Algorithmique E3

Algorithmes de tri

Daniel Vaz

daniel.ramosvaz@esiee.fr

2025-12-01

ESIEE PARIS

Annonces

TP2 et TP3 cette semaine

- TP2 : Algorithmes récursifs et tableaux
- TP3 : Algorithmes de tri
- Pensez à laisser le temps pour les exercices PLaTon !

Aperçu

- Gestion de tableaux
  - Itération
  - Allocation et initialisation de tableaux
- Algorithmes de tri
  - Tri à bulles (bubble sort)
  - Tri par sélection (selection sort)
  - Tri par insertion (insertion sort)
- Recherche dans un tableau

Gestion de tableaux

La structure d'un tableau

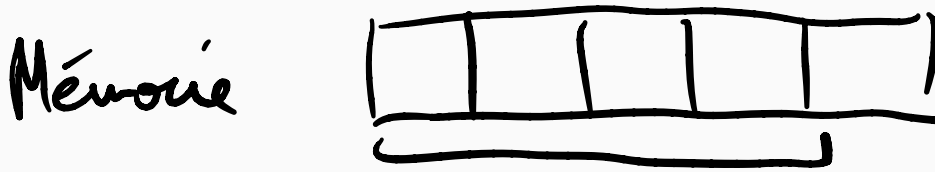


Tableau : une suite contiguë d'éléments de même type

La structure d'un tableau

Tableau : une suite contiguë d'éléments de même type

Les éléments sont indexés par la position (entier)

- Élément k du tableau t se notera $t[k]$
- La numérotation commence à partir de zéro
- Accès à un élément en temps constant

La limite du tableau est spécifiée :

- soit en précisant la longueur,
- soit en précisant une sentinelle ($'\backslash 0'$ pour les chaînes en C)

$t = [1, 2, 3]$, $n = 3$

Gestion de tableaux

Itération

Itération sur les tableaux

Sur les tableaux t avec longueur :

- fonctions doivent recevoir la longueur n
- itérer sur les éléments 0 à $n - 1$ (for / while)

Sur les tableaux t avec sentinelle

- fonctions peuvent recevoir la sentinelle s (ou elle est implicite)
- itérer jusqu'à la sentinelle

Itération en précisant la longueur

Algorithme EstTrie

Données: tableau t, longueur n de t

Résultat: **true** si t est trié, sinon **false**

```
1: for i de 1 à n − 1 do
2:     if t[i − 1] > t[i] then
3:         return false
4:     end if
5: end for
6: return true
```

Itération avec sentinelle

Algorithme LongueurChaine

Données: chaîne de caractères t (type C)

Résultat: longueur de la chaîne t

1: $\ell \leftarrow 0$
2: **while** $t[\ell] \neq '\backslash 0'$ **do**
3: $\quad \ell \leftarrow \ell + 1$
4: **end while**
5: **return** ℓ

Initialisation avec longueur

Algorithme Fibonacci

Données: tableau t, longueur n de t

Résultat: rien, tableau t est rempli avec la suite de Fibonacci

1: $t[0] \leftarrow 0$, $t[1] \leftarrow 1$
2: **for** i de 2 à $n-1$ **do**
3: $\quad t[i] \leftarrow t[i-1] + t[i-2]$
4: **end for**

Modification avec sentinelle

Algorithme Majuscule

Données: chaîne de caractères t (type C)

Résultat: rien, convertit t en majuscules

1: $i \leftarrow 0$
2: **while** $t[i] \neq '\backslash 0'$ **do**
3: **if** $'\mathtt{a}' \leq t[i] \wedge t[i] \leq '\mathtt{z}'$ **then**
4: $t[i] \leftarrow t[i] + ('\mathtt{A}' - '\mathtt{a}')$
5: **end if**
6: $i \leftarrow i + 1$
7: **end while**

	pseudo	C
et	$\wedge$	&&
ou	$\vee$	||
non	$\neg$	!

Itération sur les tableaux – Avantages

Tableau avec longueur :

- \+ Longueur en temps $\Theta(1)$
- \+ Itération en sens inverse
- – Toujours 2 paramètres

Tableau avec sentinelle

- \+ Un seule paramètre
- \+ Facile à modifier
- – Longueur en temps $\Theta(n)$

Gestion de tableaux

Allocation et initialisation de tableaux

Les tableaux dans la mémoire

Tableaux peuvent être alloués dans la mémoire pile ou tas

Mémoire pile : Stocke information sur l'exécution des fonctions

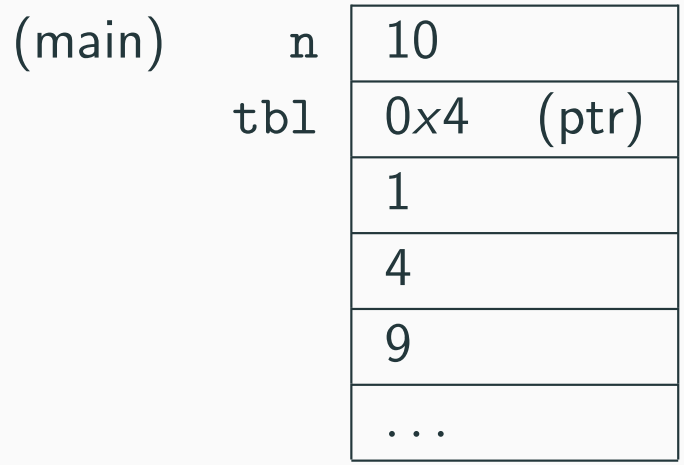
- Exemple : `int tbl[10];`
- Mémoire alloué pour une fonction et libéré à la sortie
- Facile à utiliser, normalement limité à 1–8 MB

Mémoire tas : Mémoire allouée manuellement

- Exemple : `int* tbl = (int*) malloc(10*sizeof(int));`
- Permet d'utiliser grandes quantités de mémoire
- Reste accessible dans toutes les fonctions
- Nécessite une libération manuelle
  - si on perd le pointeur → *memory leak*

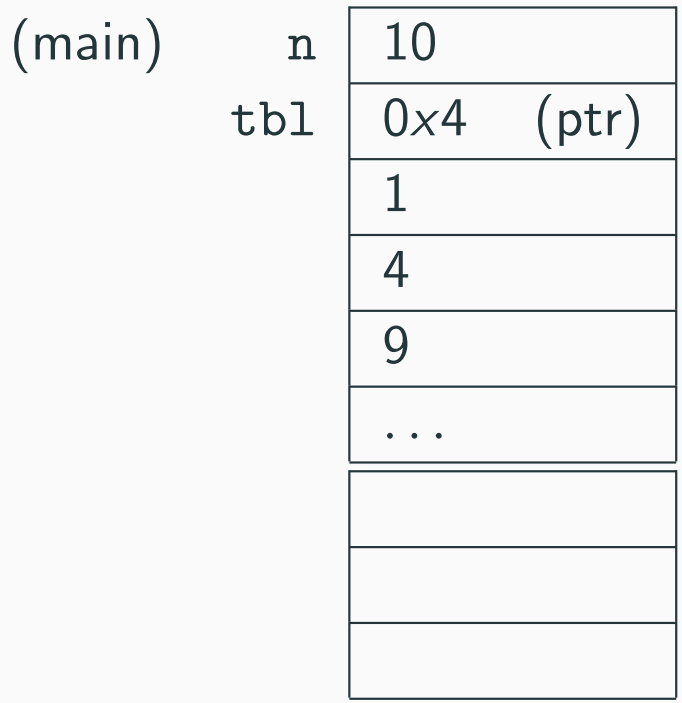
Allocation automatique

```
1 int main() {
2     int n = 10;
3     int tbl[n];
4     for (int i=0; i<n; i++) {
5         tbl[i] = (i+1)*(i+1);
6     }
7     // ...
8 }
```



