Algorithmique et Programmation 1 : les listes chaînées

1re année Ensimag

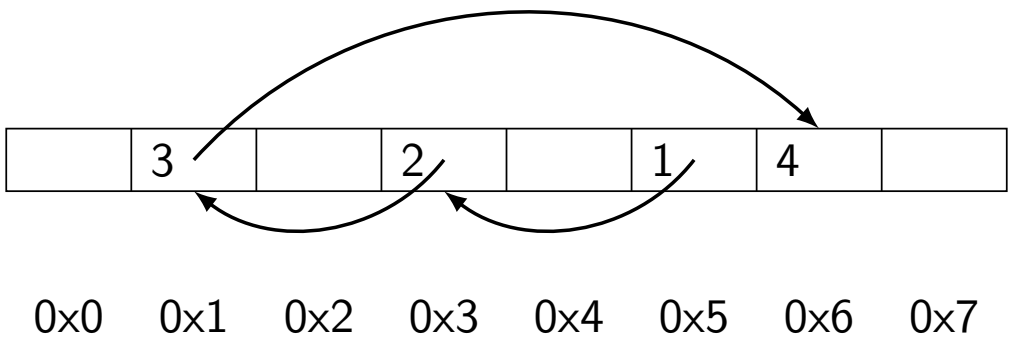
https://algo-prog-1.pages.ensimag.fr/web/

Rappel sur les tableaux

- Les éléments d'un tableau sont stockés de façon contiguë en mémoire
- On peut accéder à chaque élément en coût constant
  - `tab[i+1]` est stocké X octets après `tab[i]`, où X est la taille d'un élément
  - si A est l'adresse de début du tableau (c'est-à-dire l'adresse de la case d'indice 0), alors `tab[i]` est à l'adresse $A + i \times X$
- Un tableau statique a une taille (nombre d'éléments) fixée une fois pour toute
- Un tableau dynamique peut grandir, mais cela a un coût
  - s'il n'y a pas la place pour élargir le tableau, on doit en allouer un nouveau et recopier les éléments

Vocabulaire

- Une liste chaînée est une structure de données séquentielle dynamique (sans taille limite, sauf la capacité mémoire)
- Chaque élément de la liste s'appelle une *cellule*
- Chaque cellule contient
  - une valeur du type de données qu'on veut stocker
  - un *lien* (pointeur) vers la cellule suivante
- Les cellules ne sont pas (forcément) contiguës en mémoire, ni ordonnées
  - exemple : `1 -> 2 -> 3 -> 4`



Concrètement, en Go

```
type cellule struct {
    val  int
    suiv *cellule
}
```

- On manipule une liste via un pointeur vers la première cellule
  - qu'on appelle la *tête de liste*

```
type liste *cellule
var lsc liste
```

- Le champ `suiv` de la dernière cellule de la liste vaut `nil` ($\emptyset$)

Une liste chaînée est un type *construit*

- le programmeur sait que la 1re cellule pointe sur la 2e...
- Go ne « voit » que des cellules dispersées dans la mémoire

Impact sur les performances

- On ne peut pas accéder directement à la i^e cellule
  - on doit parcourir la liste à partir de la 1^{re} cellule

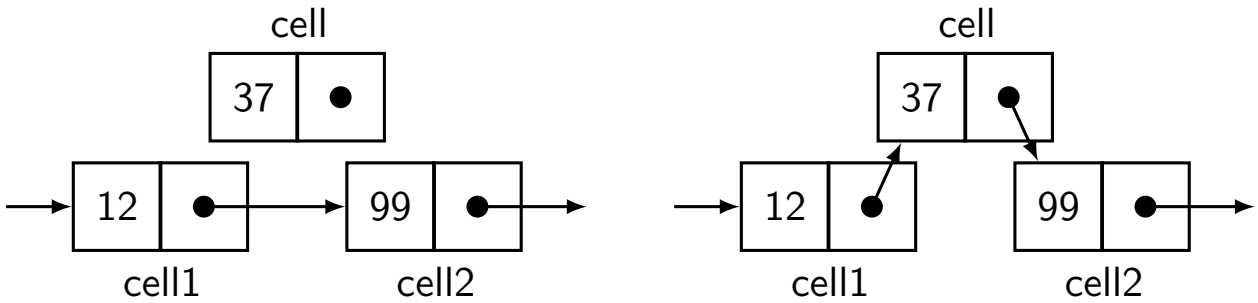
  $\Rightarrow$ coût linéaire

Parcourir une liste (dessins au tableau)

