Algorithmique E3

Structures de données

Daniel Vaz

daniel.ramosvaz@esiee.fr

2026-01-05

Examen

Quoi étudier ?

CM :

- Supports de cours
- Algorithmes, structures, caractéristiques, ...

TD :

- Résoudre tous les exercices
- Bien comprendre les contenus autour des exercices
- Être capable de résoudre exercices simples rapidement

TP :

- Mettre en œuvre les algorithmes
- Résoudre les exercices plus difficiles en pseudo-code
- Bien connaître les particularités de l'implémentation en C

Quoi étudier ? – CM

Important : être capable d'expliquer le fonctionnement et appliquer les algorithmes, structures, ...

Pas nécessaire de mémoriser, mais pouvoir recréer l'algorithme :

- Avec le même fonctionnement
- Peut être avec du code différent
- Exceptions : Partition du tri rapide, *master theorem*
  - Il faut savoir que fait une fonction de partition
  - La fonction / le théorème seront données si nécessaire

Exemple : tri par insertion

Quoi étudier ? – TD

Les exercices de l'examen sont similaires à ceux du TD.

- Résoudre tous les exercices
  - Pratique directe pour les exercices de l'examen
- Bien comprendre les contenus autour des exercices
- Être capable de résoudre exercices simples bien et rapidement
  - $f \in O(g)$, exécuter des algorithmes, ...
  - TD2 Ex. 1, 2, 6
- Pratiquer exercices difficiles pour développer vos compétences
  - TD2 Ex. 3 – 5

Exercice simple Appliquer le tri sélection au tableau :
$[6, 7, 2, 4, 8, 9, 11, 1]$.

Exercice plus difficile En utilisant les idées du tri rapide, écrire un algorithme qui trouve le k-ième plus grand élément dans un tableau.

Quoi étudier ? – TP

L'utilité est plus cachée :

- Mettre en oeuvre les algorithmes
  - Solidifie et accélère l'apprentissage
  - Permet l'auto-correction
- Résoudre les exercices plus difficiles en pseudo-code
  - Transformer une description en pseudo-code
- Bien connaître les particularités de l'implémentation en C
  - Structures de données en C
  - Comment écrire une fonction de permutation
  - Mémoire pile vs tas

Quoi étudier ? – TP

Exercise : corriger la fonction

```
1 void permute(int * a, int * b) {
2     b = a;
3     a = b;
4 }
```

Structures de données

Structures de donnée

Façon de stocker et organiser des données, comportant :

- les données et les relation entre les données,
- des fonctions et opérateurs qui on applique aux données.

Objectifs :

- Accès efficace aux données
  - Un tas permet d'accéder rapidement à la valeur maximale
- Définir des méthodes standard pour l'accès aux données
  - La plupart des langages disposent d'un tableau dynamique avec des méthodes similaires
- Élément clé pour concevoir des algorithmes efficaces
  - Tri par tas, chemin le plus court, algorithmes de traversée
- Gérer de grandes quantités de données
  - Réseaux sociaux, systèmes de recommandation, nombreuses applications

Tableau dynamique

- Tableau qui permet des insertions et des suppressions
- Opérations efficaces pour les derniers éléments

Mise en œuvre

- Prévoir des cases vides pour permettre insertions.
- Redimensionnement si toutes les cases sont occupés
- Données :
  - un tableau t pour stocker les éléments
  - le nombre n d'éléments occupés
  - le nombre m d'éléments alloués
- Opérations : insertion, redimensionnement, ...

Tableau dynamique

On utilise une structure C pour grouper les données.

```c
// tableau_int.h
typedef struct {
    int* t;
    int n;
    int m;
} tbl_int;

tbl_int tbl_int_nouveau(int m);
void tbl_int_redim(tbl_int * tbl, int m);
void tbl_int_inserer(tbl_int * tbl, int el, int pos);
void tbl_int_trier(tbl_int* tbl);
int tbl_int_rechercher(tbl_int* tbl, int x);
int tbl_int_inverser(tbl_int* tbl);
```

Structures de données

Les listes chaînées

Les listes chaînées

Une liste chaînée est une structure de données :

- Pour stocker une séquence d'éléments (d'un même type)
- Organisé de manière non-contiguë dans la mémoire
- Permet d'insérer et supprimer des éléments efficacement
- Peut servir de base à d'autres structures
  - piles, files, ...
- Opérations principales : insertion, suppression, recherche

Structure d'une liste chaînée

Structure :

- un ensemble d'éléments (les maillons)
- chaque élément contient :
  - un valeur (contenu) et
  - un pointeur (chaînage) vers l'élément suivant
- Petites détails :
  - `NULL` : liste vide / sentinelle
  - `CONTENU`(m) : contenu, `SUIVANT`(m) : élément suivant de m.

Structure d'une liste chaînée

```
1  // liste_chainee.h
2  typedef struct _maillon_int {
3      int val;
4      struct _maillon_int* suiv;
5  } maillon_int, *liste_int;
6
7  maillon_int* nouveau_maillon_int(int val);
8  maillon_int* liste_int_inserer_pos(liste_int* lst, int
       i, int val);
9  maillon_int* liste_int_inserer_apres(maillon_int* m,
       int val);
10 void liste_int_supprimer_pos(liste_int* lst, int i);
11 void liste_int_supprimer_apres(maillon_int* m);
```

Structures de données

Complexité

Complexité

	Tableaux	Listes
Créer	$\Theta(1)$	$\Theta(1)$
Accès à la position i	$\Theta(1)$	$\Theta(i)$
Insertion à la position i	$\Theta(n-i)$*	$\Theta(i)$
(après un élément donné)		$\Theta(1)$
Suppression à la position i	$\Theta(n-i)$	$\Theta(i)$
(après un élément donné)		$\Theta(1)$
Recherche d'un élément	$\Theta(n)$	$\Theta(n)$
(dans une séquence trié)	$\Theta(\log n)$	$\Theta(n)$
Tri	$\Theta(n \log n)$	$\Theta(n \log n)$?

Tri par fusion

Tri par fusion

Méthode : Diviser pour régner

- Divise la liste en deux, trie chaque moitié, puis les fusionne.