Allocation automatique (2)

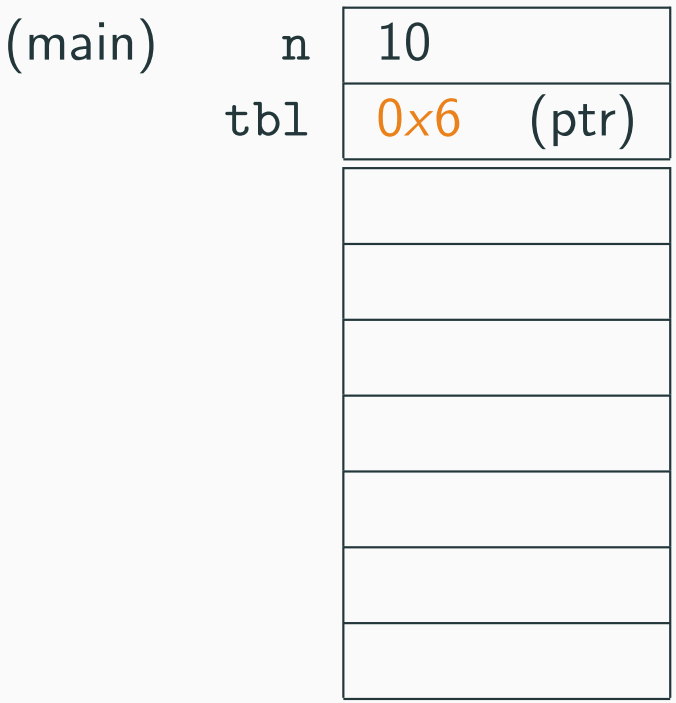
```
1  void init(int* tbl, int n) {
2      for (int i=0; i<n; i++) {
3          tbl[i] = (i+1)*(i+1);
4      }
5  }
6
7  int main() {
8      int n = 10;
9      int tbl[n];
10     init(tbl, n);
11     // ...
12 }
```



Allocation automatique (3)

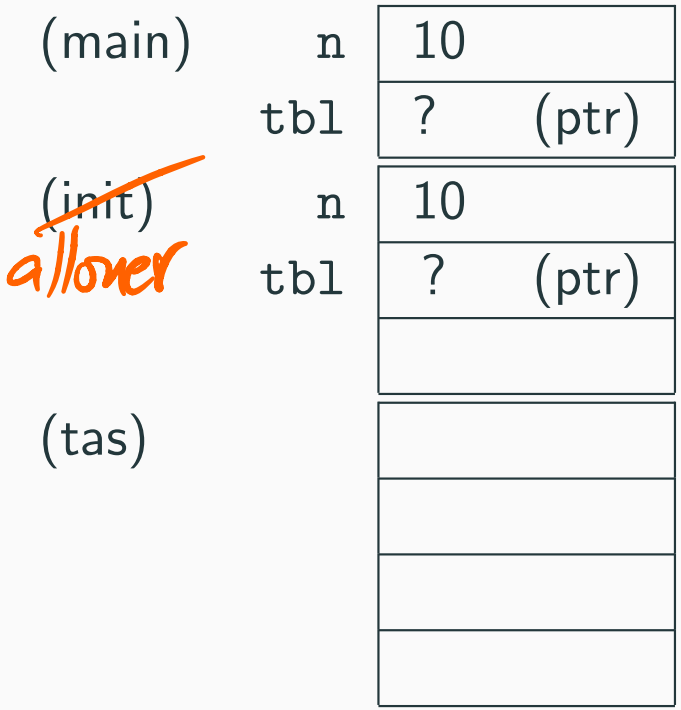
À ne pas faire ! → Segmentation fault

```
1  int* init(int n) {
2      int tbl[n];
3      for (int i=0; i<n; i++) {
4          tbl[i] = (i+1)*(i+1);
5      }
6      return tbl;
7  }
8
9  int main() {
10     int n = 10;
11     int* tbl = init(n);
12     // ...
13 }
```



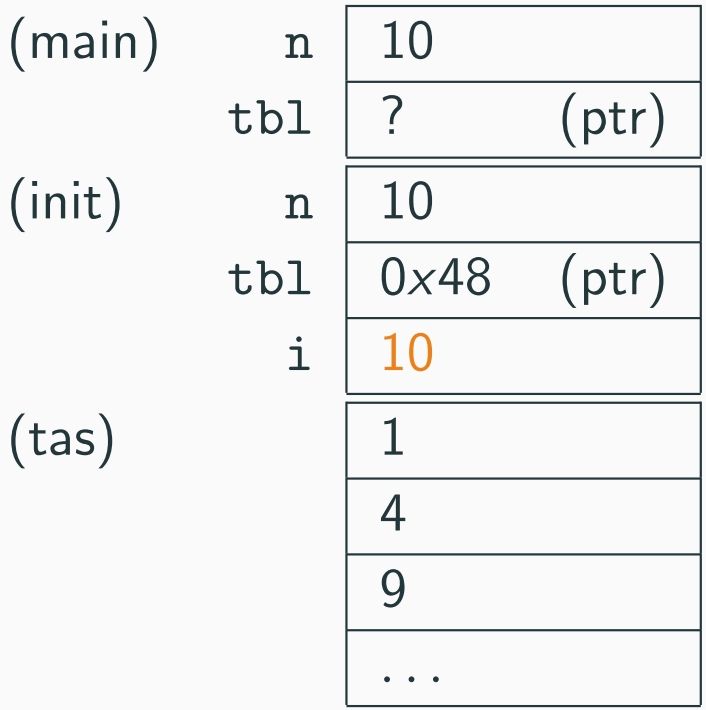
Allocation dynamique

```
1  int* allouer(int n) {
2      int* tbl = (int*)
           malloc(n*sizeof(int));
3      for (int i=0; i<n; i++) {
4          tbl[i] = (i+1)*(i+1);
5      }
6      return tbl;
7  }
8
9  int main() {
10     int n = 10;
11     int* tbl = allouer(n);
12     // ...
13     free(tbl);
14 }
```