- On parcourt un tableau avec un indice et une boucle
  - `for idx := 0; idx < len(tab); idx++ {}`
- On parcourt une liste avec un pointeur sur une cellule (`cour`)
- Le pointeur est initialisé pour pointer sur la tête de liste
- On le fait avancer de champ `suiv` en champ `suiv`
  - `cour = cour.suiv` dans une boucle `for`
- On s'arrête quand `cour == nil`
  - ou quand `cour.suiv == nil` si on doit s'arrêter sur la dernière cellule

Ajouter un élément dans la liste (entre `cell1` et `cell2`)

1. Allouer une cellule, c'est-à-dire réserver de l'espace mémoire
   - En Go, tout simplement `cell := new(cellule)`
2. Initialiser ses champs
   - `cell.val = 37`
   - `cell.suiv = cell2`
3. chaîner la nouvelle cellule à sa place dans la liste
   - `cell1.suiv = cell`

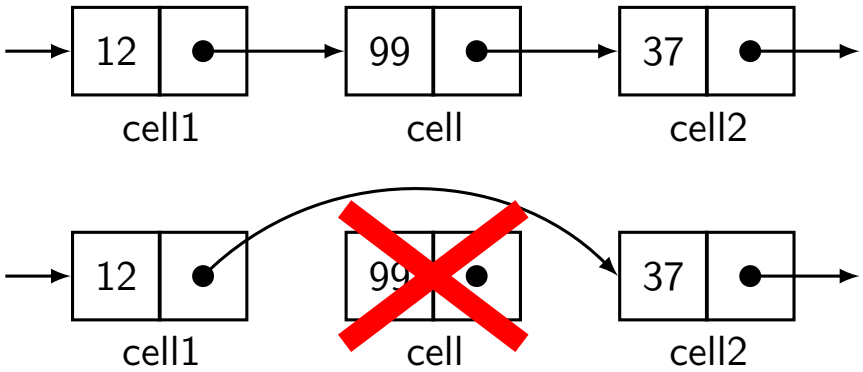


Questions au tableau

- Comment insérer en tête et en queue de liste ?

Supprimer un élément de la liste

1. Parcourir la liste pour s'arrêter sur la cellule (`cell1`) **précédant** celle qu'on veut supprimer (`cell`)
2. « Court-circuiter » la cellule à supprimer en chaînant `cell1.suiv` à `cell2`
3. Désallouer la cellule, c'est-à-dire libérer la mémoire
   - en Go, c'est fait tout seul par le ramasse-miette



Questions au tableau

- Comment supprimer en queue et en tête de liste ?

Technique de l'élément fictif en tête

- On alloue un « faux » élément avant la 1re cellule de la liste
- On chaîne l'élément fictif en tête de liste (`fictif.suiv = tete`) sans initialiser sa valeur (~~`fictif.val =`~~) (mais bon en Go ça sera 0...)
- On parcourt la liste en partant de l'élément fictif pour analyser le suivant
  - si c'est la tête de liste qui doit être supprimée, on est directement sur son élément précédent (le fictif)

  $\Rightarrow$ pas besoin de dissocier le cas de la première cellule
- À la fin de l'algorithme, on désalloue le fictif (en Go on laisse le GC le faire)
- On verra en TP plus concrètement à quoi ça sert

Schémas de programmation séquentielle

- Les schémas vus sur les tableaux s'appliquent aussi aux listes chaînées
- Parcours (type 1)

```
initialisation                      // cour = tete
tant que non fini                   // cour != nil
    traiter élément courant         // cour.val
    avancer                         // cour = cour.suiv
finalisation
```

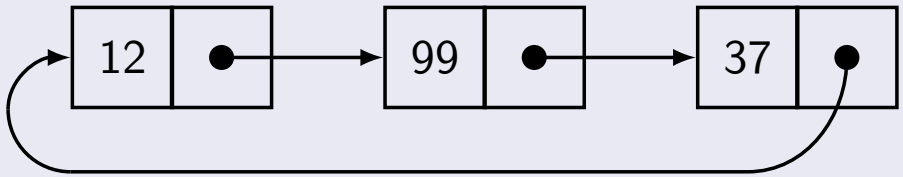
- Recherche

```
initialisation
// cour != nil && cour.val != ...
tant que non fini ET PUIS non trouvé
    avancer
finalisation
```

