Algorithmique et Programmation 1 : les tableaux

1re année Ensimag

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

Définition

- Un tableau est une structure de données
  - contenant des valeurs appelées éléments
  - qui sont toutes du même type (type de base du tableau)
  - et qui sont accessibles via un indice (index)
  - typiquement via une syntaxe du style `tab[idx]`
- L'accès à chaque élément se fait en temps constant
  - il est aussi coûteux d'accéder à `tab[1]` qu'à `tab[1_000_000]`
- Les éléments sont stockés de façon contiguë en mémoire
  - à la différence des listes chaînées qu'on verra plus tard

Valeur	**45**	**154**	**58**	**78**	**31**	**5**	**74**
Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**

Deux sortes de tableaux

- Les tableaux statiques (ex. en C)
  - statique veut dire de taille (nombre d'éléments) constante
  - les tableaux « historiques », qui existent en Go
  - dans certains langages (comme Go), la taille fait partie du type

  $\Rightarrow$ `var tab [3]int` et `var tab [4]int` ont des types différents (pour ceux qui ont fait du C, oui les crochets sont avant le type de base)
- Les tableaux dynamiques (ex. en Python)
  - la taille peut changer au cours de l'exécution du programme
  - on les trouve dans des langages plus « modernes »
  - en Go, ils s'appellent *slices*
  - on peut s'en servir comme des tableaux statiques

  $\Rightarrow$ c'est ce qu'on fera dans les prochains TP

Syntaxe en Go

```
// un tableau de trois entiers signés :
// - initialisé à 0, 0, 0
// - d'indices 0, 1 et 2
var tab [3]int
fmt.Println(len(tab)) // affiche "3"
fmt.Println(tab)      // affiche "[0 0 0]"
// on change chaque valeur individuellement
tab[1], tab[2] = 1, 2
fmt.Println(tab)      // affiche [0 1 2]
```

Erreur très fréquente

```
tab[3] = 3
// invalid argument: index 3 out of bounds [0:3]
```

$\Rightarrow$ oui le `:3]` n'est pas très clair, mais Go ne parle pas en math...

Utilisations typiques des tableaux

- Et de beaucoup d'autres structures séquentielles
  - c'est-à-dire pouvant être parcourues un élément après l'autre, dans un certain ordre
- Beaucoup d'algorithmes se ramènent à deux schémas classiques : parcours et recherche
  - un parcours applique une opération à chaque élément
  - une recherche teste la présence d'un élément validant une propriété
  - un parcours va donc parcourir toute la structure (chaque élément)...
  - alors qu'une recherche s'arrêtera dès qu'on trouve un élément validant la propriété...
  - mais parcourra potentiellement toute la structure si aucun élément ne valide la propriété

Deux catégories de parcours

- Les parcours de type 1 sont les plus courants

Parcours de type 1

En pseudo-code :

```
initialisation
tant que non fini
    traiter l'élément courant
    avancer
finalisation
```

Parcours de type 1

```
for idx := 0; idx < len(tab); idx++ {
    fmt.Print(tab[idx], " ")
}
fmt.Println()
```

- `idx := 0` : initialisation
- `idx < len(tab)` : non fini
- `fmt.Print(tab[idx], " ")` : traiter l'élément courant
- `idx++` : avancer
- `fmt.Println()` : finalisation

Parcours de type 2

```
initialisation
tant que non fini
    avancer // on avance avant de traiter l'élément
    traiter l'élément courant
finalisation
```

- Au début, on n'est pas encore « dans » la structure

```
// initialisation
lecteur := bufio.NewScanner(os.Stdin)
// non fini et avancer en une seule fonction
for lecteur.Scan() {
    // traiter l'élément courant
    fmt.Println(lecteur.Text())
}
```

Recherche

```
initialisation
tant que non fini ET PUIS non trouvé
    avancer
finalisation
```

- ET PUIS est une évaluation « paresseuse »
- faux ET x est toujours faux
- vrai OU y est toujours vrai

⇒ on n'évalue pas le deuxième opérande dans ces cas-là

- `&&` et `||` en Go sont paresseux (en C aussi)
- En pratique, très peu de langages non-paresseux
  - en Ada, on trouve **`and`** et **`and then`** (et **`or`** et **`or else`**)

Recherche

```
1  func premierEntierPair() {
2      var idx int = 0
3      for ; idx < len(tab) && tab[idx]%2 == 1; idx++ {
4      }
5      if idx < len(tab) {
6          fmt.Println("Entier pair à l'indice", idx)
7      } // sinon on n'affiche rien
8  }
```

Questions

1. Ligne 2 : pourquoi pas dans le `for` ?
2. Ligne 5 : pourquoi pas `tab[idx]%2 == 0` ?
3. Et donc, pourquoi un `&&` paresseux est indispensable ?

