le Prix Turing (Prix Nobel d'informatique)

1992 invente le codage UTF-8 (gestion des caractères internationaux) utilisé par 99% des pages web

Pourquoi créer le langage Go ?

« Nous voulions inventer le C du 21e siècle »

Pourquoi ce langage ?

Motivations de Go (d'après ses auteurs)

- Un langage avec un typage statique fort ⇒ fiabilité
  - vs. Python (typage dynamique)
- Un langage compilé ⇒ efficacité
  - vs. Java (machine virtuelle)
- Un langage simple et clair ⇒ facilité de programmation
  - vs. C++ (fréquement considéré comme trop complexe)
- Ces arguments sont aussi (à peu près) valables pour le C

Alors pourquoi ne pas faire du C en AP1 ?

- C (1972) est le langage le plus utilisé de l'histoire de l'info
- Certains aspects sont désuets et découragent les débutants
- Mais on fera un peu de C à la fin d'AP1 et beaucoup au 2e semestre quand même !

Go est un langage impératif

- Un programme est une succession d'instructions (d'ordres) décrivant ce qu'on veut faire

```
ouvrir_la_porte_du_frigo
prendre_un_yaourt
fermer_la_porte_du_frigo
si le_yaourt_est_perime alors
    le_jeter_a_la_poubelle
    va_racheter_des_yaourts
sinon
    retirer_l_opercule
    prendre_une_cuillere
    tant_que le_yaourt_n_est_pas_fini faire
        manger_une_cuillere_de_yaourt
    fin_faire
fin_si
```

Pourquoi des langages de programmation ?

- Un processeur est un circuit capable de comprendre et d'exécuter des instructions
- Une instruction est un ordre élémentaire écrit en « langage machine » (qui est aussi un langage impératif)
- Ces instructions sont trop « bas-niveau » pour écrire des programmes complexes
  - afficher un message à l'écran : des milliers d'instructions
- Ces instructions sont aussi totalement dépendantes du processeur, ce qui rend difficile le portage des programmes

$\Rightarrow$ on a inventé des langages de « haut-niveau » qui facilitent le travail du programmeur

$\Rightarrow$ on a donc besoin d'un outil pour traduire un programme haut-niveau vers le langage compris par la machine

Cet outil s'appelle un compilateur

- C'est donc un programme qui analyse un autre programme
- Il en profite pour vérifier la correction du code
  - correction syntaxique, c'est-à-dire le respect des règles du langage
  - mais pas la correction des algorithmes !
- On peut résumer de façon schématique le cycle de développement d'un logiciel :
  1. le programmeur écrit son programme dans un langage de haut-niveau
  2. il compile le programme pour produire un fichier « exécutable » (ou « binaire »)
  3. il exécute le binaire pour tester le programme
  4. il se rend compte que ça ne marche pas et revient à l'étape 1

Go est un langage procédural

- Le programme peut être découpé en fonctions
  - on appelle parfois procédure une fonction sans résultat
- Une fonction est d'abord définie

```
fonction manger_un_yaourt:
    tant_que le_yaourt_n_est_pas_fini faire
        manger_une_cuillere_de_yaourt
    fin_faire
```

- Pour pouvoir ensuite être appelée

```
...
retirer_l_opercule
prendre_une_cuillere
manger_un_yaourt
...
```

⇒ En définissant une fonction, on invente une nouvelle instruction

Notions de modules et *packages*

- Les langages fournissent des bibliothèques de fonctions
  - pour éviter d'avoir à recoder les actions les plus classiques
- On parle de *package* pour désigner une bibliothèque
  - on doit importer les bibliothèques pour utiliser leurs fonctions dans un programme
- On peut aussi simplement découper son programme en plusieurs fichiers
  - pour faciliter la lecture ou réutiliser certaines parties dans d'autres programmes
- On appelle modules les fichiers constituant un programme
  - en Go, il n'y a rien à faire pour gérer le découpage en fichiers
- Ce vocabulaire est très variable d'un langage à l'autre !