Listes triées

- On utilise parfois des listes triées (ex. listes de priorités)
- Peu de différences avec des listes classiques
  - insertion d'un élément à sa place dans la liste
  - recherche rapide, on s'arrête dès que `cour.val > valRech`

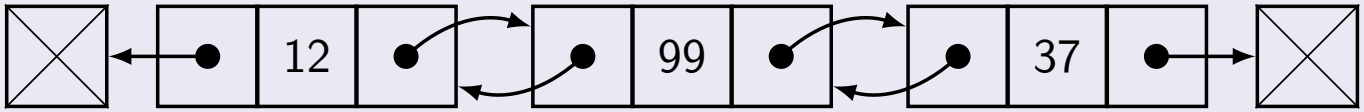
Listes circulaires



- Tout simplement : `queue.suiv == tete` et pas `nil`
- Utilisées par exemple pour implanter une file FIFO non-bornée
  - on garde un pointeur vers la queue (le plus récent)
  - on insère X après l'élément en queue (X devient la queue)
  - on retire l'élément en tête, c'est-à-dire `queue.suiv`

  $\Rightarrow$ insérer et retirer en coût constant

Listes doublement chaînées



- En plus du champ `suiv`, on a aussi un champ `prec`
- En général, on a aussi des éléments fictifs en tête et en queue
- On peut facilement parcourir la liste dans les deux sens
  - par exemple pour implanter une pile non-bornée avec insertion et retrait en queue

En résumé

- Toutes ces propriétés peuvent être combinées si besoin
  - on peut utiliser des listes doublement chaînées circulaires triées avec fictifs en tête et queue si c'est utile pour l'algorithme à implanter !

Consignes générales pour les exercices

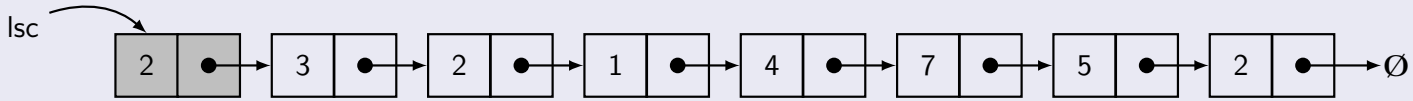
- On travaille toujours par modification du chaînage
  - les cellules ne changent jamais de place dans la mémoire
  - on ne modifie jamais les champs `val`
- On n'alloue jamais de nouvelle cellule
  - à l'exception d'éventuels éléments fictifs si nécessaire
- On respecte l'ordre de grandeur du coût demandé
  - comme pour les tableaux : coût linéaire, quadratique...

On peut utiliser autant de pointeurs que nécessaire

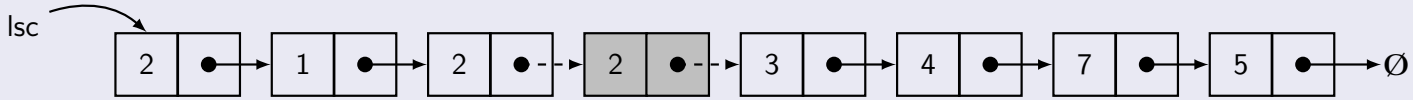
- Bien différencier
  - des variables pointeurs `cour := lsc`, `cour = cour.suiv...`
  - l'allocation de cellules `cell := new(cellule)`

Le pivot

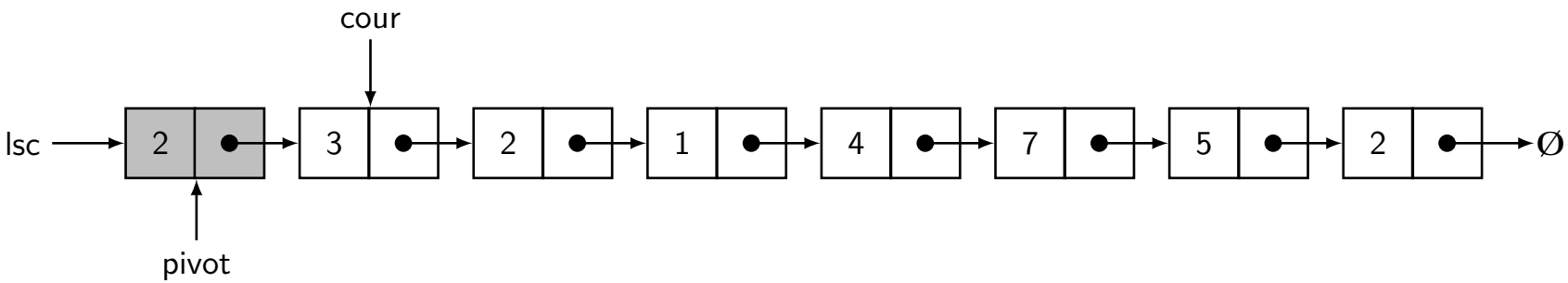
- On veut écrire la fonction `pivot(lsc liste) liste`
  - avec `type liste *cellule`
- On part d'une liste chaînée d'entiers non-vide
- On sélectionne un des éléments (par exemple le premier)

