Algorithmique et Programmation 1 : les boucles

1re année Ensimag

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

On veut afficher les chiffres de 1 et 5 dans le terminal

```
func main() {
    fmt.Println(1)
    fmt.Println(2)
    fmt.Println(3)
    fmt.Println(4)
    fmt.Println(5)
}
```

Code répétitif à coup de copié-collé

- Difficile à maintenir : changer l'affichage $\Rightarrow$ 5 lignes à changer
- Plus de lignes de code $\Rightarrow$ plus de risques de faire des erreurs

Modèles de boucles classiques

while : tant_que

```
tant_que une_condition_booléenne est vraie faire
    quelque-chose
fin_faire
```

for : pour_chaque

```
pour_chaque valeur dans un_intervalle faire
    quelque-chose
fin_faire
```

- *foreach* proposé dans de nombreux langages modernes
  - en Python : `for val in tab:`
  - mais pas en C par exemple

Les boucles en Go

En Go

- Pas de `while` du tout
- Un `for` « à la C » qui est un `while` déguisé
- Un *foreach* qu'on verra plus tard (`for` et `range`)

Schéma général d'un `for` en Go

```
for idx := 1; idx <= 5; idx++ {
    fmt.Println(idx)
}
```

- Les points-virgules séparent les trois parties du `for`
- Les accolades définissent un sous-bloc

Schéma général d'un `for` en Go

```
for idx := 1; idx <= 5; idx++ {
    fmt.Println(idx)
}
```

`idx := 1`

- Initialisation de l'indice à la borne inférieure de l'intervalle
  - en pratique, on peut mettre n'importe quelle instruction
- Instruction exécutée une fois avant d'entrer dans la boucle
- Variable `idx` *locale* à la boucle (sous-bloc)

Les boucles en Go

Schéma général d'un `for` en Go

```
for idx := 1; idx <= 5; idx++ {
    fmt.Println(idx)
}
```

`idx <= 5`

- Condition de continuation de la boucle (i.e. booléen)
  - en pratique, peut être indépendante de l'indice
- Évaluée à chaque tour de boucle, avant d'exécuter le corps
  - condition fausse dès le début $\Rightarrow$ on n'entre pas dans la boucle

Schéma général d'un `for` en Go

```
for idx := 1; idx <= 5; idx++ {
    fmt.Println(idx)
}
```

`idx++`

- Rappel : `idx++` est un raccourci pour `idx = idx + 1`
- Instruction exécutée à chaque tour de boucle, après le corps
  - en pratique, on peut écrire n'importe-quelle instruction

Rappel : notion de sous-bloc

En Go, on peut écrire

```
idx := 1                  // variable externe au sous-bloc
{                         // début du sous-bloc
    idx := 2              // variable locale au sous-bloc
    fmt.Println(idx) // affiche 2
}                         // fin du sous-bloc
fmt.Println(idx)      // affiche 1
```

- Le `idx` local au sous-bloc **masque** le `idx` externe
- En pratique, ce sont **deux variables différentes**

Masquage de variable

- C'est en général une mauvaise pratique...
- On peut toujours trouver un autre nom !

Rappel : notion de sous-bloc

Attention c'est presque pareil !

```
idx := 1                    // variable externe au sous-bloc
{                           // début du sous-bloc
    idx = 2                 // c'est la variable externe !
    fmt.Println(idx) // affiche 2
}                           // fin du sous-bloc
fmt.Println(idx)            // affiche 2
```

Rappel

- `idx := 2` déclare une nouvelle variable et l'initialise
- `idx = 2` change la valeur d'une variable pré-existante

Dépliage d'une boucle

```
for idx := 1; idx <= 5; idx++ {
    fmt.Println(idx)
}
```

```
{
    idx := 1            // idx est initialisé à 1
    // idx <= 5 ? oui : on rentre dans la boucle
    fmt.Println(idx) // affiche 1
    idx++               // idx vaut 2
    // idx <= 5 ? oui : on continue
    ...
    idx++               // idx vaut 6
    // idx <= 5 ? non, on sort de la boucle
}
```

- On sort de la boucle quand `idx` va « un coup trop loin »
- Ça sera important notamment quand on utilisera des tableaux

Chaque partie du `for` est facultative

On peut écrire

```
idx := 1
for ; idx <= 5; idx++ {
    fmt.Println(idx)
}
```

Syntaxe

- Le point-virgule est nécessaire
- Dans ce cas, `idx` **n'est pas** local à la boucle

Chaque partie du `for` est facultative

On peut aussi écrire

```
idx := 1
for ; ; idx ++ {
    if idx > 5 {
        break
    }
    fmt.Println(idx)
}
```

`break` ?