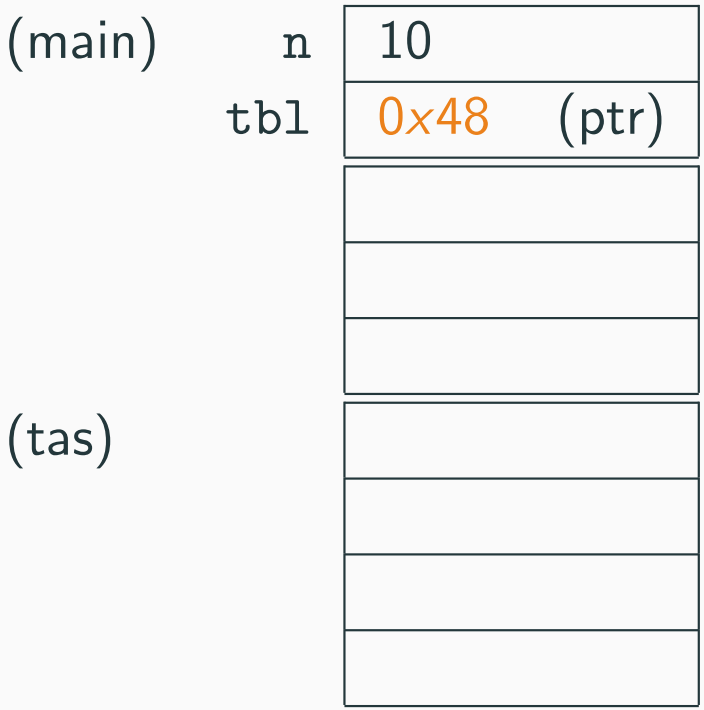
Allocation dynamique

```
1  int* allouer(int n) {
2      int* tbl = (int*)
           malloc(n*sizeof(int));
3      for (int i=0; i<n; i++) {
4          tbl[i] = (i+1)*(i+1);
5      }
6      return tbl;
7  }
8
9  int main() {
10     int n = 10;
11     int* tbl = allouer(n);
12     // ...
13     free(tbl);
14 }
```



Allocation dynamique

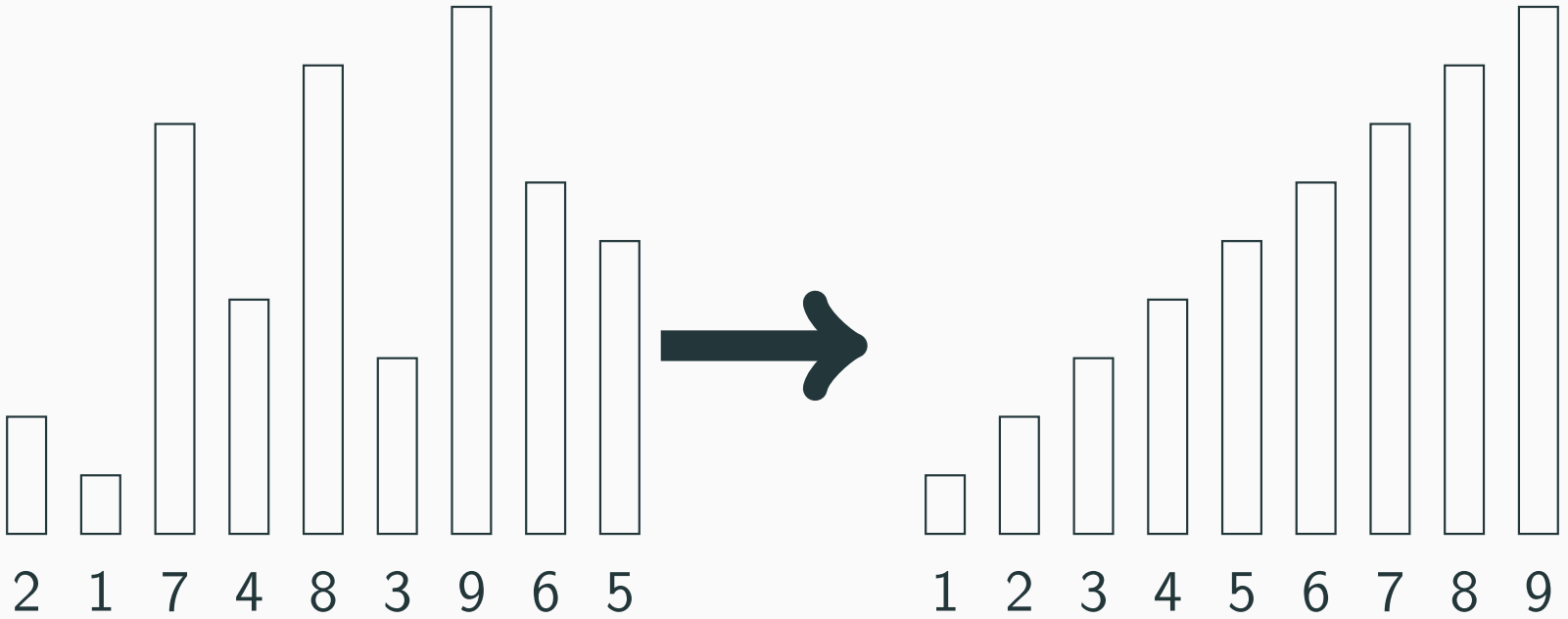
```
1  int* allouer(int n) {
2      int* tbl = (int*)
           malloc(n*sizeof(int));
3      for (int i=0; i<n; i++) {
4          tbl[i] = (i+1)*(i+1);
5      }
6      return tbl;
7  }
8
9  int main() {
10     int n = 10;
11     int* tbl = allouer(n);
12     // ...
13     free(tbl);
14 }
```



Algorithmes de tri

Problème de tri

Trier : réarranger les éléments de sorte que $t[i] \leq t[j]$ si $i < j$
(mais en fait, il suffit de vérifier que $t[i-1] \leq t[i]$)



Algorithme de tri

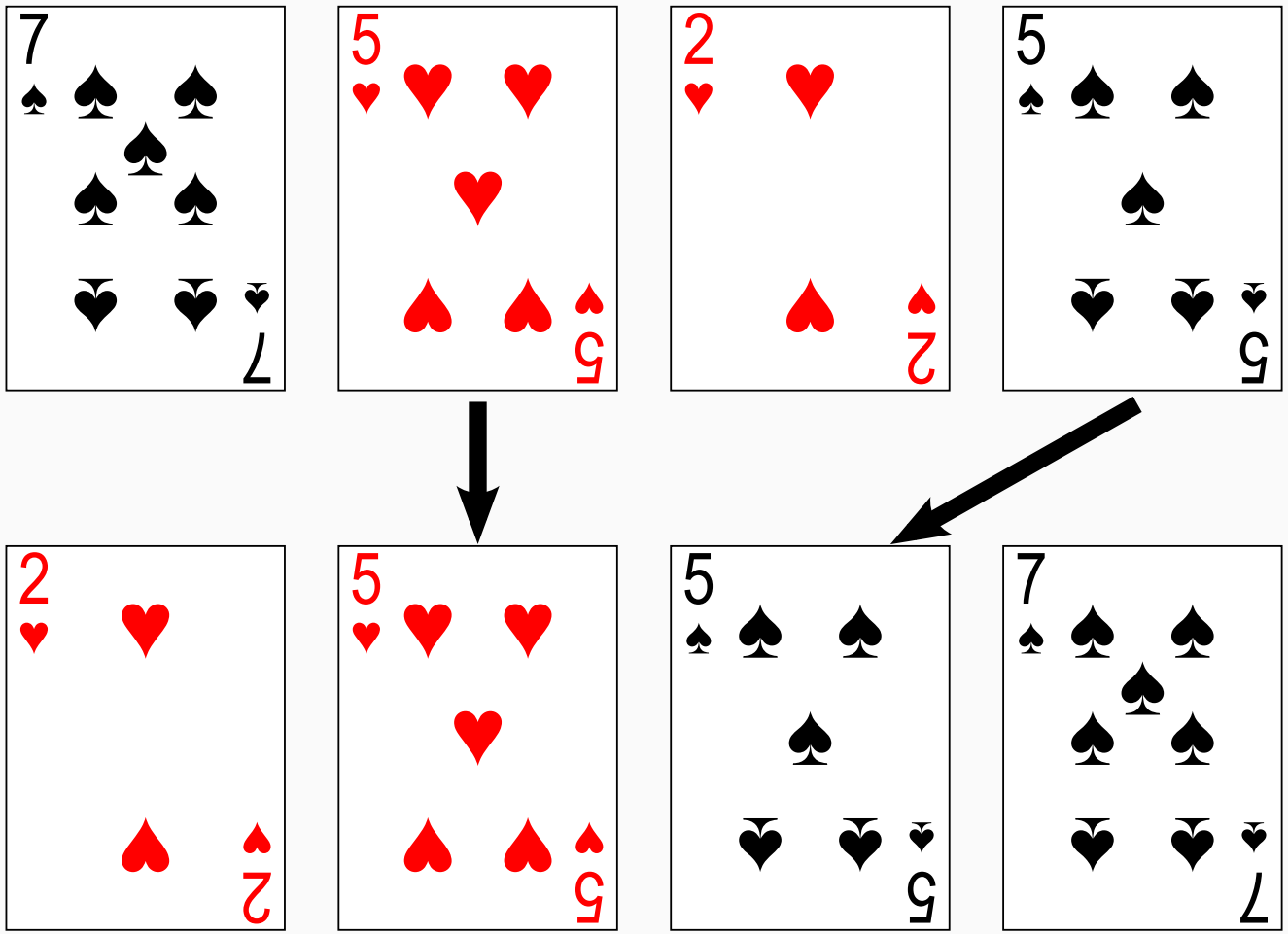
- Prennent un tableau en entrée, retournent ~~la liste~~ le tableau (ou rien)