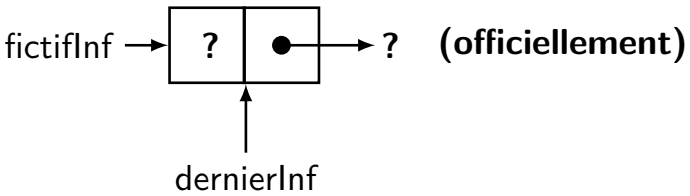


- On découpe la liste en deux sous-listes
  - les éléments inférieurs ou égaux au pivot
  - les éléments strictement supérieurs au pivot
- On finit par recoller les deux sous-listes à la fin (avec le pivot au milieu) pour construire une liste complète

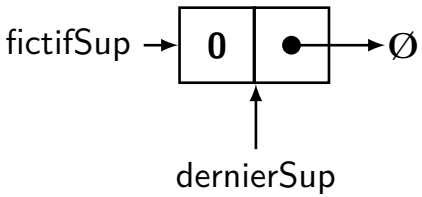


- État initial
  - on parcourt la liste initiale avec un pointeur `cour`
  - on doit garder le début et la fin des listes $\leq$ et $>$...
  - pour recoller les listes à la fin, pas pour préserver l'ordre
  - on utilise des fictifs pour homogénéiser les premières insertions



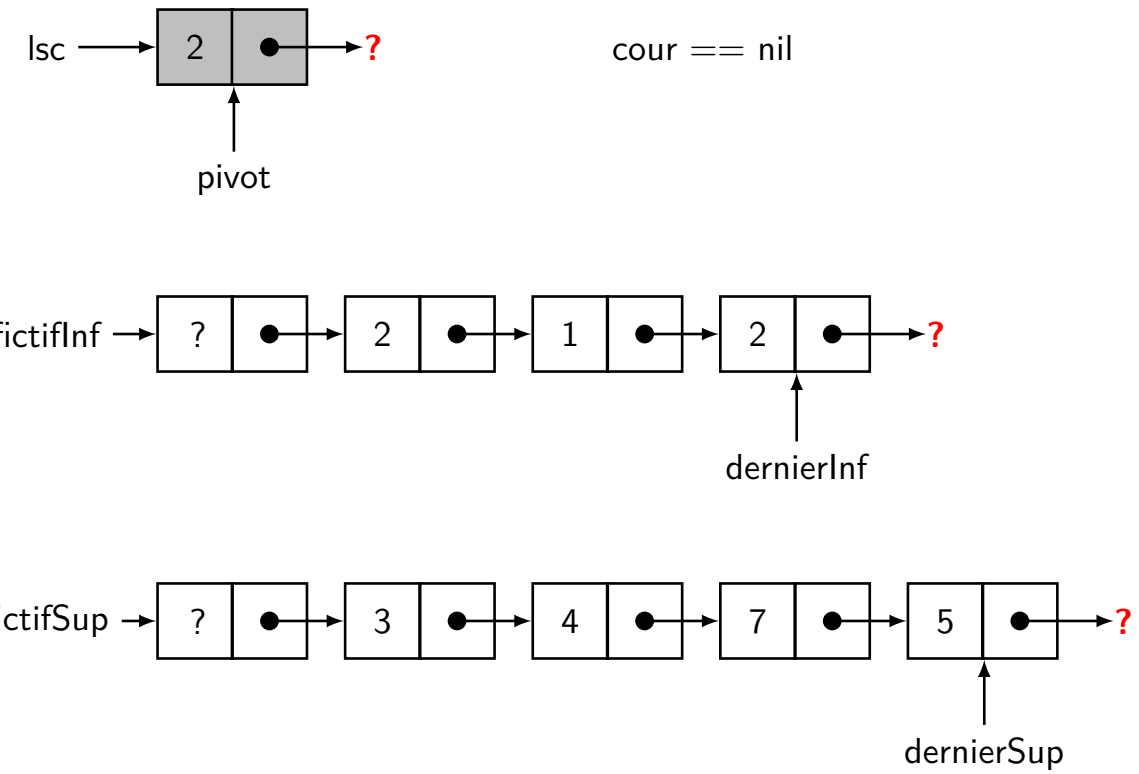


(officiellement)