Découpage, compilation et exécution

- Le compilateur Go doit avoir accès à tous les fichiers du programme pour produire l'exécutable
- Compilation monolithique (vs. séparée en C)
  - par exemple, `go build prog.go mod1.go mod2.go`
  - cette commande produit le fichier exécutable `prog`
- Ici, `prog.go` est le *programme principal*
- Chaque programme principal contient une fonction `main`
  - cette fonction est le *point d'entrée du programme*
  - ça veut dire qu'exécuter `./prog` appelle directement `main`

Gestion des *packages* en Go

- Le compilateur va chercher tout seul les *packages* prédéfinis
- On n'écrira pas nos propres *packages* dans ce cours
  - mais on utilisera tout le temps ceux fournis par Go
- pour appeler une fonction fournie par un *package*, on préfixe le nom d'une fonction par le nom du *package*
  - par exemple, `math.Sqrt(x)` pour calculer $\sqrt{x}$
- par défaut, un programme Go appartient au *package* `main`
  - pas besoin de préfixer les fonctions du *package* `main`

Majuscule ou minuscule ?

- Les noms des fonctions des *packages* fournis commencent par une majuscule (fonctions « publiques »)
- Les noms des fonctions de nos programmes commenceront par une minuscule (fonctions « privées » du *package* `main`)

Résumons tout ça avec un exemple. . .

```
package main

import (
    "fmt"
    "math"
)

func afficherRacineDeDeux() {
    fmt.Println(math.Sqrt(2))
}

func main() {
    fmt.Print("La racine carrée de 2 est ")
    afficherRacineDeDeux()
}
```

Les variables

- En plus d'exécuter des instructions regroupées en fonctions, un programme impératif utilise des variables
- Une variable est une zone de stockage dans la mémoire de l'ordinateur, qui sert à mémoriser des valeurs
- Une variable est définie
  - par son nom, qui sert au programmeur à s'y retrouver
  - par son type, qui décrit à la fois sa taille (l'espace occupée dans la mémoire) et sa nature
    - un nombre, par exemple un entier naturel ou relatif
    - un texte, par exemple `"Coucou !"`, ...
- Les valeurs stockées dans les variables évoluent au fur et à mesure de l'évolution du programme

```
x = math.Sqrt(2)
x = x + 2
```

Les variables

- On déclare une variable avec le mot clé `var` en précisant son type (qui inclut la taille en octets de la variable)
  - `var entier int`
  - `var texte string`
- Déclarer une variable veut dire
  - la rendre visible pour s'en servir dans le programme
  - réserver l'espace mémoire pour stocker sa valeur (allocation)
- Le typage est statique et fort
  - c'est le programmeur qui déclare le type quand il écrit le code
  - ce type est définitif, on ne peut pas changer le type en cours de programme (vs. typage dynamique en Python)
  - le typage contraint les opérations applicables à la variable, on ne peut pas stocker un réel (flottant) dans une variable entière

Initialisation des variables

- En Go, les variables sont pré-initialisées avec une « valeur nulle » (*zero value*) lors de leur déclaration
  - utiliser une variable non initialisée est une erreur courante
  - la mémoire d'un ordinateur n'est jamais « vide » (cela n'a pas de sens pour un composant électronique)
  - la mémoire ne contient pas forcément des 0, mais peut-être des valeurs résiduelles
  - la valeur nulle dépend du type
    - `int` : 0
    - `string` : ""
    - ...
- On peut préciser une valeur initiale lors de la déclaration
  - `var entier int = 5`
  - `var texte string = "Coucou !"`

Noms des variables

- Un nom de variable doit commencer par une lettre, suivie d'une ou plusieurs lettres ou chiffres
  - les accents sont autorisés (UTF-8)
  - Go recommande le *camel case*, par exemple `unNomDeVariable`
  - vs. le C qui utilise le *snake case*, par ex. `un_nom_de_variable`
  - en Go, une majuscule comme première lettre a un sens (variable « publique » qu'on n'utilisera pas dans ce cours)

  $\Rightarrow$ on utilisera toujours des noms de variables (et fonctions) commençant par une minuscule
- On **doit** utiliser des noms de variables parlants
  - `var rayonDuCercle int` ou `var prenom string` : glop
  - `var i int` ou `var uneChaine string` : pas glop
  - imaginez que vous relirez votre programme dans un an en ayant oublié ce qu'il est censé faire !