Algorithme de tri

- Prennent un tableau en entrée, retournent la liste (ou rien)
- Tri en place : modifie directement le tableau d'entrée
  - Économe en mémoire, car il n'a alloue pas une nouvelle liste
  - Pas nécessaire de retourner la liste
- Tri stable : préserve l'ordre initial des éléments que sont considérés comme égaux
  - Tri d'objets complexes en fonction d'un attribut
  - Utile quand multiples opérations de tri sont exécutés
  - Trier ["Pomme", "Poire", "Orange"] par le première caractère

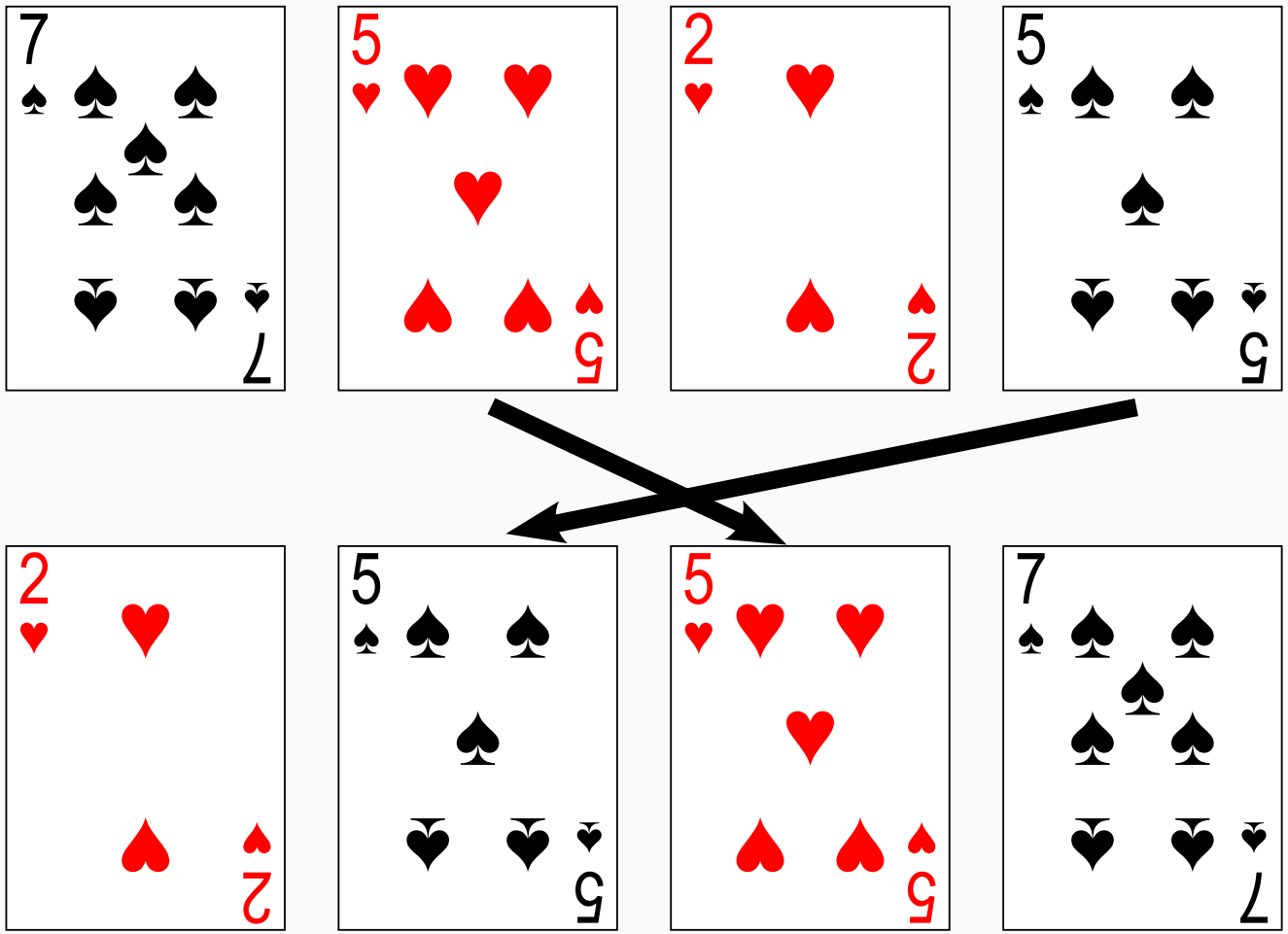
→ Orange, Pomme, Poire

Tri stable



Stable

Tri stable



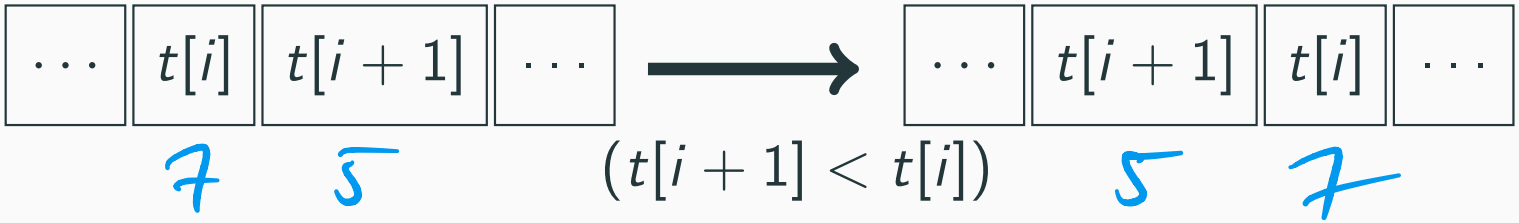
Pas stable

Algorithmes de tri

Tri à bulles (bubble sort)

Tri à bulles

- Une opération simple : permutation d'éléments adjacents



Tri à bulles

- Une opération simple : permutation d'éléments adjacents

$(t[i+1] < t[i])$

- Balayage :
  - Parcourir le tableau en entier,
  - En comparant les éléments consécutifs,
  - Si pas ordonnés, on les permute.
- Chaque balayage déplace un élément à sa position finale
  - i-ème balayage amène le i-ème plus grand élément.

n balayages ⟶ tableau trié

Permutation (Annexe)

Problème : Permuter les contenus de x et y.