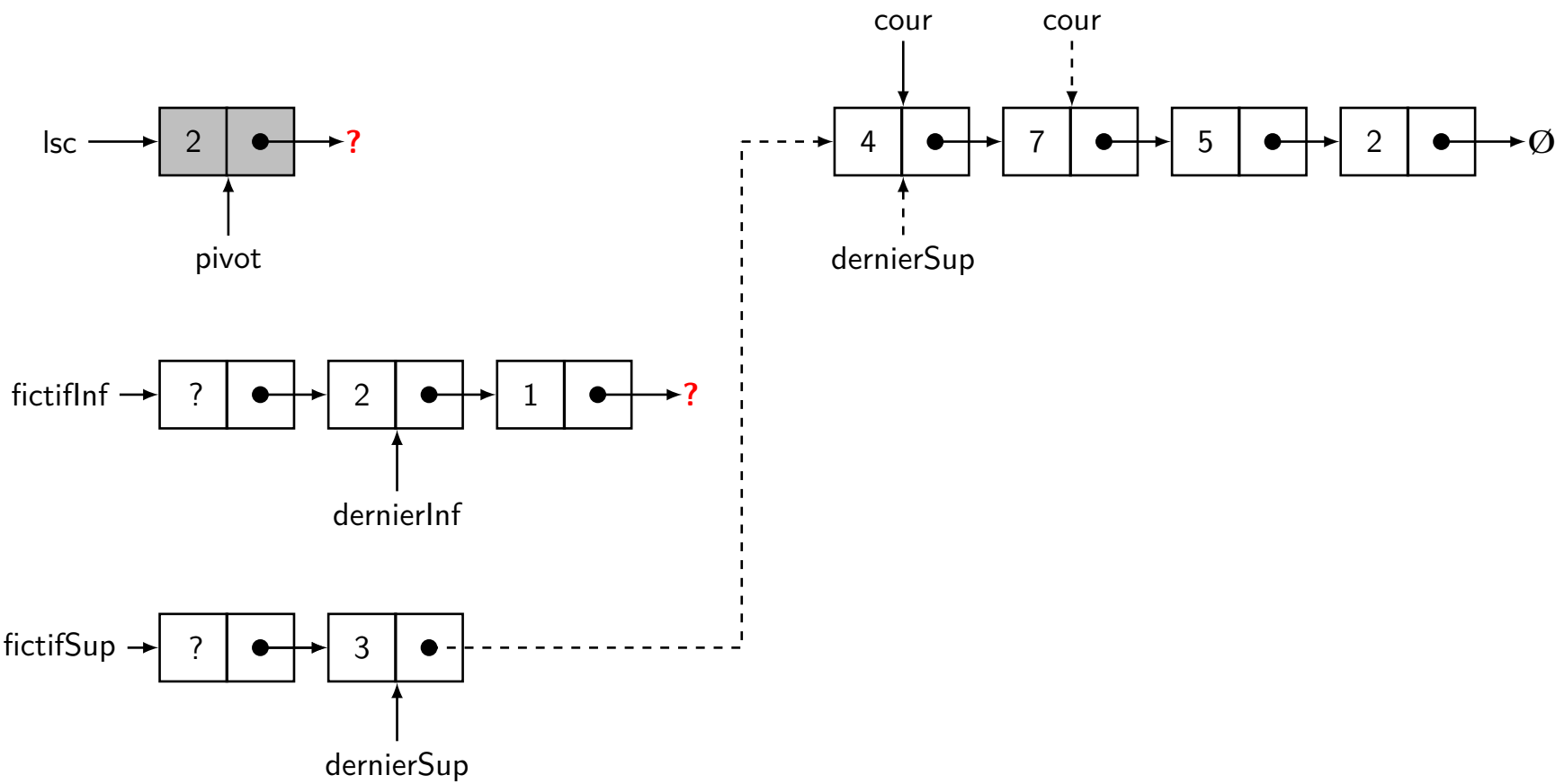


(en Go)