- Provoque la sortie immédiate de la boucle **englobante** (cf. boucles imbriquées)
- Noter qu'on inverse la condition (de sortie ici *vs.* de continuation avant)

Chaque partie du `for` est facultative

On peut même écrire

```
idx := 1
for {
    if idx > 5 {
        break
    }
    fmt.Println(idx)
    idx++
}
```

Syntaxe

- Là, on ne met pas les deux points-virgules
- `for` sans rien derrière est en fait une boucle infinie

Boucle `while`

N'existe pas en Go, mais s'écrit facilement

```
idx := 1
for idx <= 5 {
    fmt.Println(idx)
    idx++
}
```

Syntaxe

- Là non plus, on ne met pas les deux points-virgules

`while` est plus général que *foreach*

- Pas d'énumération séquentielle d'un intervalle

```
for lireEntier() != valeurSecrète {
    fmt.Println("Pas trouvé, essayez encore !")
}
```

Un autre mot-clé utile

- `continue` redémarre immédiatement le corps de la boucle
- Peut être utile pour éviter un `else`
- À utiliser avec modération, ça complique la compréhension du flot d'exécution

Détecteur de nombres pairs

```
for idx := 1; idx < 7; idx++ {
    fmt.Println("Je tente le", idx)
    if idx%2 == 1 {
        continue
    }
    fmt.Println("=>", idx, "est un nombre pair !")
}
```

Boucles imbriquées

Même principe

```
for lig := 0; lig < 3; lig++ {
    for col := 0; col < 3; col++ {
        fmt.Print(lig*3 + col)
    }
    fmt.Println()
}
```

Affichage

```
012
345
678
```

Boucles imbriquées

break ne sort que d'un niveau de boucle

```
for lig := 0; lig < 3; lig++ {
    for col := 0; ; col++ {
        if col >= 3 {
            break
        }
        fmt.Print(lig*3 + col)
    }
    fmt.Println()
}
```

Attention

- Ne pas abuser des `break` (complique le flot d'exécution)
- Boucle équivalente avec un booléen `fini` dans la condition de continuation

Notions de coûts

Théorie de la complexité d'un algorithme, selon différents critères

- Coût en temps d'exécution
  - on parle d'efficacité de l'algorithme
  - et on a envie que ça aille le plus vite possible !
- Coût en espace mémoire
  - on parle d'empreinte mémoire de l'algorithme
  - et on a envie qu'elle soit la plus petite possible
- Coût en énergie
  - souvent lié à l'efficacité de l'algorithme, mais pas uniquement
  - estimation : ChatGPT-3 consommerait autant qu'environ 100 foyers (américains) par an
- Coût en nombre de transistors, ...

Dans le cours d'AP1

- On reste au niveau de l'intuition informelle
  - calculs mathématiques précis en AP2
- Algorithmes raisonnablement efficaces
  - pas les meilleurs algos, mais qui restent simples
- Idem pour l'empreinte mémoire
  - on essaie autant que possible de travailler *en place*
  - ex. : on trie un tableau en échangeant ses éléments, sans allouer un nouveau tableau
- Pas d'analyse du coût en énergie
  - c'est beaucoup plus compliqué, il faut déjà apprendre à programmer

Efficacité d'un algorithme

- On utilisera la notation $O(n)$ (« grand o de n »)
- Comparaison asymptotique
  - étude de la vitesse de croissance d'une fonction
  - en fonction d'un paramètre : la taille (le nombre) des données
  - ex. : la taille d'un tableau, le nombre d'éléments d'une liste, ...
- Intuitivement : comment se comporte mon algorithme quand n devient très grand ?

Différentes sortes de coûts en temps d'exécution

- Coût dans le meilleur cas ou dans le pire cas
  - cas particuliers exceptionnels
- Coût moyen
  - cas général $\Rightarrow$ c'est de ça qu'on parle ici

Complexités classiques

- coût constant : $O(1)$
  - accès à une variable, à une case d'un tableau
- logarithmique : $O(log(n))$
  - recherche dichotomique
- linéaire : $O(n)$
  - parcours d'un tableau, d'une liste
- linéarithmique : $O(n \times log(n))$
  - tris efficaces en AP2
- quadratique : $O(n^2)$
  - tris simples qu'on verra ce semestre

De façon très intuitive

- Coût linéaire

```
for i := 0; i < n; i++ {
    fmt.Println(i)
}
```

- Coût quadratique

```
for i := 0; i < n; i++ {
    for j := 0; j < n; j++ {
        fmt.Println(i, j)
    }
}
```

- Mais beaucoup d'exceptions
  - ex. tri du nain de jardin en TP : une seule boucle, mais $O(n^2)$

Fonctions de la bibliothèque

- Totalement dépendant de l'implantation et du langage
  - en Go, `len(chaine)` a un coût constant
    ```
    for i := 0; i < len(chaine); i++ {} // OK
    ```
  - en C, `strlen(chaine)` a un coût linéaire
    ```
    for (int i = 0; i < strlen(chaine); i++) {} // NOK
    ```
    - fonction à coût linéaire dans une boucle $\Rightarrow$ coût quadratique
  - il vaut mieux écrire
    ```
    int taille = strlen(chaine);
    for (int i = 0; i < taille; i++) {} // OK
    ```

Le coût est en général indiqué dans la documentation

Il faut donc la lire attentivement !