Idée : Utiliser une variable d'échange

Algorithme Permute

Données: x et y éléments de type s

1: $tmp \leftarrow x$ *// Variable d'échange*
2: $x \leftarrow y$
3: $y \leftarrow tmp$

Permutation (Annexe)

Code C

```c
void permute(double * px, double * py){
    double z = *px;
    *px = *py;
    *py = z;
}
```

Algorithme de tri à bulles

Algorithme TriBulleBase

Données: Tableau t de type s, longueur du tableau n

```
1: for 1 ≤ b < n do
2:     for 0 ≤ i < n − b do
3:         if t[i + 1] < t[i] then
4:             PERMUTE(t[i + 1], t[i])
5:         end if
6:     end for
7: end for
```

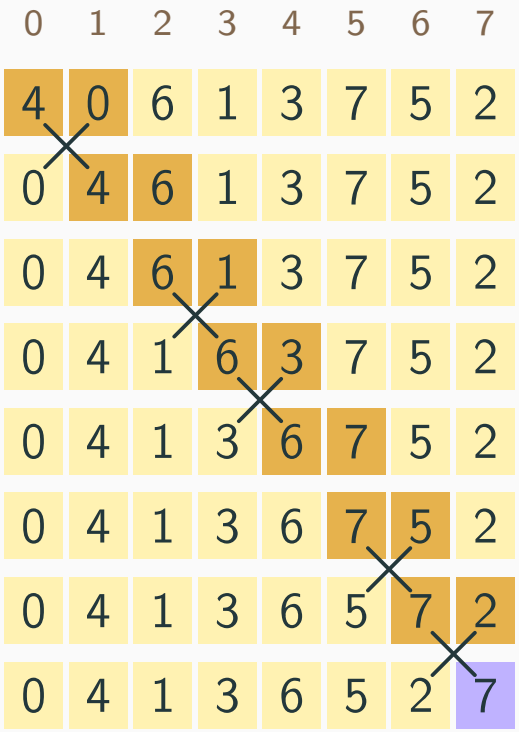
Exemple

Trier le tableau $t = [4, 0, 6, 1, 3, 7, 5, 2]$ en appliquant le tri à bulles

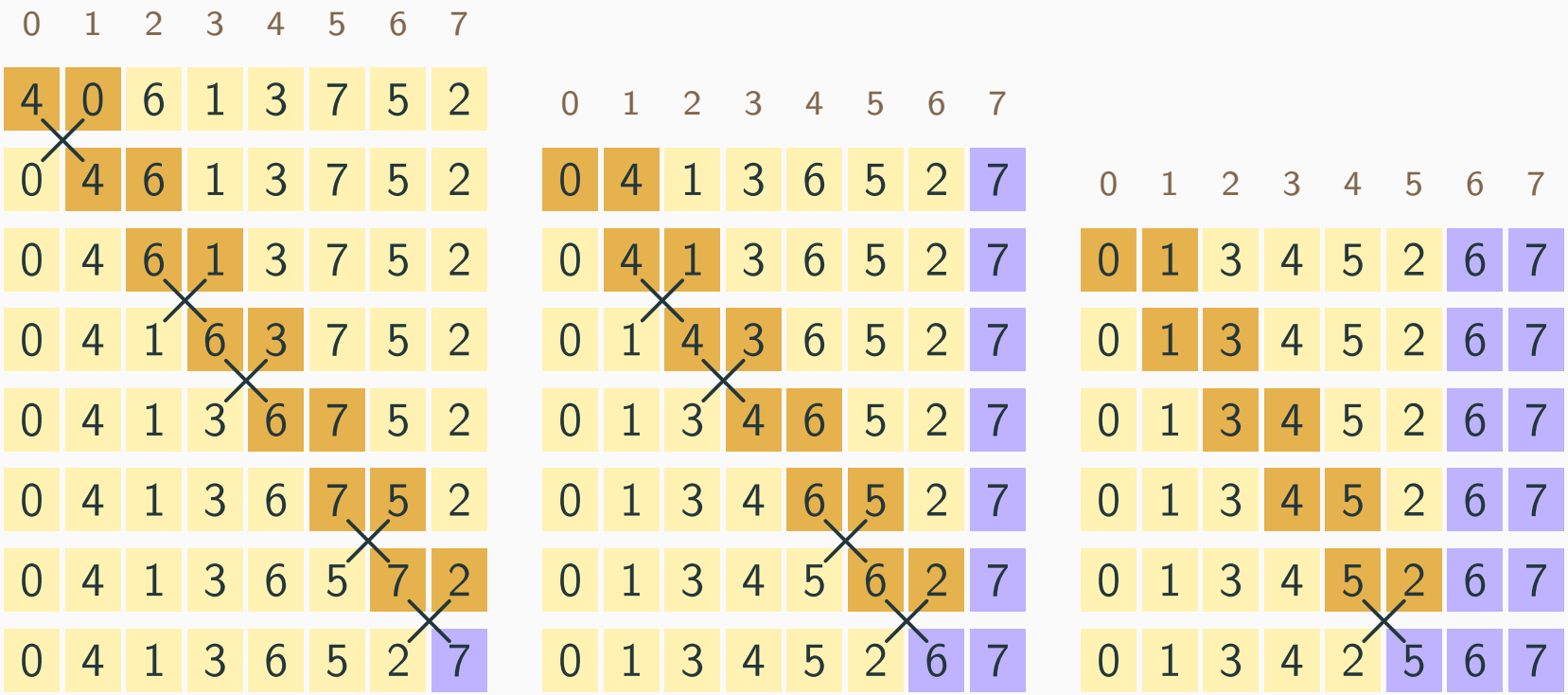
- Calculer le nombre d'opérations nécessaires pour l'exécuter
- Déduire la complexité de l'algorithme

Exécution d'un tri à bulles (1)