Tri par fusion

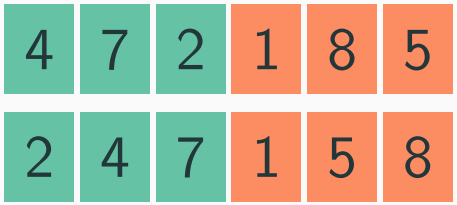
Méthode : Diviser pour régner

- Divise la liste en deux, trie chaque moitié, puis les fusionne.

Tri par fusion

Méthode : Diviser pour régner

- Divise la liste en deux, trie chaque moitié, puis les fusionne.



Tri par fusion

Méthode : Diviser pour régner

- Divise la liste en deux, trie chaque moitié, puis les fusionne.

- Sur les tableaux, n'est pas en place
- Mais sur les listes, en place et très pratique
- L'algorithme modifie la liste et ses chaînages pour la trier

Tri par fusion

Algorithme TriFusion

Données: Liste `lst`

```
1: if lst = NULL ∨ SUIVANT(lst) = NULL then return
2: p ← lst, q ← lst
3: while q ≠ NULL ∧ SUIVANT(q) ≠ NULL do
4:     p ← SUIVANT(p)
5:     q ← SUIVANT(SUIVANT(q))
6: end while
7: q ← SUIVANT(p), SUIVANT(p) ← NULL
8: TRIFUSION(lst)
9: TRIFUSION(q)
10: lst ← FUSIONNER(lst, q)
```

$\Theta(n)$

Trouver la moitié de la liste

q est le 2ème moitié

$\log_2 2 = 1$

Complexité ? $a = 2$, $b = 2$, n/b, $f(n) \in \Theta(n)$

$\Theta(n \log n)$

Tri par fusion – Fusionner

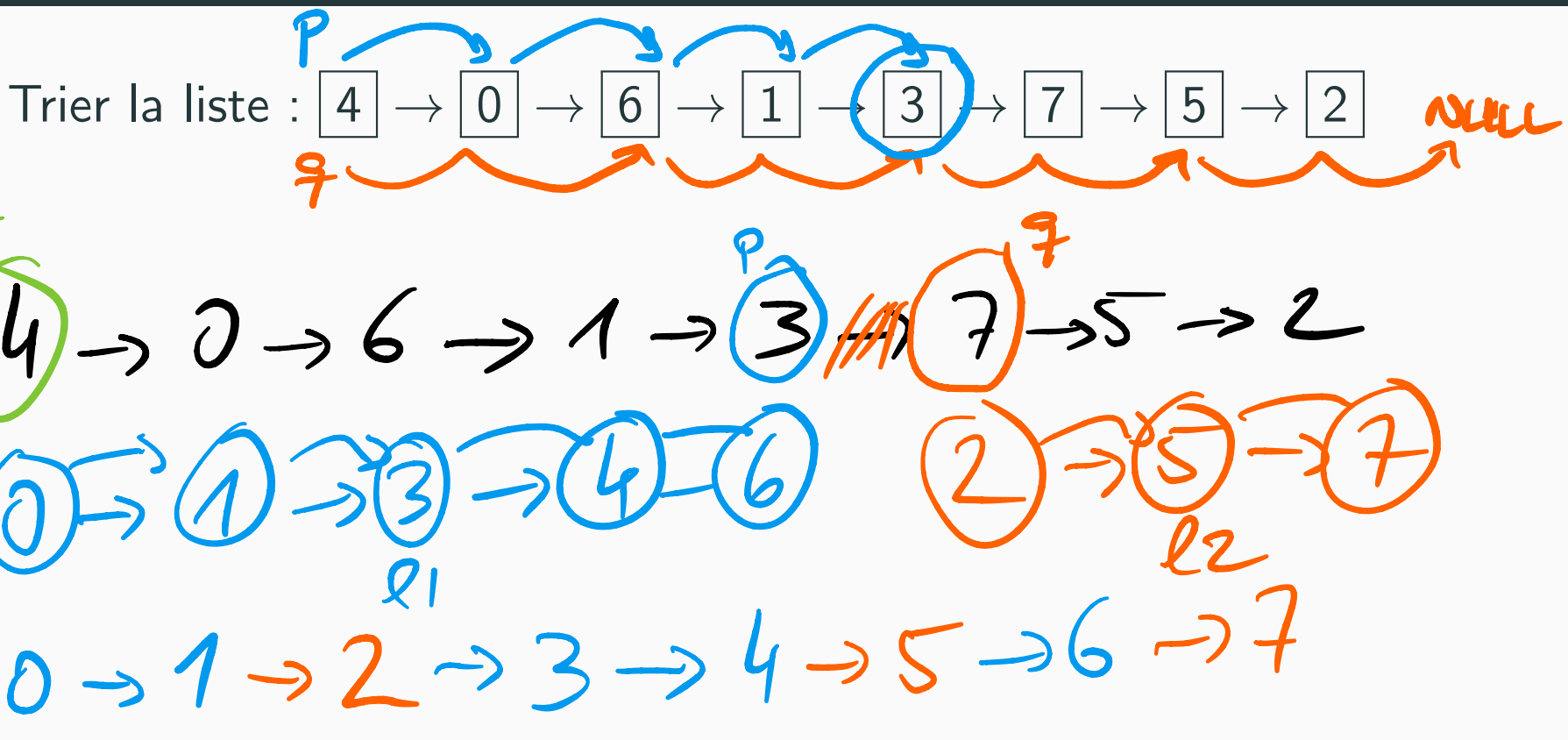
Algorithme Fusionner

Données: Listes l1 et l2

```
 1: lst ← NouveauMaillon(0), p ← lst
 2: while l1 ≠ NULL ∨ l2 ≠ NULL do
 3:     if l2 ≠ NULL ∨ CONTENU(l1) ≤ CONTENU(l2) then
 4:         SUIVANT(p) ← l1, p ← SUIVANT(p)
 5:         l1 ← SUIVANT(l1)
 6:     else
 7:         SUIVANT(p) ← l2, p ← SUIVANT(p)
 8:         l2 ← SUIVANT(l2)
 9:     end if
10: end while
11: SUIVANT(p) ← NULL
12: p ← SUIVANT(lst), Free(lst)
13: return p
```