Invariants et variants de boucle

Contexte

- Des propriétés logiques permettant de « garantir » la correction d'un algorithme
- En AP1, pas de définitions formelles, mais une compréhension intuitive
- Prouver qu'un algorithme est correct est un problème complexe et souvent ouvert

Définition

- Un invariant de boucle est une propriété logique (booléenne)
  - vraie avant le début de la boucle
  - préservée pendant l'exécution de la boucle
  - donc toujours vraie en sortie de la boucle

Exemple

- On recherche l'indice de l'entier maximal dans un tableau contenant au moins un élément

```
func rechercheIndiceDuMax() int {
    var idxMax int = 0
    // Prop. établie : idxMax == indice de l'élément max
    for idx := 1; idx < len(tab); idx++ {
        // Préservée : les 3 parties du for ne changent
            pas idxMax
        if tab[idx] > tab[idxMax] {
            idxMax = idx
        }
        // Préservée : idxMax est bien mis à jour
    }
    // => La propriété est vraie en sortie de boucle
    return idxMax
}
```

Définition

- Un variant de boucle est une variable dont l'évolution « garantit » la terminaison de la boucle

```
func rechercherIndiceDuMax() int {
    // Au fait, est-ce vraiment une recherche ?
    var idxMax int = 0
    // idx est le variant de la boucle :
    // - il part de 1
    // - il est incrémenté de 1 à chaque tour
    // => il atteindra len(tab) et la boucle se terminera
    for idx := 1; idx < len(tab); idx++ {
        if tab[idx] > tab[idxMax] {
            idxMax = idx
        }
    }
    return idxMax
}
```

Erreur classique

- On aurait pu écrire aussi `idx != len(tab)`
- Mais attention :

```
func afficherUnSurDeux() {
    // Au fait, on peut initialiser un tableau comme ça
    var tab = [5]int{2, 1, 7, 3, 1}
    for idx := 0; idx != len(tab); idx += 2 {
        fmt.Println(tab[idx])
    }
}
```

- Affiche « 2 7 1 panic : runtime error : index out of range [6] with length 5 »
- Il est plus sûr d'utiliser < $\Rightarrow$ programmation défensive

En pratique

- On ne cherchera pas à prouver des invariants ou variants
- On réfléchira avec des dessins
  - état initial du tableau (avant de commencer la boucle)
  - état final du tableau (le résultat attendu en fin de boucle)
  - état intermédiaire (passage de l'étape I à I+1)