balayage 1

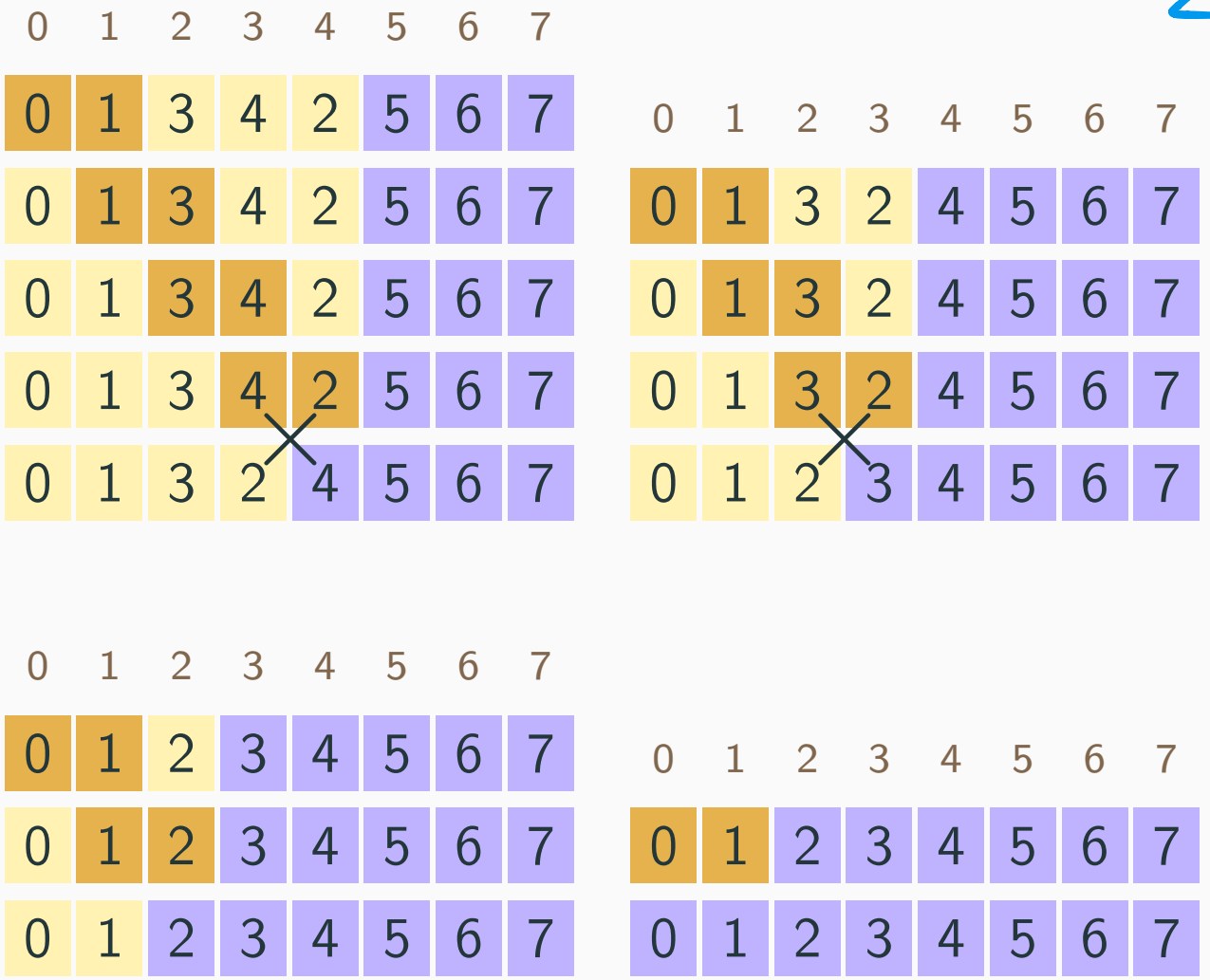


Exécution d'un tri à bulles (1)



Exécution d'un tri à bulles (2)

$n : \frac{n(n+1)}{2} = \Theta(n^2)$



Opérations : $7+6+5+4+3+2+1 = 28$

Exemple

Trier le tableau $t = [4, 0, 6, 1, 3, 7, 5, 2]$ en appliquant le tri à bulles

- Calculer le nombre d'opérations nécessaires pour l'exécuter
- Déduire la complexité de l'algorithme $\Theta(n^2)$

Exemple

Trier le tableau $t = [4, 0, 6, 1, 3, 7, 5, 2]$ en appliquant le tri à bulles

- Calculer le nombre d'opérations nécessaires pour l'exécuter
- Déduire la complexité de l'algorithme

Comment peut-on améliorer l'algorithme ?

- S'arrêter si la liste est déjà trié
- Balayage bidirectionnel $\rightarrow$ tri cocktail
- Comparer avec des éléments plus loin $\rightarrow$ tri à peigne

Algorithme de tri à bulles optimisé

Algorithme TriBulleOpt

Données: Tableau t de type s, longueur du tableau n

```
1:  fini ← false
2:  while fini = false do
3:      fini ← true
4:      for 0 ≤ i < n do
5:          if t[i + 1] < t[i] then
6:              PERMUTE(t[i + 1], t[i])
7:              fini ← false
8:          end if
9:      end for
10:     n ← n − 1
11: end while
```

→ supposer que c'est trié

Exemple

Trier le tableau $t = [0, 1, 2, 3, 4, 5, 6, 7]$ en appliquant le tri à bulles

- Calculer le nombre d'opérations nécessaires pour l'exécuter
- Déduire la complexité de l'algorithme en meilleur cas

base : 28 op $\rightarrow \Theta(n^2)$

amélioré : 7 op $\rightarrow \Omega(n)$

Tri à bulles (optimisé)

Complexité :

- Meilleur cas : $\Theta(n)$
- Cas moyen : $\Theta(n^2)$
- Pire cas : $\Theta(n^2)$

Algorithme de tri en place et stable :

- En place : seule la liste d'entrée est nécessaire
- Stable : deux éléments égaux ne sont pas échangés

Algorithmes de tri

Tri par sélection (selection sort)

Tri par sélection

- Opération : sélectionner le plus grand élément :
  - On cherche dans le tableau le plus grand élément
  - On permute cet élément avec le dernier élément du tableau

($t[i]$ maximal)

- Après, on recommence sur le reste du tableau
- La i-ème itération positionne le i-ème plus grand élément

Algorithme de tri par sélection

Algorithme TriSelection

Données: Tableau t de type s, longueur du tableau n

1: **while** $n > 1$ **do**
2: $p \leftarrow \text{IndiceMax}(t, n)$
3: $\text{Permute}(t[p], t[n-1])$
4: $n \leftarrow n - 1$
5: **end while**

for i=0 à n-1
p ← INDICEMAX(t, n-i)
PERMUTE(t[p], t[n-i-1])
end for.

Indice plus grand (annexe de selection sort)

Problème

Recherche de l'indice du plus grand élément.

Algorithme IndiceMax

Données: Tableau t de type s, longueur du tableau n

Résultat: Position p l'indice du plus grand élément

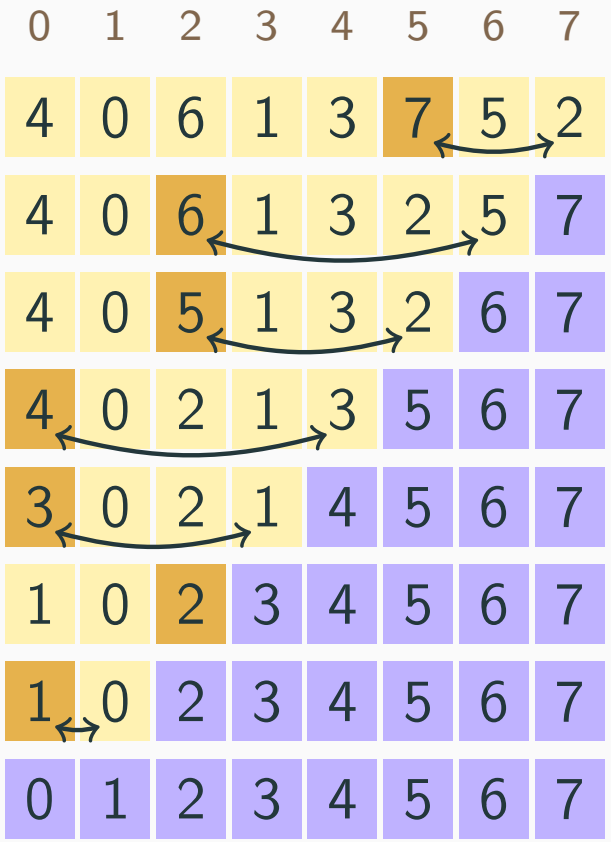
```
1: p ← 0
2: for 1 ≤ i < n do
3:     if t[i] > t[p] then
4:         p ← i
5:     end if
6: end for
7: return p
```