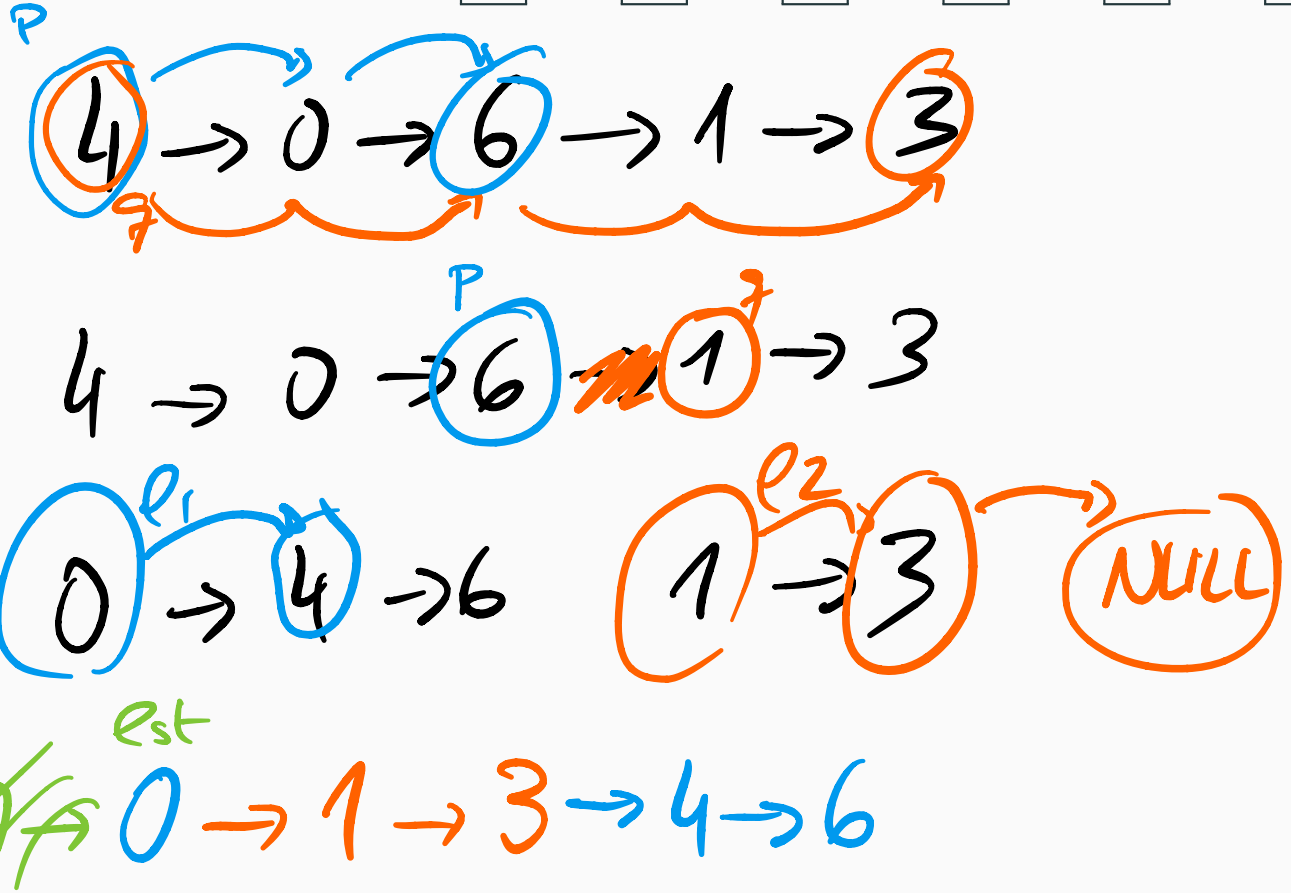
→ pointeur vers dernier element (du résultat)

Exécution d'un tri fusion



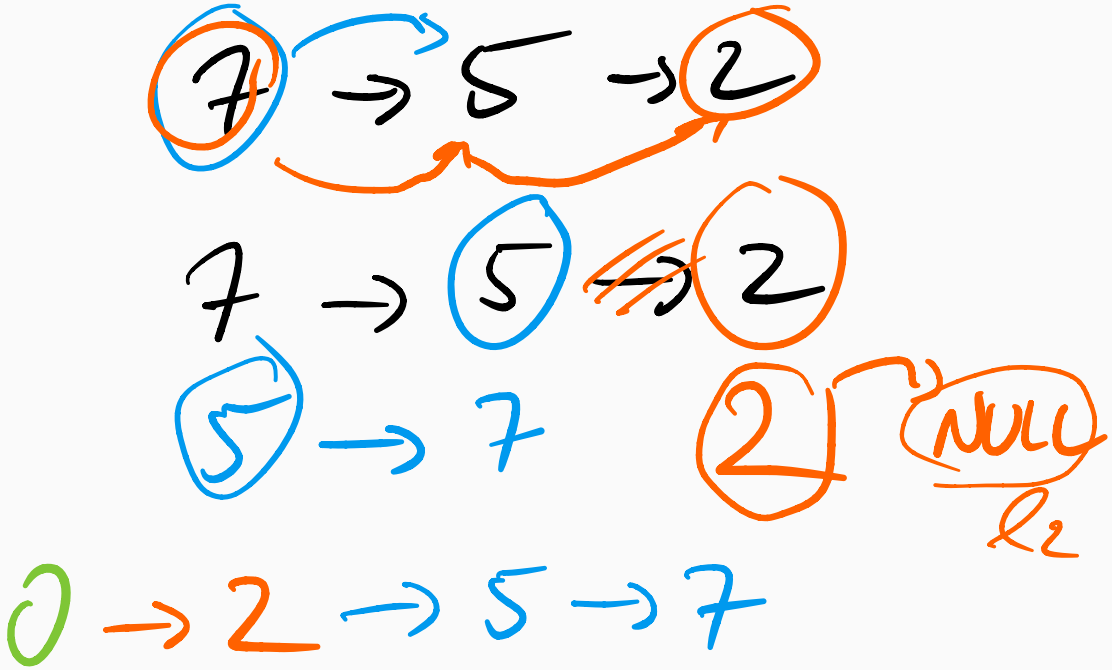
Exécution d'un tri fusion

Trier la liste : 4 → 0 → 6 → 1 → 3 → 7 → 5 → 2



Exécution d'un tri fusion

Trier la liste : $\boxed{4} \to \boxed{0} \to \boxed{6} \to \boxed{1} \to \boxed{3} \to \boxed{7} \to \boxed{5} \to \boxed{2}$



Complexité

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

Le tri fusion est stable et en place sur des listes.