Déclarations courtes

- Au lieu d'écrire `var entier int = 5`, on peut écrire `entier := 5` sans préciser le type
- Le compilateur *infère* (devine) le type en fonction de la valeur initiale affectée
- Ici, il devine que le type est `int`, mais c'est peut-être faux (on verra plus loin comment éviter cela)

Attention

- Ne pas confondre
  - `x = 5` qui affecte la valeur 5 à une variable existante
  - `x := 5` qui déclare une nouvelle variable initialisée à 5
- Et on ne vous a pas encore présenté la comparaison `x == 5`

Notion de constantes

- On appelle constante une valeur (5, `"Coucou !"`)
- Mais aussi par abus de langage une *variable constante*
  - c'est-à-dire une variable dont la valeur ne varie pas
- On déclare une constante avec le mot clé `const`
  - `const monNom string = "Toto"`
  - `const pi = 3.14`
  - mais **jamais** avec l'opérateur `:=`
- En pratique, il n'y a pas beaucoup de différences entre une variable et une constante
  - rendre explicite (pour le lecteur) le fait que la valeur ne changera jamais
  - se protéger contre une erreur, le compilateur indiquera une erreur si on essaie de modifier une constante

Types de variables classiques

- On distingue
  - les types *scalaire* : une seule valeur par variable
  - les types *agrégés* (ou complexes) : plusieurs valeurs dans une variable
  - par exemple un nombre complexe contient deux valeurs (partie réelle et partie imaginaire)
- en Go, on trouve
  - des entiers signés (relatifs) : `int` (valeurs dans $[-2^{31}..2^{31}-1]$)
  - des entiers non signés (naturels) : `uint` ($[0..2^{32}-1]$)
  - des flottants (réels) : `float64` (intervalle très grand)
  - des booléens : `bool` ($\{false, true\}$)
  - des chaînes de caractères : `string`, c'est-à-dire une suite de caractères UTF-8 potentiellement « infinie » (type agrégé)
  - et beaucoup d'autres qu'on verra en TP...

Notion de transtypage (*type casting*)

- On peut forcer le compilateur à changer le type d'une valeur
  - par exemple pour guider l'inférence : `x := uint(5)`
  - ou pour contourner le typage fort de Go

```
var rel int = -5
var nat uint = 5
res := rel + nat // « mismatched types int and uint »
res := rel + int(nat) // ça c'est bon
```

Mais pourtant en C / Python / . . . ça marche !

Le système de typage de Go est **vraiment** strict !

Au passage, les commentaires

- On peut ajouter des commentaires dans un programme
  - c'est même très fortement recommandé !
- Deux types de commentaires en Go

```
// Un commentaire d'une ligne
/*
  Un commentaire
  sur plusieurs lignes
*/
```

Mais c'est quoi un bon commentaire ?

- Un bon commentaire met l'accent sur un point important

```
// Je divise y par z => inutile, je sais lire !
x := y / z
// on est sûr que z n'est jamais nul ici => important
x := y / z
```

Le type booléen

- Deux valeurs possibles `false` et `true`
- Des opérateurs spécifiques
  - `a && b` réalise un « et-logique » vrai ssi `a` et `b` sont `true`
    - cela veut dire forcément les deux
  - `a || b` réalise un « ou-logique » vrai ssi `a` ou `b` est `true`
    - cela veut dire au moins un des deux, mais aussi les deux
  - `!a` réalise une négation logique vrai ssi `a` est `false`

Attention à ne pas confondre

- Les opérateurs binaires (bit à bit) `&`, `|` et `^` font autre chose

Comparaisons

- Une comparaison a pour résultat un booléen
  - `a < 5` ou `b >= 5` ou `c == 5` ou `d != 5`

Structure de contrôle `if-then-else`

- En informatique, on a souvent besoin de prendre des décisions
  - si `x < 0` alors je fais quelque-chose sinon je fais autre-chose

```
if x < 0 {
    // Je fais quelque-chose
} else {
    // Je fais autre-chose
}
```

- Les accolades définissent la notion de sous-bloc de code
- Les parties du `if` sont exclusives, et pas limités à deux

```
if x < 0 {
    // Je fais quelque-chose
} else if x == 0 {
    // Je fais autre-chose
} else {
    // Je fais une autre autre-chose
}
```

`if` avec initialisation intégrée

- On peut écrire aussi

```
if x := lireEntierAuClavier(); x < 0 {
    // Je fais quelque-chose
} else {
    // Je fais autre-chose
}
```

- La variable `x` n'est *visible* (n'existe) qu'à l'intérieur du `if`

```
if x := lireEntierAuClavier(); x < 0 {
    // Je fais quelque-chose
} else {
    // Je fais autre-chose
}
x = x + 1 // Erreur, x n'existe pas
```

$\Rightarrow$ notion de visibilité d'une variable

Sous-blocs et visibilité des variables

- La notion de visibilité est générale en Go

```
x := 5 // x vaut 5
{
    x := 10 // un AUTRE x vaut 10
}
// le 1er x vaut toujours 5
```

- Le deuxième `x` *masque* le premier
- Ce sont **deux variables différentes** qui s'appellent pareil
- En général, ce n'est pas une bonne idée...