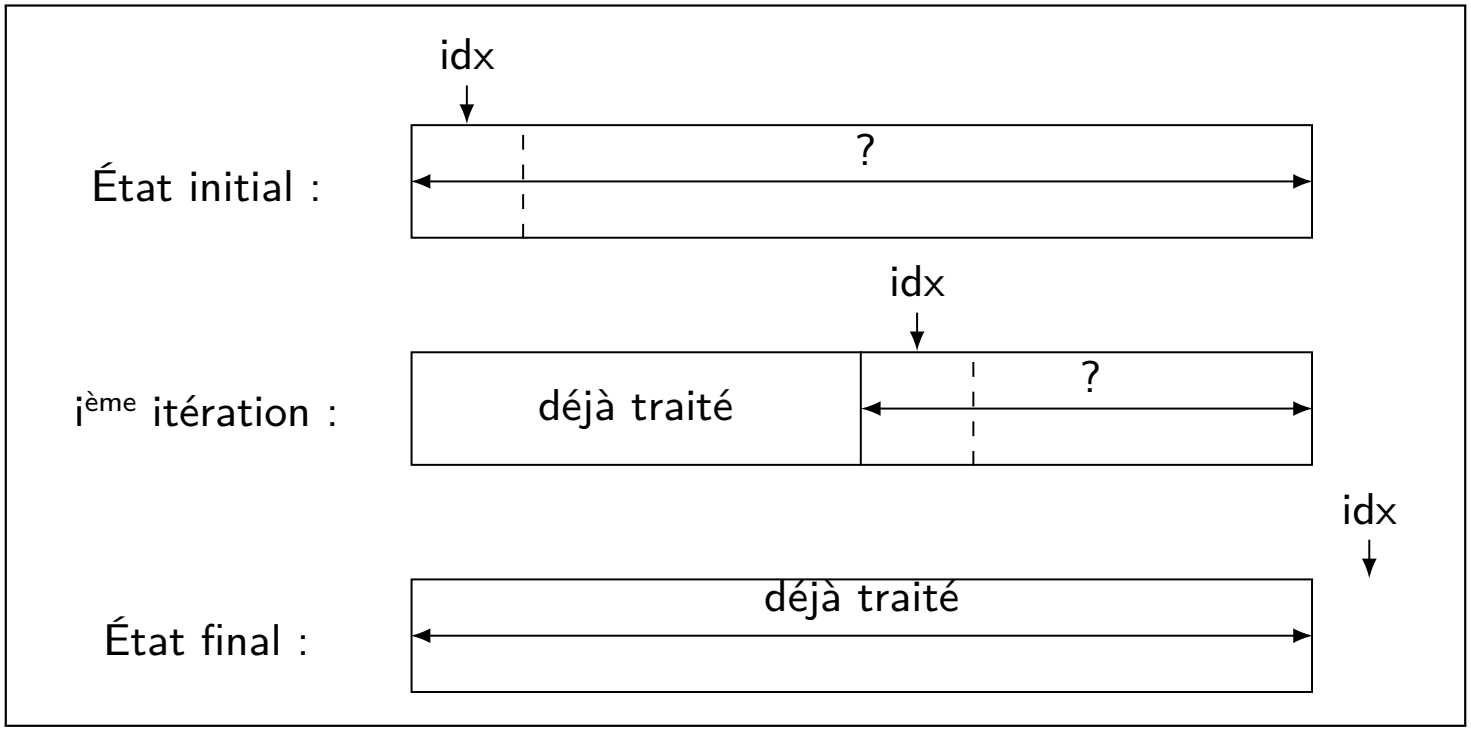
Exemple

- On veut ajouter 1 à chaque valeur d'un tableau d'entiers
- On va sûrement avoir un indice `idx` pour parcourir le tableau case par case

Analyse (dessin diapo suivante)

- État initial
  - `idx = 0`, il pointe sur la première case non traitée, c'est-à-dire la première case du tableau
  - aucun élément n'est traité
- État final
  - `idx = len(tab)`, c'est-à-dire en dehors du tableau
  - tous les éléments sont traités
- État intermédiaire
  - `idx` pointe sur la première case non traitée
  - on passe de l à $l+1$ en traitant `tab[idx]` et en avançant à `idx = idx + 1`

Invariants et variants de boucle




```
func ajouterUn() {
    for idx := 0; idx < len(tab); idx++ {
        tab[idx]++
    }
}
```

Spécification

- On choisit une case du tableau (par exemple la première)
- On considère que la valeur de cette case est le pivot
- On veut réorganiser les éléments du tableau selon la propriété suivante
  - tous les éléments « à gauche » du pivot (c'est-à-dire tous les éléments d'indices strictement inférieurs à l'indice du pivot) sont inférieurs ou égaux au pivot
  - tous les éléments « à droite » du pivot sont strictement supérieurs au pivot
- On travaille par échange d'éléments

  `tab[idx1], tab[idx2] = tab[idx2], tab[idx1]`
- À la fin, les deux parties ne sont pas forcément triées
- Cet algorithme est la base d'un tri rapide en $O(n \times log(n))$ appelé étonnamment le *quicksort*