Tableaux : complexité $\Theta(n \log n)$, pas en place ($\Theta(n)$ mémoire)

Tas

Tas

Tableau-tas

Tableau-tas

Tableau-tas : tableau avec des caractéristiques spécifiques

- Chaque élément a deux fils (ou moins)
  - Ce sont les deux éléments suivants disponibles,
  - Les fils de $t[0]$ sont $t[1]$ et $t[2]$,
  - Les fils de $t[1]$ sont $t[3]$ et $t[4]$,
  - Les fils de $t[2]$ sont $t[5]$ et $t[6]$,
  - Les fils de $t[3]$ sont $t[7]$ et $t[8]$,
  - Les fils de $t[i]$ sont $t[2i+1]$ et $t[2i+2]$.
- Chaque $t[i]$ doit être supérieur ou égal à ses fils

$t[i] \geq t[2i+1]$ $t[i] \geq t[2i+2]$, pour chaque i.

Caractéristiques d'un tableau-tas

Tableau-tas : tableau avec des caractéristiques spécifiques

- Chaque élément a deux fils (ou moins)
  - Ce sont les deux éléments suivants disponibles,
  - Les fils de $t[i]$ sont $t[2i+1]$ et $t[2i+2]$.
- Chaque $t[i]$ doit être supérieur ou égal à ses fils

Utilisation :

- Le maximum est toujours $t[0]$!
- On peut supprimer le dernier élément en temps $\Theta(1)$,
- On peut modifier l'élément $t[0]$ et rétablir le tas en $\Theta(\log n)$,
- On peut construire un tas à partir d'un tableau en $\Theta(n)$.

Construire un tas

0	1	2	3	4	5	6	7	
3	0	5	1	4	7	6	2	
3	0	5	1	4	7	6	**2**	$i = 7$
3	0	5	1	4	7	**6**	2	$i = 6$
3	0	5	1	4	**7**	6	2	$i = 5$
3	0	5	1	**4**	7	6	2	$i = 4$
3	0	5	**1**	4	7	6	2	$i = 3$

7,8

Construire un tas

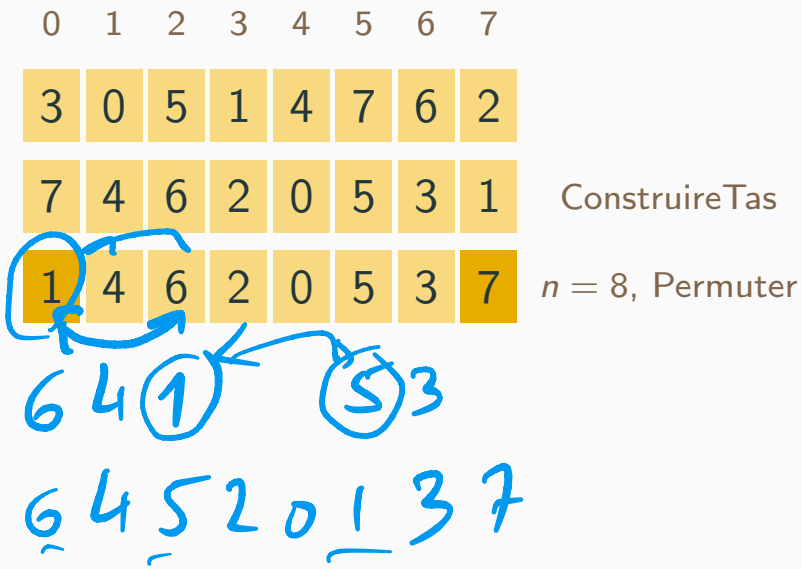
0	1	2	3	4	5	6	7	
3	0	5	1	4	7	6	2	
3	0	5	1	4	7	6	**2**	$i = 7$
3	0	5	1	4	7	**6**	2	$i = 6$
3	0	5	1	4	**7**	6	2	$i = 5$
3	0	5	1	**4**	7	6	2	$i = 4$
3	0	5	**1**	4	7	6	2	$i = 3$
3	0	5	2	4	7	6	**1**	
3	0	**5**	2	4	7	6	1	$i = 2$

Construire un tas

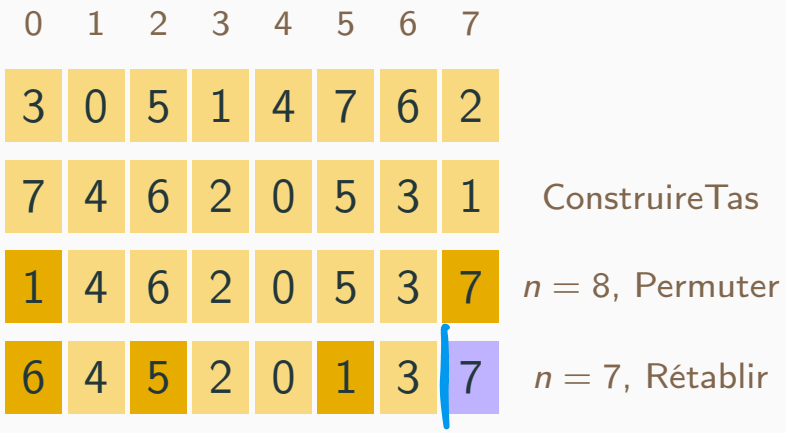
0	1	2	3	4	5	6	7	
3	0	5	1	4	7	6	2	
3	0	5	1	4	7	6	**2**	$i = 7$
3	0	5	1	4	7	**6**	2	$i = 6$
3	0	5	1	4	**7**	6	2	$i = 5$
3	0	5	1	**4**	7	6	2	$i = 4$
3	0	5	**1**	4	7	6	2	$i = 3$
3	0	5	2	4	7	6	**1**	
3	0	**5**	2	4	7	6	1	$i = 2$
3	0	7	2	4	**5**	6	1	

0	1	2	3	4	5	6	7	
3	0	7	2	4	5	6	1	
3	**0**	7	2	4	5	6	1	$i = 1$
3	4	7	2	**0**	5	6	1	
3	4	7	2	0	5	6	1	$i = 0$
7	4	**3**	2	0	5	6	1	
7	4	6	2	0	5	**3**	1	
7	4	6	2	0	5	3	1	