Exemple

Trier $t = [4, 0, 6, 1, 3, 7, 5, 2]$ en appliquant le tri par sélection

- Calculer le nombre d'opérations nécessaires pour l'exécuter
- Déduire la complexité de l'algorithme

Exécution d'un tri par sélection



7 comparaisons
6
5
4
3
2
1

28 → $\Theta(n^2)$

Tri par sélection

Complexité :

- Meilleur cas : $\Theta(n^2)$
- Cas moyen : $\Theta(n^2)$
- Pire cas : $\Theta(n^2)$

Algorithme de tri en place, mais pas stable :

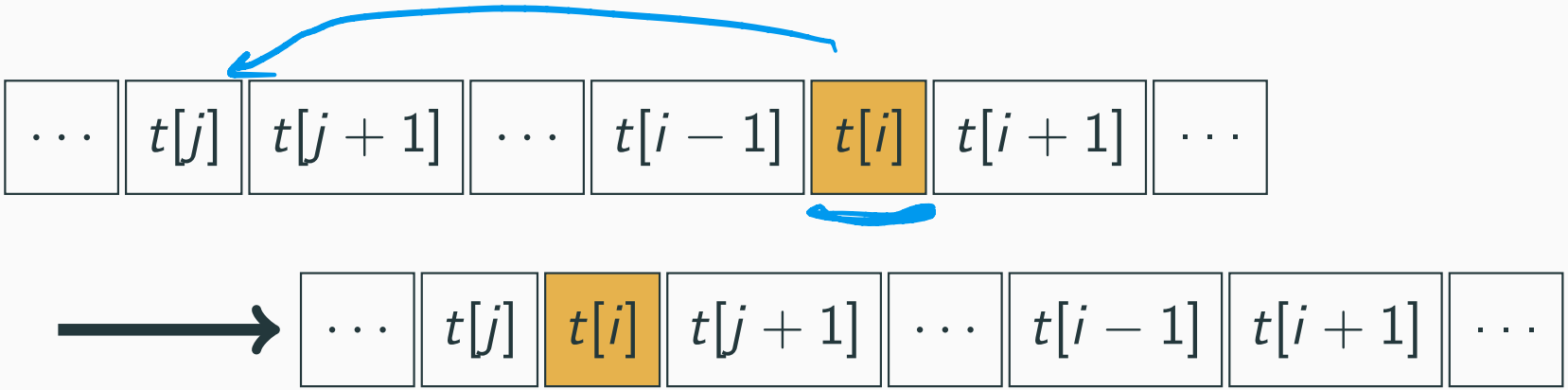
- En place : seule la liste d'entrée est nécessaire

Algorithmes de tri

Tri par insertion (insertion sort)

Tri par insertion

- Opération : insérer un élément dans sa position à gauche :
  - Supposons que le sous-tableau $t[0..i-1]$ est trié
  - On insère l'élément $t[i]$ parmi $t[0..i-1]$
  - Par conséquence, $t[0..i]$ est trié



$(t[j] \leq t[i] < t[j+1])$

Algorithme de tri par insertion

Algorithme TriInsertion

Données: Tableau t de type s, longueur du tableau n

```
1: for 1 ≤ i < n do
2:     x ← t[i]
3:     j ← i
4:     while j > 0 ∧ t[j − 1] > x do
5:         t[j] ← t[j − 1]
6:         j ← j − 1
7:     end while
8:     t[j] ← x
9: end for
```

décalage

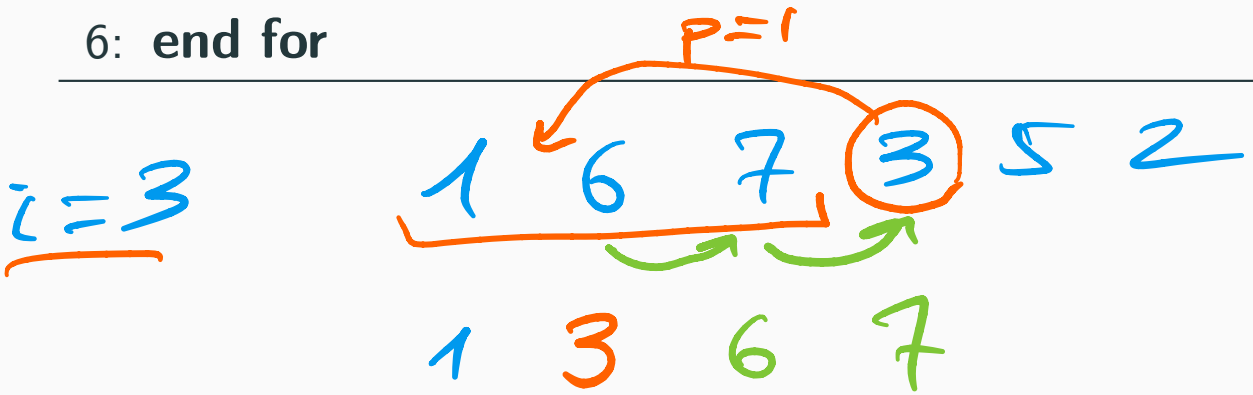
t[0..i] sera trié

Tri par insertion (insertion sort)

Algorithme TriInsertion2

Données: Tableau t de type s, longueur du tableau n

```
1: for 1 ≤ i < n do
2:     x ← t[i]
3:     p ← RechercheDicho(t, i, x)
4:     Decalage(t, p, i − 1)
5:     t[p] ← x
6: end for
```

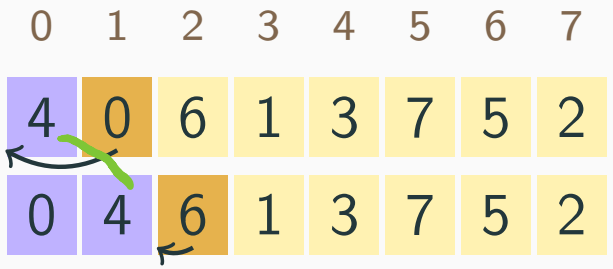


Exemple

Trier $t = [4, 0, 6, 1, 3, 7, 5, 2]$ en appliquant le tri par insertion

- Calculer le nombre d'opérations nécessaires pour l'exécuter
- Déduire la complexité de l'algorithme