Algorithme du pivot

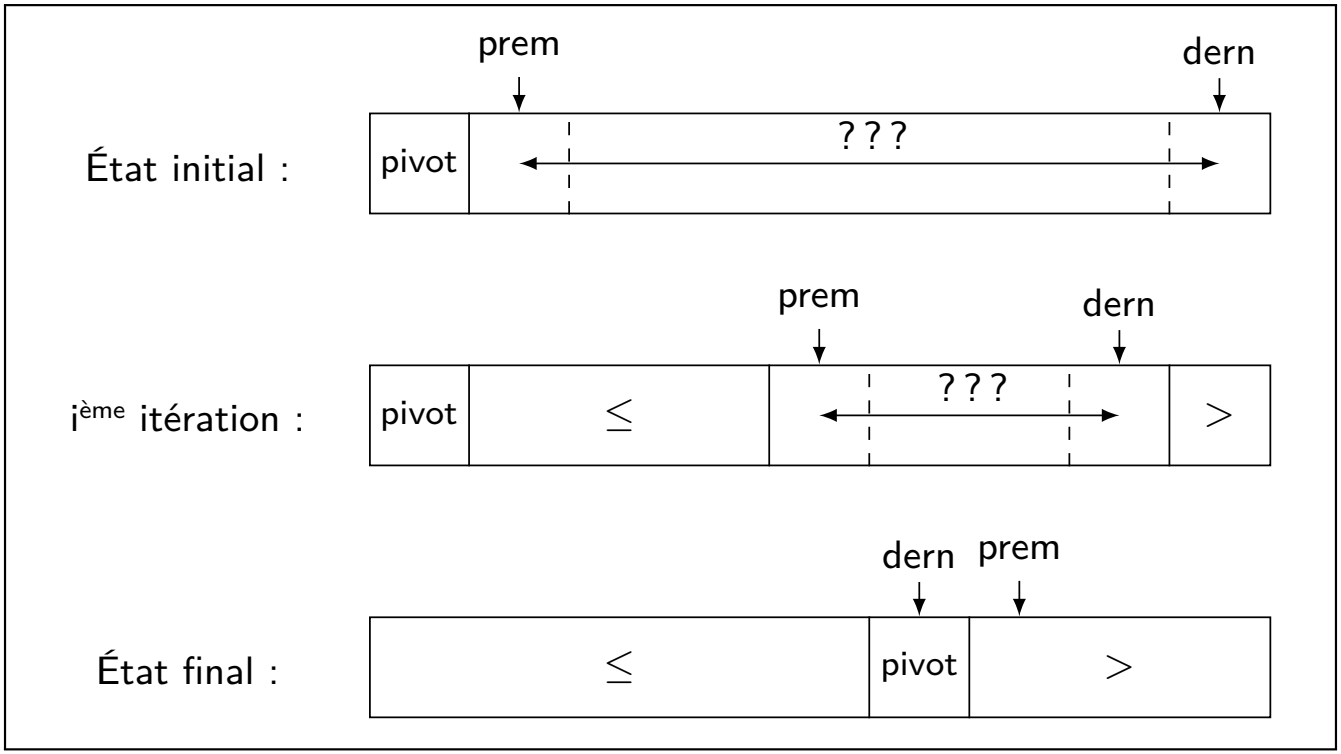
Exercice (mêmes questions pour les exercices suivants)

- Dessinez les états initial, final et intermédiaire
- Intuitez comment passer de I à I+1
- On implantera le code en TP

Indices pour le pivot

- On peut utiliser deux indices `prem` et `dern` pour délimiter la zone des éléments non traités dans le tableau
- Cette zone va rétrécir à chaque tour de boucle, soit en avançant `prem` d'une case vers la droite, soit en reculant `dern` d'une case vers la gauche
- On s'arrêtera quand `prem` et `dern` se croiseront