Exécution d'un tri par tas



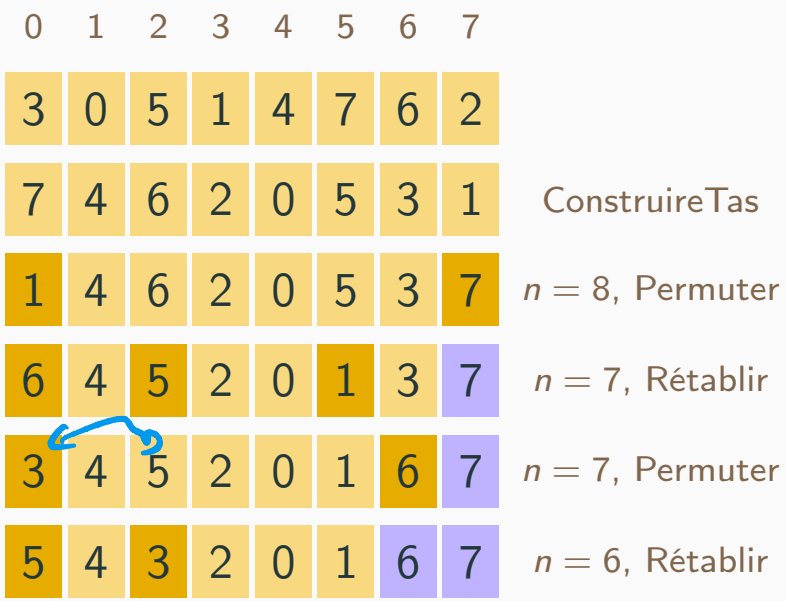
Exécution d'un tri par tas



Exécution d'un tri par tas

0	1	2	3	4	5	6	7	
3	0	5	1	4	7	6	2	
7	4	6	2	0	5	3	1	ConstruireTas
1	4	6	2	0	5	3	7	$n = 8$, Permuter
6	4	5	2	0	1	3	7	$n = 7$, Rétablir
3	4	5	2	0	1	6	7	$n = 7$, Permuter

Exécution d'un tri par tas



Exécution d'un tri par tas

0	1	2	3	4	5	6	7	
3	0	5	1	4	7	6	2	
7	4	6	2	0	5	3	1	ConstruireTas
1	4	6	2	0	5	3	7	$n = 8$, Permuter
6	4	5	2	0	1	3	7	$n = 7$, Rétablir
3	4	5	2	0	1	6	7	$n = 7$, Permuter
5	4	3	2	0	1	6	7	$n = 6$, Rétablir
1	4	3	2	0	5	6	7	$n = 6$, Permuter
4	2	3	1	0	5	6	7	$n = 5$, Rétablir
0	2	3	1	4	5	6	7	$n = 5$, Permuter

Exécution d'un tri par tas

0	1	2	3	4	5	6	7	
3	0	5	1	4	7	6	2	
7	4	6	2	0	5	3	1	ConstruireTas
1	4	6	2	0	5	3	7	$n = 8$, Permuter
6	4	5	2	0	1	3	7	$n = 7$, Rétablir
3	4	5	2	0	1	6	7	$n = 7$, Permuter
5	4	3	2	0	1	6	7	$n = 6$, Rétablir
1	4	3	2	0	5	6	7	$n = 6$, Permuter
4	2	3	1	0	5	6	7	$n = 5$, Rétablir
0	2	3	1	4	5	6	7	$n = 5$, Permuter
3	2	0	1	4	5	6	7	$n = 4$, Rétablir

0	1	2	3	4	5	6	7	
3	2	0	1	4	5	6	7	$n = 4$
1	2	0	3	4	5	6	7	$n = 4$, Permuter
2	1	0	3	4	5	6	7	$n = 3$, Rétablir
0	1	2	3	4	5	6	7	$n = 3$, Permuter
1	0	2	3	4	5	6	7	$n = 2$, Rétablir
0	1	2	3	4	5	6	7	$n = 2$, Permuter
0	1	2	3	4	5	6	7	$n = 2$, Rétablir

Exécution d'un tri par tas

2i+1 2i+2

0	1	2	3	4	5	6	7	
3	0	5	1	4	7	6	2	
7	4	6	2	0	5	3	1	ConstruireTas
1	4	6	2	0	5	3	7	$n = 8$, Permuter
6	4	5	2	0	1	3	7	$n = 7$, Rétablir
3	4	5	2	0	1	6	7	$n = 7$, Permuter
5	4	3	2	0	1	6	7	$n = 6$, Rétablir
1	4	3	2	0	5	6	7	$n = 6$, Permuter
4	2	3	1	0	5	6	7	$n = 5$, Rétablir
0	2	3	1	4	5	6	7	$n = 5$, Permuter
3	2	0	1	4	5	6	7	$n = 4$, Rétablir

0	1	2	3	4	5	6	7	
3	2	0	1	4	5	6	7	$n = 4$
1	2	0	3	4	5	6	7	$n = 4$, Permuter
2	1	0	3	4	5	6	7	$n = 3$, Rétablir
0	1	2	3	4	5	6	7	$n = 3$, Permuter
1	0	2	3	4	5	6	7	$n = 2$, Rétablir
0	1	2	3	4	5	6	7	$n = 2$, Permuter
0	1	2	3	4	5	6	7	$n = 2$, Rétablir
0	1	2	3	4	5	6	7	$n = 1$

Tas

Structure de données

File de priorité

Un tas est une implémentation classique d'une file de priorité

File de priorité : type abstrait qui permet les opérations :

- enfiler : insérer une valeur
- défiler : retourner (et supprimer) la plus grande valeur
- vérifier si la structure est vide

Utilisations :

- Chemin plus court
- Compression de données
- Gestion de réseaux

Structure d'un tas

Implémentation d'un tas :

- Généralement implémenté sous la forme d'un tas-tableau
  - En raison de son efficacité
- Cependant, autres structures sont possibles
  - Sa structure hiérarchique suggère un arbre

Structure d'un tas

Structure :