Exécution d'un tri par insertion



Exécution d'un tri par insertion

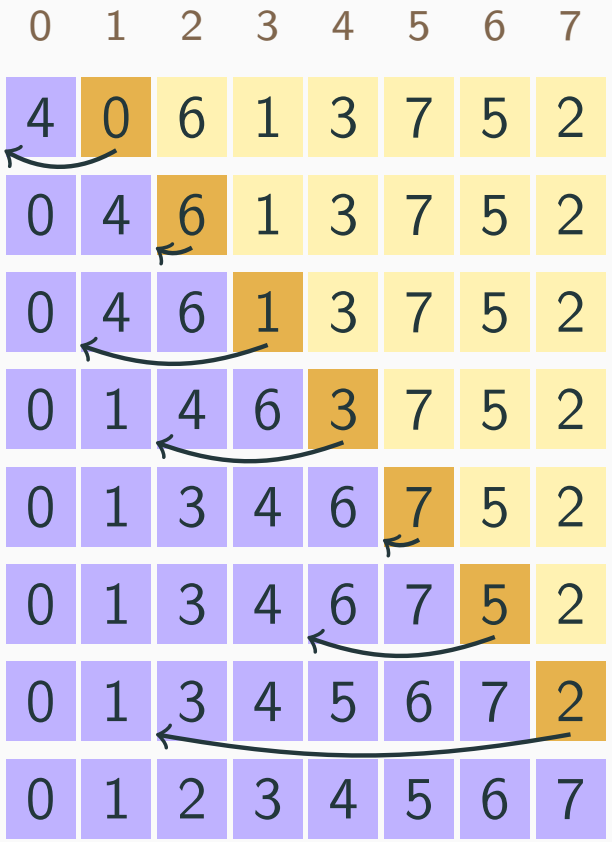
0	1	2	3	4	5	6	7
4	0	6	1	3	7	5	2
0	4	6	1	3	7	5	2
0	4	6	1	3	7	5	2

p=1

Exécution d'un tri par insertion

Exécution d'un tri par insertion

Exécution d'un tri par insertion



Tri par insertion

Complexité :

- Meilleur cas : $\Theta(n)$
- Cas moyen : $\Theta(n^2)$
- Pire cas : $\Theta(n^2)$

(code TRIINSERTION)

Tri par insertion

Complexité :

- Meilleur cas : $\Theta(n)$
- Cas moyen : $\Theta(n^2)$
- Pire cas : $\Theta(n^2)$

Algorithme de tri en place et stable :

- En place : seule la liste d'entrée est nécessaire
- Stable : l'ordre des éléments égaux est préservée

Recherche dans un tableau

Recherche de base dans un tableau

Problème

Rechercher la première occurrence d'une valeur dans un tableau.

Retourne sa position si elle existe, -1 sinon.

Idée

Parcours du tableau jusqu'à trouver la valeur cherchée ou la fin.

Recherche de base dans un tableau

Algorithme RechercheBase

Données: tableau t, n longueur du tableau, élément x

Résultat: position de x dans t, ou -1 si pas présent

```
1: i ← 0
2: while i < n ∧ t[i] ≠ x do
3:     i ← i + 1
4: end while
5: if i < n then
6:     return i
7: else
8:     return −1
9: end if
```

list.index → $\mathcal{O}(n)$

Recherche séquentielle dans un tableau trié

Et si le tableau est trié ?

Algorithme PosSeq

Données: tableau trié t, n longueur du tableau, élément x

Résultat: position de x dans t, ou -1 si pas présent

```
1: i ← 0
2: while i < n ∧ t[i] < x do
3:     i ← i + 1
4: end while
5: if i < n ∧ t[i] = x then
6:     return i
7: else
8:     return −1
9: end if
```

Recherche dichotomique dans un tableau trié

Peut-on faire mieux ?

Idée

- Tester l'élément au milieu du tableau
- Puis recommencer dans la moitié pertinente du tableau

Recherche dichotomique dans un tableau trié

Algorithme RechercheDicho

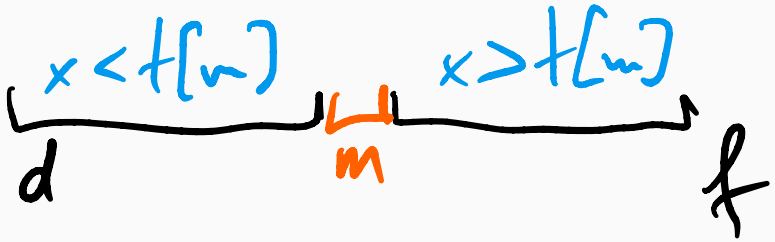
Données: tableau trié t, n longueur du tableau, élément x

Résultat: position de x dans t, ou -1 si pas présent

```
1:  d ← 0, f ← n
2:  while d < f do
3:      m ← (d + f)/2
4:      if t[m] = x then
5:          return m
6:      else if t[m] < x then
7:          d ← m + 1
8:      else
9:          f ← m
10:     end if
11: end while
12: return −1
```

trouver x entre d (inclus) et f (exclus)

// Division entière



Recherche dichotomique récursive

Algorithme RechercheDichoRecursive

Données: tableau trié t, d début, f fin, élément x

Résultat: position de x dans $t[d..f-1]$, ou -1 si pas présent

```
1: if t[d] = x then
2:     return d
3: else if f − d ≤ 1 then
4:     return −1
5: end if
6: m ← (d + f)/2                                  // Division entière
7: if t[m] ≤ x then
8:     return RechercheDichoRecursive(t, m, f, x)
9: else
10:    return RechercheDichoRecursive(t, d, m, x)
11: end if
```

d=3, f=4

sous-pb a=1, taille b=2 opérations f(n)=Θ(1)

Diviser pour régner – Master Theorem

Un problème : quelques opérations + plusieurs sous-problèmes.

Theorem

Si $C(n) = aC(n/b) + O(n)$ *(avec* $C(1)$ *constante)*

- *si* $a < b$: $C(n) = O(n)$
- *si* $a = b$: $C(n) = O(n \log n)$
- *si* $a > b$: $C(n) = O(n^{\log_b a})$

Diviser pour régner – Master Theorem

$a=1$, $b=2$ $f(n)=\Theta(1)$

Theorem (Master Theorem)

Si $C(n) = aC(n/b) + f(n)$, $\lambda = \log_b a = 0$

- *si* $f(n) = O(n^{\lambda-\varepsilon})$, *avec* $\varepsilon > 0$, *alors* $C(n) = \Theta(n^{\lambda})$,
- *si* $f(n) = \Theta(n^{\lambda})$, *alors* $C(n) = \Theta(n^{\lambda} \log n)$, $= \Theta(\log n)$
- *si* $f(n) = \Omega(n^{\lambda+\varepsilon})$, *avec* $\varepsilon > 0$,
  et il existe $c < 1$ *tel que* $a \cdot f(n/b) \leq c \cdot f(n)$ *pour* $n > n_0$,
  alors, $C(n) = \Theta(f(n))$

Recherche dichotomique :

$C(n) = 1 \cdot C(n/2) + \Theta(1) \rightarrow \lambda = \log_2 1 = 0$

$C(n) = \Theta(\log n)$

Diviser pour régner – Master Theorem

Theorem (Master Theorem)

Si $C(n) = aC(n/b) + f(n), \quad \lambda = \log_b a$

- *si* $f(n) = O(n^{\lambda-\varepsilon})$, *avec* $\varepsilon > 0$, *alors* $C(n) = \Theta(n^\lambda)$,
- *si* $f(n) = \Theta(n^\lambda)$, *alors* $C(n) = \Theta(n^\lambda \log n)$,
- *si* $f(n) = \Omega(n^{\lambda+\varepsilon})$, *avec* $\varepsilon > 0$,
  et il existe $c < 1$ *tel que* $a \cdot f(n/b) \leq c \cdot f(n)$ *pour* $n > n_0$,
  alors, $C(n) = \Theta(f(n))$

Recherche dichotomique :

$C(n) = 1 \cdot C(n/2) + \Theta(1) \rightarrow \lambda = \log_2 1 = 0$

$f(n) = \Theta(1) = \Theta(n^\lambda)$, alors $C(n) = \Theta(n^\lambda \log n) = \Theta(\log n)$