Algorithme du pivot



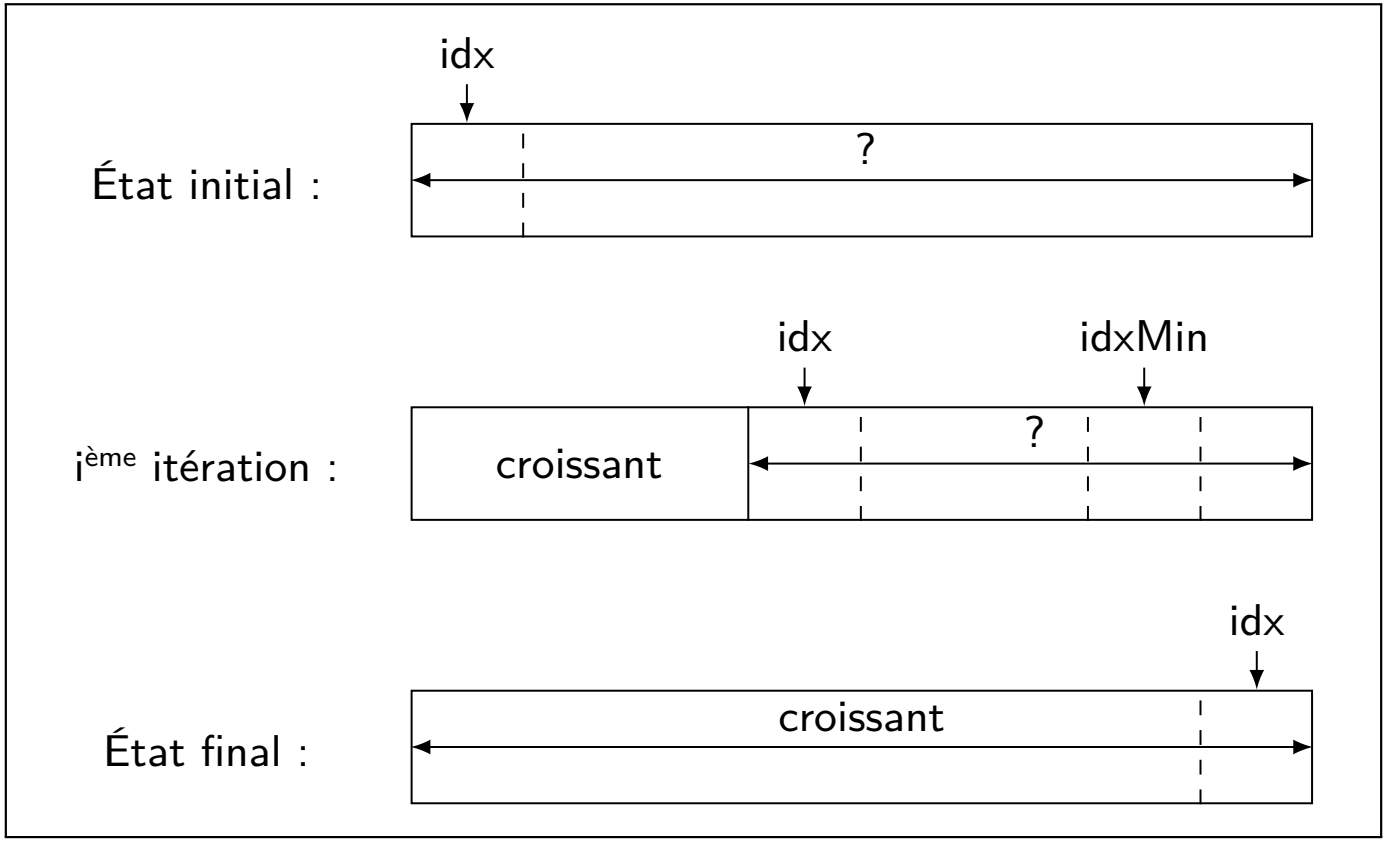
- I $\Rightarrow$ I+1 : on va comparer `tab[prem]` avec le pivot `tab[0]`
  - si `tab[prem] <= tab[0]`, il suffit d'avancer `prem++`
  - sinon, il faut échanger `tab[prem]` et `tab[dern]` et reculer `dern` d'une case vers la gauche
- État final : il faut penser à échanger `tab[0]` et `tab[dern]` pour placer le pivot comme frontière des deux zones

Tri par sélection du minimum

Spécification

- On veut trier les éléments du tableau par ordre croissant
- On cherche l'élément minimum dans le tableau et on le place (échange) dans la première case du tableau
- On cherche l'élément minimum dans la fin du tableau et on le place dans la deuxième case du tableau
  - ici fin du tableau veut dire tout le tableau sauf la première case
- On cherche l'élément minimum dans la fin du tableau et on le place dans la troisième case du tableau
  - ici fin du tableau veut dire tout le tableau sauf les deux premières cases
- ...
- On peut s'arrêter quand la fin du tableau ne contient plus qu'une case, c'est forcément le maximum