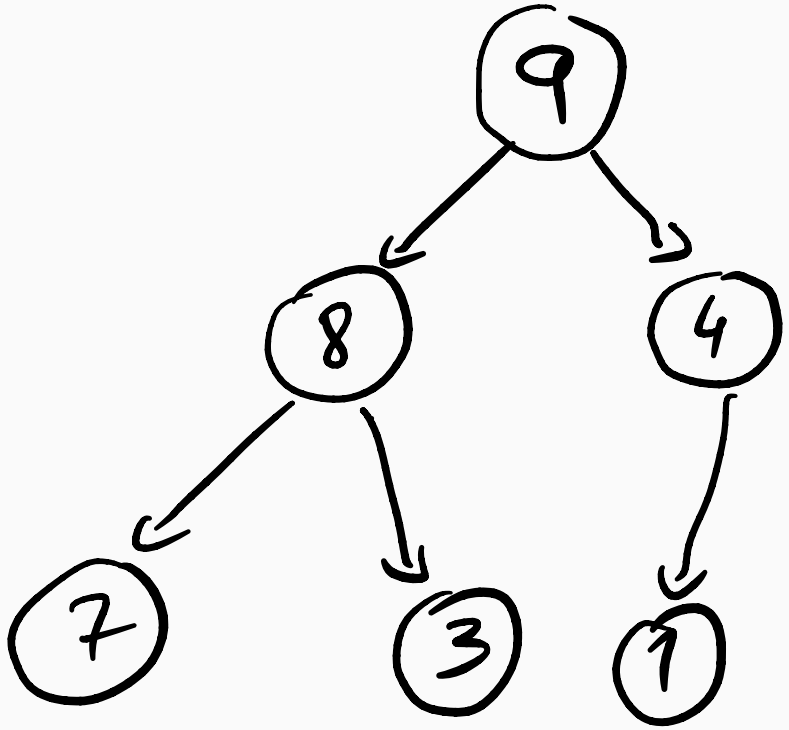
- Un arbre binaire presque complet
  - Arbre binaire : chaque élément a deux fils (ou moins)
  - Presque complet : tous niveaux remplis, dernier sur la gauche
- Éléments ordonnées selon la propriété de tas :
  - La valeur d'un élément est supérieur ou égal aux valeurs des fils
- Mise en œuvre :
  - arbre : chaque élément contient deux pointeurs
  - tableau : les fils de i sont aux positions $2i + 1$, $2i + 2$
    - Plus efficace !

Tas : tableau vs arbre

tableau-tas

Exemple : $[9, 8, 4, 7, 3, 1]$

arbre-tas

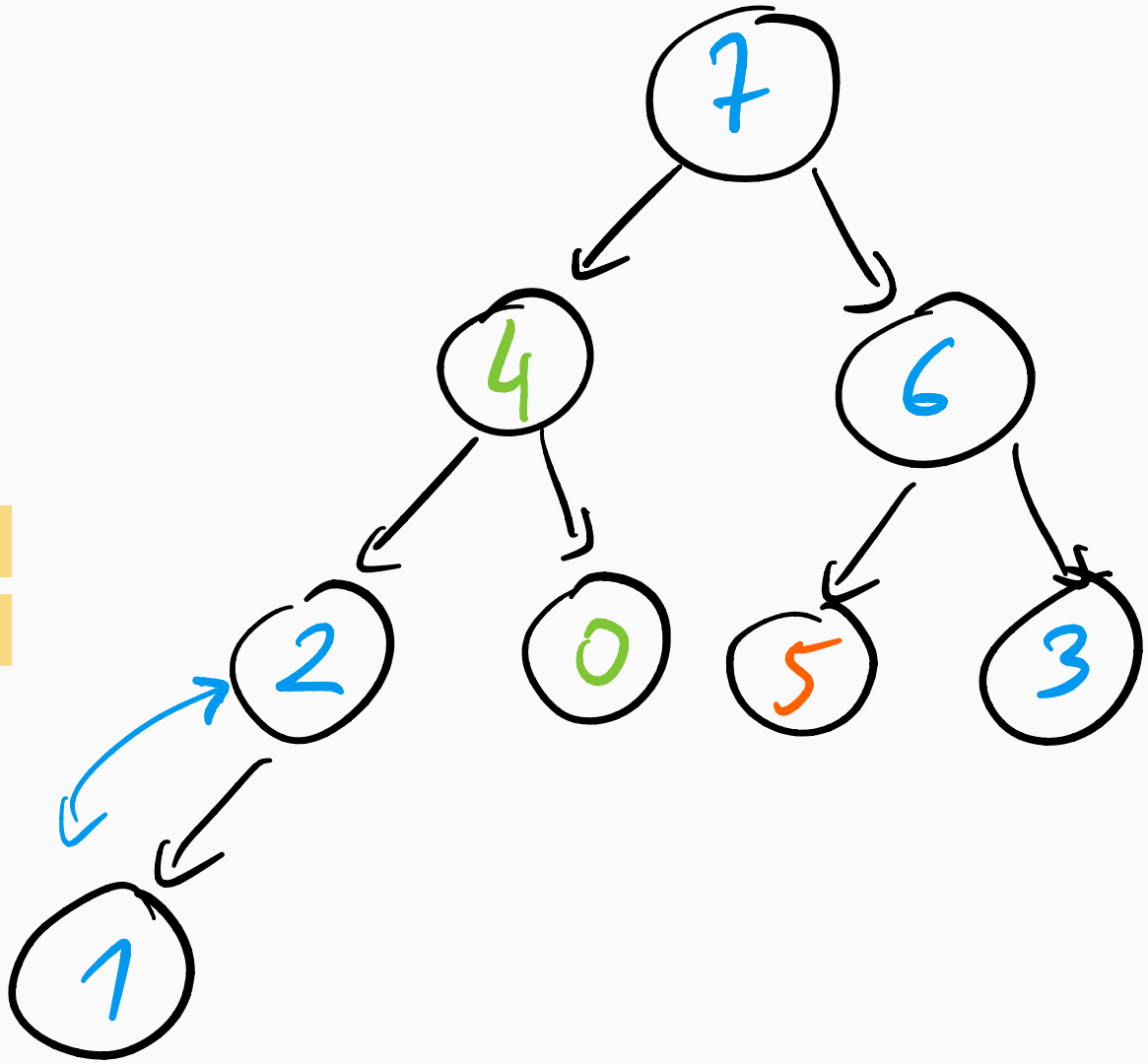


Tas : tableau vs arbre

Exemple : $[9, 8, 4, 7, 3, 1]$

Construire un arbre-tas

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



Construire un arbre-tas

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

Tas en arbre

```c
typedef struct _noeud_bin_int {
    int val;
    struct _noeud_bin_int* fils_gauche;
    struct _noeud_bin_int* fils_droite;
} noeud_bin_int, *arbre_bin_int;

void tas_arbre_construire(arbre_bin_int* tas);
noeud_bin_int* tas_arbre_inserer(arbre_bin_int* tas,
    int val);
int tas_arbre_extraire_max(arbre_bin_int* tas);
```

Tas en tableau

```c
#include <tableau_int.h>

void tas_tableau_construire(tableau_int* tas);
int tas_tableau_inserer(tableau_int* tas, int val);
int tas_tableau_extraire_max(tableau_int* tas);
```

Complexité

Complexité des opérations :

- Construire un tas : $\Theta(n)$
- Insérer un élément : $\Theta(\log n)$
- Extraire maximum : $\Theta(\log n)$

Révision

Révision

Algorithmes et Récursivité

Qu'est-ce qu'un Algorithme ?