- État final (avant de recoller les parties)

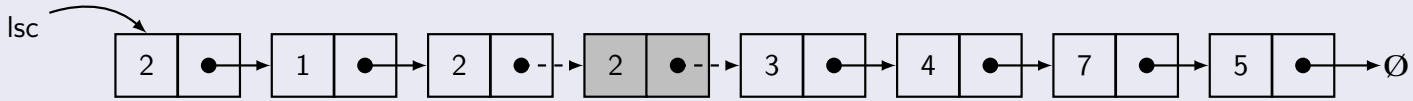


- État intermédiaire (i → i+1)



Le pivot

- On finalise en recollant les trois morceaux
  - `dernInf.suiv = pivot`
  - `pivot.suiv = fictifSup.suiv`
  - `dernSup.suiv = nil`
- Et on renvoie la nouvelle liste complète
  - `return fictifInf.suiv`



- Coût linéaire, en l'occurrence un seul parcours de la liste
- On utilisera la fonction avec `lsc = pivot(lsc)`
  - on pourrait aussi écrire une `func pivot(lsc *liste)` sans rien renvoyer
  - mais **surtout pas** `func pivot(lsc liste)` : pourquoi ?

Inversion d'une liste

- On veut inverser une liste chaînée (potentiellement vide)
  - par exemple `1 -> 2 -> 3 -> 4` devient `4 -> 3 -> 2 -> 1` (la liste n'est pas forcément triée)
  - uniquement par modification des chaînages **sans manipuler les valeurs**
  - **sans allouer de nouvelles cellules**
  - la fonction `inverser(lsc liste) liste` sera appelée par `lsc = inverser(lsc)`
- Coût attendu : linéaire (un seul parcours en l'occurrence)

Exercice (correction au tableau)

- Dessiner les états initial, final et intermédiaire de l'algorithme
- Indice : $\approx$ insertion en tête, mais sans allocation de cellule
- Pas besoin de fictif ici

Découpage en sous-listes

- On veut découper la liste en deux sous-listes
  - contenant les mêmes éléments, dans le même ordre
  - en coupant la liste initiale au milieu, à un élément près
  - `1 -> 2 -> 3 -> 4` donnera `1 -> 2` et `3 -> 4`
  - `1 -> 2 -> 3 -> 4 -> 5` donnera `1 -> 2 -> 3` et `4 -> 5`
- Coût attendu : linéaire (un seul parcours en fait)
- Toujours sans allouer de cellule (même pas de fictif)
- En modifiant les chaînages, sans manipuler les valeurs

Exercice (correction au tableau)

- Dessiner les états initial et final (interm. : pas intéressant !)
- Indice : le lièvre va deux fois plus vite que la tortue...
  - taille de la liste initiale : paire ou impaire ?
  - dérouler l'algo sur les deux exemples ci-dessus pour comprendre

Tri par sélection du maximum d'une liste

- On veut trier la liste par ordre croissant
  - au début la liste résultat est vide
  - on parcourt la liste initiale pour trouver l'élément maximal
  - on l'insère en tête de la liste résultat
  - on continue tant que la liste initiale n'est pas vide
- Toujours sans allouer de cellule (à part un fictif)
- En modifiant les chaînages, sans manipuler les valeurs
- Coût attendu : quadratique

Exercice (correction au tableau)

- Dessiner les états initial, final et intermédiaire de l'algorithme
- Indice : on aura sûrement besoin de chercher la cellule précédant le maximum
  - un fictif est intéressant en tête de la liste initiale

Tri par insertion d'une liste

- On veut trier la liste par ordre **décroissant** (pour changer)
  - au début la liste résultat est vide
  - on prend le premier élément de la liste initiale
  - on l'insère à sa place dans la liste résultat
  - on continue tant que la liste initiale n'est pas vide
- Coût attendu : quadratique
- Toujours sans allouer de cellule (à part un fictif)
- En modifiant les chaînages, sans manipuler les valeurs

Exercice (correction au tableau)

- Dessiner les états initial, final et intermédiaire de l'algorithme
- Indice : c'est un peu le principe réciproque du tri précédent
  - il est intéressant de mettre le fictif en tête de la liste résultat