Tri par sélection du minimum



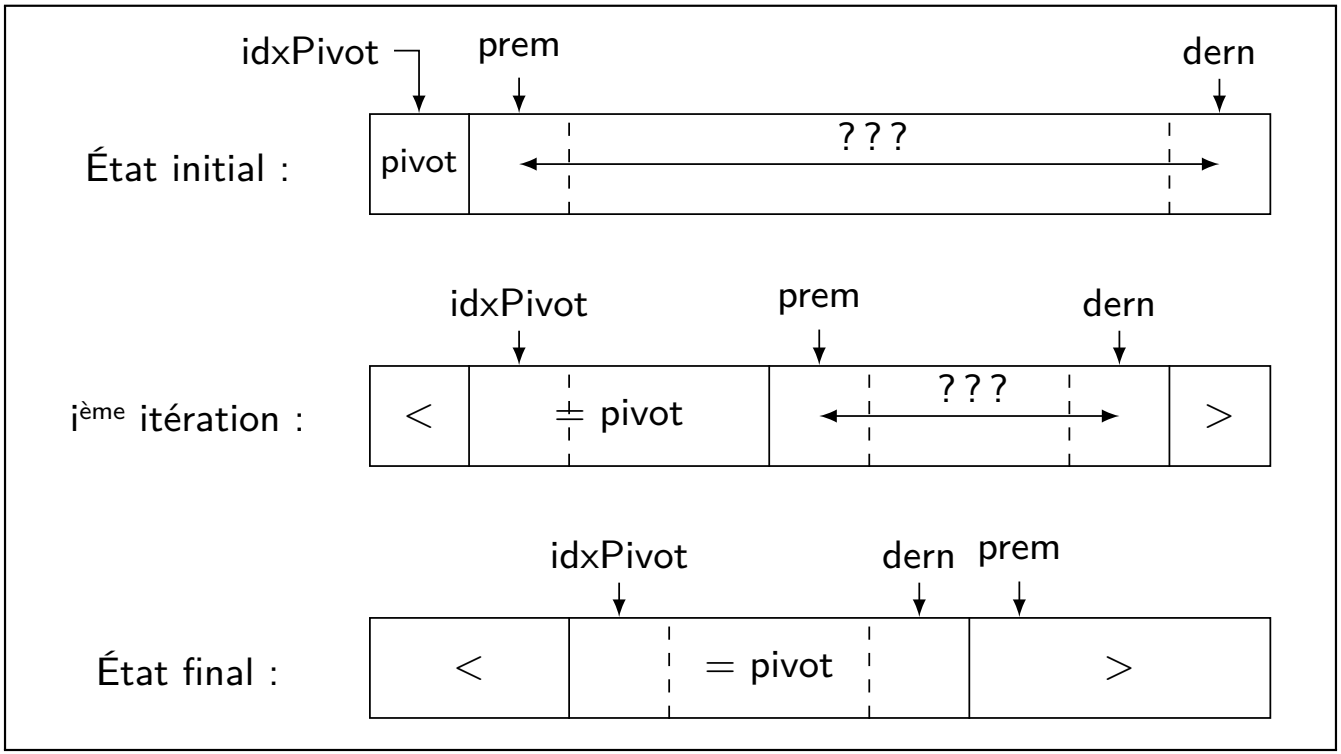
- Invariant : les éléments de la partie gauche triée par ordre croissant sont tous $\leq$ à ceux de la partie droite non traitée
- I $\Rightarrow$ I+1 : on échange `tab[idx]` et `tab[idxMin]` puis on incrémente `idx`

Algorithme du drapeau

Spécification

- Comme le pivot, mais on veut trois zones à la fin
  - la zone des inférieurs stricts au pivot, celle des égaux au pivot et celle des supérieurs stricts au pivot
- Une variante du problème du drapeau hollandais (formulé par Edsger Dijkstra), qui consiste à regrouper des éléments de trois couleurs en zones homogènes
- Indice : il faudra sûrement trois indices `prem`, `dern` et `idxPivot` qui va bouger