Algorithme : une séquence d'instructions bien définies permettant de résoudre un problème ou d'effectuer une tâche.

Caractéristiques :

- Précis : chaque étape est claire et sans ambiguïté
- Fini : l'exécution se termine après un nombre fini d'étapes
- Un algorithme prend des données en entrée...
- ... et les traite pour produire un résultat en sortie

Généralement spécifié sous forme de pseudo-code

Les fonctions

Fonction : Unité de code

- qui sépare une portion de code pour le structurer
- qui peut être appelée pour exécuter son code
- qui accepte des paramètres et peut retourner une valeur

Lorsqu'une fonction est appelée :

- un espace est réservé dans la mémoire pour l'appel
- les paramètres sont copiés dans cet espace
- les variables créées dans la fonction sont également stockées dans cet espace
- cet espace disparaît (est libéré) lorsque la fonction se termine

Récursivité

Une fonction est récursive

- si elle contient des appels à elle-même
- généralement, avec des paramètres différents
- chaque appel contient son un espace de mémoire indépendant

Deux exemples :

- Factorielle
- Plus grand commun diviseur

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 $\rightarrow$ *memory leak*

Révision

Complexité

Complexité

Complexité asymptotique de temps dans le pire cas.

- asymptotique : vers l'infini
- temps : analyse du temps d'exécution d'un algorithme
- pire case : limite supérieur indépendant des données

Notation asymptotique : taux de croissance d'une fonction

- "Grand O" : $f \in O(g) \Leftrightarrow f \preccurlyeq g$
  (croissance de f inférieur ou égal à g)
- "Grand Omega" : $f \in O(g) \Leftrightarrow f \succcurlyeq g$
  (croissance de f supérieur ou égal à g)
- "Grand Theta" : $f \in O(g) \Leftrightarrow f \approx g$
  (croissance de f égal à g)

Complexité pratique

$\Theta(f)$ est la "complexité de f" : terme dominant sans constantes

- Exemple : $a_d n^d + a_{d-1} n^{d-1} + \ldots \in \Theta(n^d)$, $a_d > 0$
- $f \in O(g) \Leftrightarrow$ "$\Theta(f)$ inférieur ou égal à $\Theta(g)$"

On peut utiliser directement les comparaisons suivantes :

$$\Theta(1) \prec \Theta(\log n) \prec \Theta(\sqrt{n}) \prec \Theta(n) \prec \Theta(n \log n) \prec \Theta(n^2) \prec \Theta(2^n) \prec \Theta(n!)$$

En plus, si $a < b$, $\Theta((\log n)^{1000}) \prec \Theta(n^{0.001})$

- $\Theta((\log n)^a) \prec \Theta((\log n)^b) \prec \Theta(n^c)$ $(c > 0)$,
- $\Theta(n^a) \prec \Theta(n^b) \prec \Theta(c^n)$ $(c > 1)$,
- $\Theta(a^n) \prec \Theta(b^n) \prec \Theta(n!)$.

$\Theta(n^2) \prec \Theta(n^3)$

$\Theta(2^n) \prec \Theta(3^n)$

Diviser pour régner

Problème = sous-problèmes + opérations (préparation / fusion)

Complexité : donnée par la récurrence

$$C(n) = aC(n/b) + f(n),$$

avec $a \geq 1$ et $b > 1$.

Notations :

- n : taille du problème
- a : nombres de sous-problèmes
- n/b : taille des sous-problèmes
- $f(n)$: coût de gestion en dehors des appels récursifs

Diviser pour régner – Master Theorem

Theorème (Master Theorem)

Si $C(n) = aC(n/b) + f(n)$,

- *si* $f(n) = O(n^c)$, *pour* $c < \log_b a$, *alors* $C(n) = \Theta(n^{\log_b a})$,
- *si* $f(n) = \Theta(n^{\log_b a})$, *alors* $C(n) = \Theta(n^{\log_b a} \log n)$,
- *si* $f(n) = \Omega(n^c)$, *pour* $c > \log_b a$, *alors*, $C(n) = \Theta(f(n))$
  en supposant que $\alpha < 1$ *existe tel que* $a \cdot f(n/b) \leq \alpha \cdot f(n)$, $n > n_0$