$\Rightarrow$ trouver un autre nom !

Attention c'est presque pareil !

```
x := 5 // x vaut 5
{
    x = 10 // on change la valeur de x
}
// x vaut bien 10
```

- On n'a pas alloué une nouvelle variable avec `x = 10`, juste changé la valeur de la variable existante
- Donc, avec un `if`, pour revenir à nos moutons

```
x := 5 // x vaut 5
if uneCondition {
    // cas 1: x = 10 => on change la valeur de x
    // cas 2: x := 10 => on alloue un autre x limité au if
}
// cas 1 : x vaut 10
// cas 2 : x vaut 5
```

`if` avec initialisation intégrée

- Résumons, pour voir si on a bien compris

```
x := 5 // x vaut 5
if x := 10; x > 5 {
    // on rentre bien dans le if !
    // x > 5 concerne le x défini dans le if
}
// x vaut 5
```

- Est donc équivalent à

```
x := 5
{
    x := 10
    if x > 5 {
        ...
    }
}
```

Affectations multiples (à utiliser avec modération)

- En Go, on peut écrire `x, y = 3, 4` (ou `x, y := 3, 4`)
- On peut aussi écrire `x, y = 1, x...`
  - la partie droite est évaluée complètement avant d'affecter
- Tiens, et si on écrit ça ?

```
x := 1
x, y := 5, 6 // x vaut 5 et y vaut 6, et ça ne crie pas !
```

⇒ c'est bien **le même** `x` dont la valeur est changée

- c'est équivalent à `x = 5` puis `y := 6`
- Ça par contre, ça fait crier le compilateur :

```
x, y := 1, 2
x, y := 5, 6 // no new variables on left side of :=
```

⇒ il faut **au moins** une nouvelle variable en partie gauche du `:=`

Fonctions

- Un programme Go doit contenir une fonction `main` qui est le *point d'entrée du programme*

```
func main() { // prototype imposé
    x := somme(1, 2)
}
```

- Cette fonction ne prend pas de paramètre et ne renvoie rien (procédure)
- Souvent les fonctions prennent des paramètres et renvoie quelque-chose

```
func somme(a, b int) int { // tout à l'envers du C...
    c := a + b
    return c
}
```

Le mot clé `return`

- Arrête immédiatement l'exécution de la fonction
- Renvoie éventuellement une ou des valeurs

```
func traiterX(x int) {
    if x < 0 {
        return
    }
    // on fait ce qu'on doit faire avec x
    // il y a un return implicite à la fin de la fonction
}

func inverser(x, y int) (int, int) {
    return y, x
}
```

Fonctions renvoyant plusieurs valeurs

```
func xZero(x int, message string) (string, bool) {
    if x == 0 {
        return "J'ai dis pas 0 !", false
    }
    return message, true
}
```

- On récupère les 2 résultats `txt, ok := xZero(5, "OK")`
- On peut aussi ignorer les deux `xZero(5, "Yes")`
- Mais on ne peut pas ignorer un des deux et pas l'autre...

La variable prédéfinie anonyme _ sert à ça

- `_, ok := xZero(5, "A")` ou `txt, _ := xZero(5, "B")`
- Ce n'est pas une « vraie » variable `x := _ + 1`

  $\Rightarrow$ « cannot use _ as value or type »

Visibilités des variables et fonctions

```
var y int = 5 // variable globale au programme
// Note : y := 5 n'est pas possible pour des var globales
func somme(a, b int) int {
    c := a + b + y
    return c
}
func main() {
    x := somme(1, 2) + y
}
```

- Les accolades des fonctions définissent des sous-blocs
- `y` est visible dans toutes les fonctions du programme
- `c` n'est visible que dans `somme` (idem `x` pour `main`)
- `a` et `b` ne sont visibles que dans `somme`

$\Rightarrow$ paramètres d'une fonction $\equiv$ variables locales

Conclusion

Beaucoup de notions d'un coup aujourd'hui

- C'est nécessaire pour commencer à programmer
- Si vous n'avez pas tout compris ou retenu, ce n'est pas grave !
  - n'hésitez pas à revenir plus tard sur ce cours si vous êtes bloqués dans un exercice
- On va apprendre en pratiquant tout au long du semestre
- D'ailleurs, on commence tout de suite avec les premiers TP !

Tous les supports sont sur le web

https://algo-prog-1.pages.ensimag.fr/web/