Algorithme du drapeau



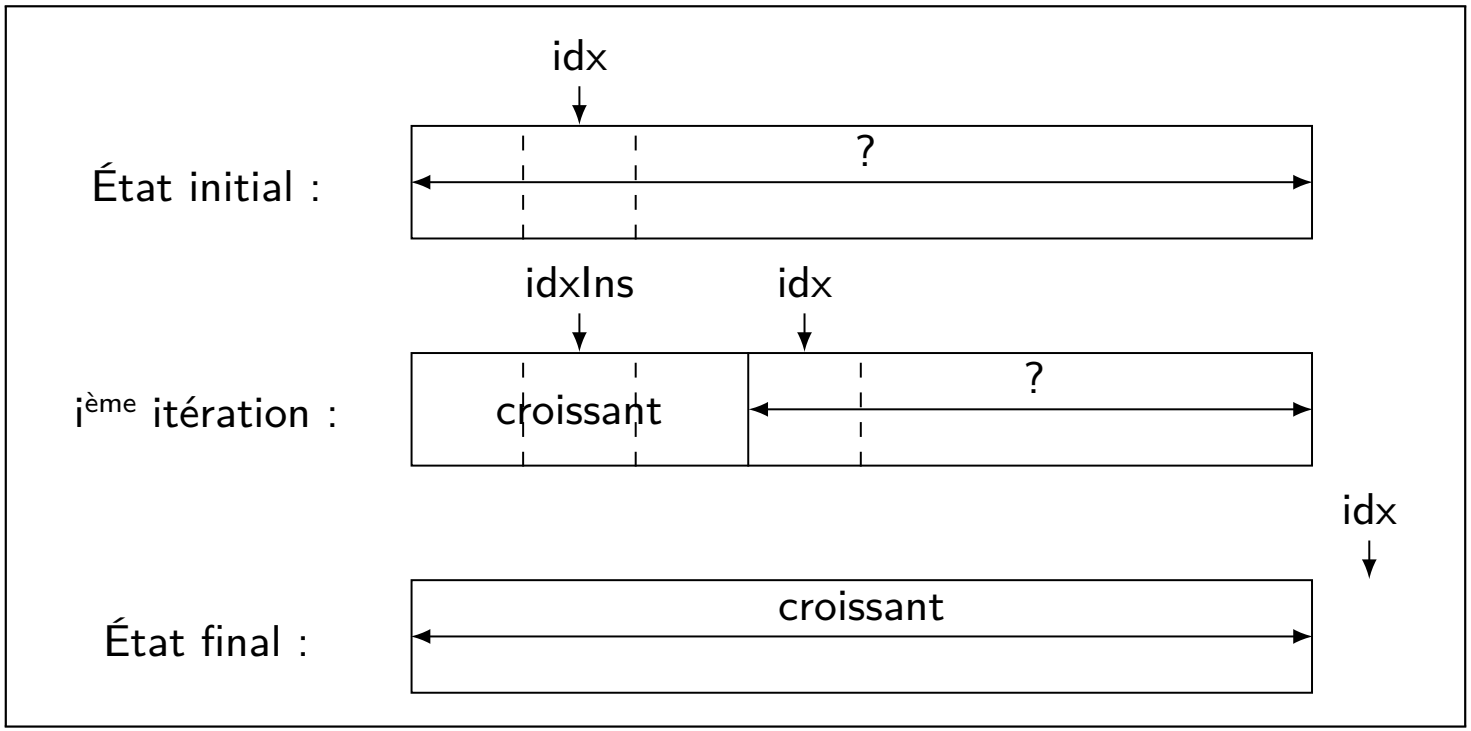
- I $\Rightarrow$ I+1 : on va comparer `tab[prem]` avec la valeur pivot
  - si `tab[prem] == tab[idxPivot]`, il suffit d'avancer `prem++`
  - si `tab[prem] < tab[idxPivot]`, il faut échanger `tab[prem]` et `tab[idxPivot]`, et incrémenter `prem` et `idxPivot`
  - sinon, il faut échanger `tab[prem]` et `tab[dern]` et reculer `dern` d'une case vers la gauche comme pour le pivot

Tri par insertion

Spécification

- On veut trier les éléments du tableau par ordre croissant
- Le début du tableau est trié et la fin pas encore
  - le début du tableau est entre les indices 0 et `idx-1` inclus
  - la fin est le reste du tableau
- On remonte (dans le sens des indices décroissants) le début du tableau pour chercher l'endroit où insérer `tab[idx]`
  - au passage, on décale chaque élément d'une case vers la droite
- On insère `tab[idx]` à sa place dans le début du tableau, et on incrémente `idx`
- `idx` peut donc partir directement de la deuxième case
- C'est un peu le principe réciproque du tri précédent
  - sélection : on cherche quel élément placer à l'indice `idx`
  - insertion : on cherche à quel indice placer `tab[idx]`

Tri par insertion



- I $\Rightarrow$ I+1 : on cherche l'endroit où insérer `tab[idx]` tout en décalant les éléments, puis on incrémente `idx`
- On ne peut pas s'arrêter avant la fin, car `tab[len(tab) - 1]` est peut-être une petite valeur à insérer avant