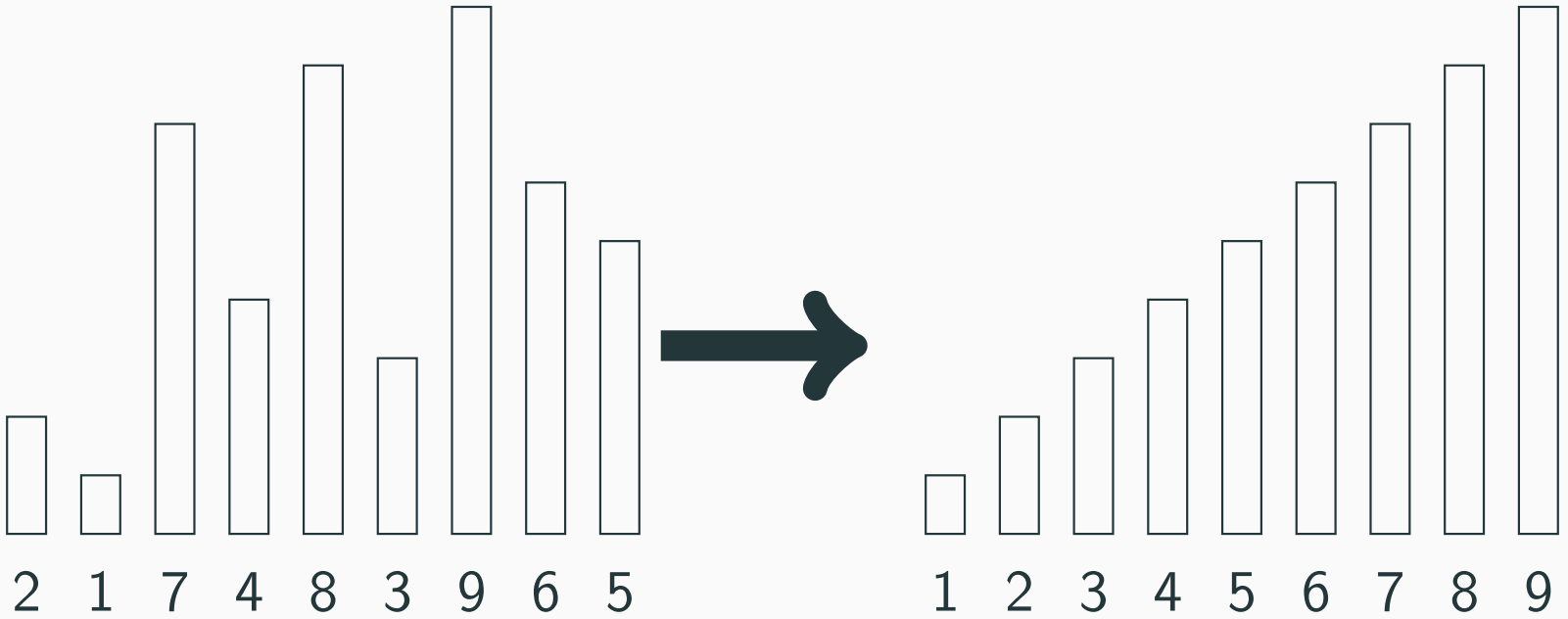
Tri par fusion : $a=2$, $b=2$ $f(n) \in \Theta(n)$
$\rightarrow C(n) \in \Theta(n \cdot \log n)$

Révision

Algorithmes de tri

Problème de tri

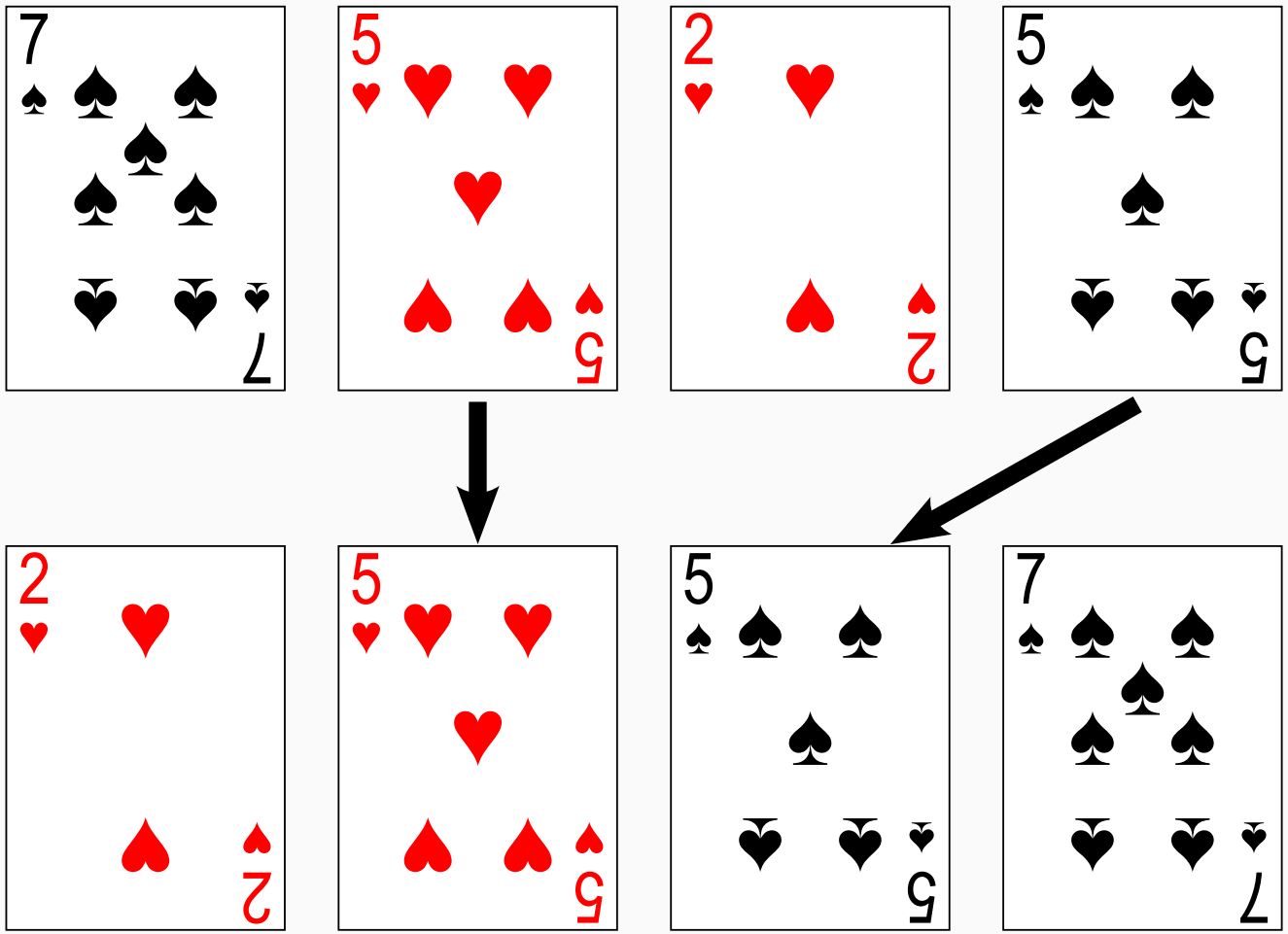
Trier : réarranger les éléments de sorte que $t[i] \leq t[j]$ si $i < j$
(mais en fait, il suffit de vérifier que $t[i-1] \leq t[i]$)



Algorithme de tri

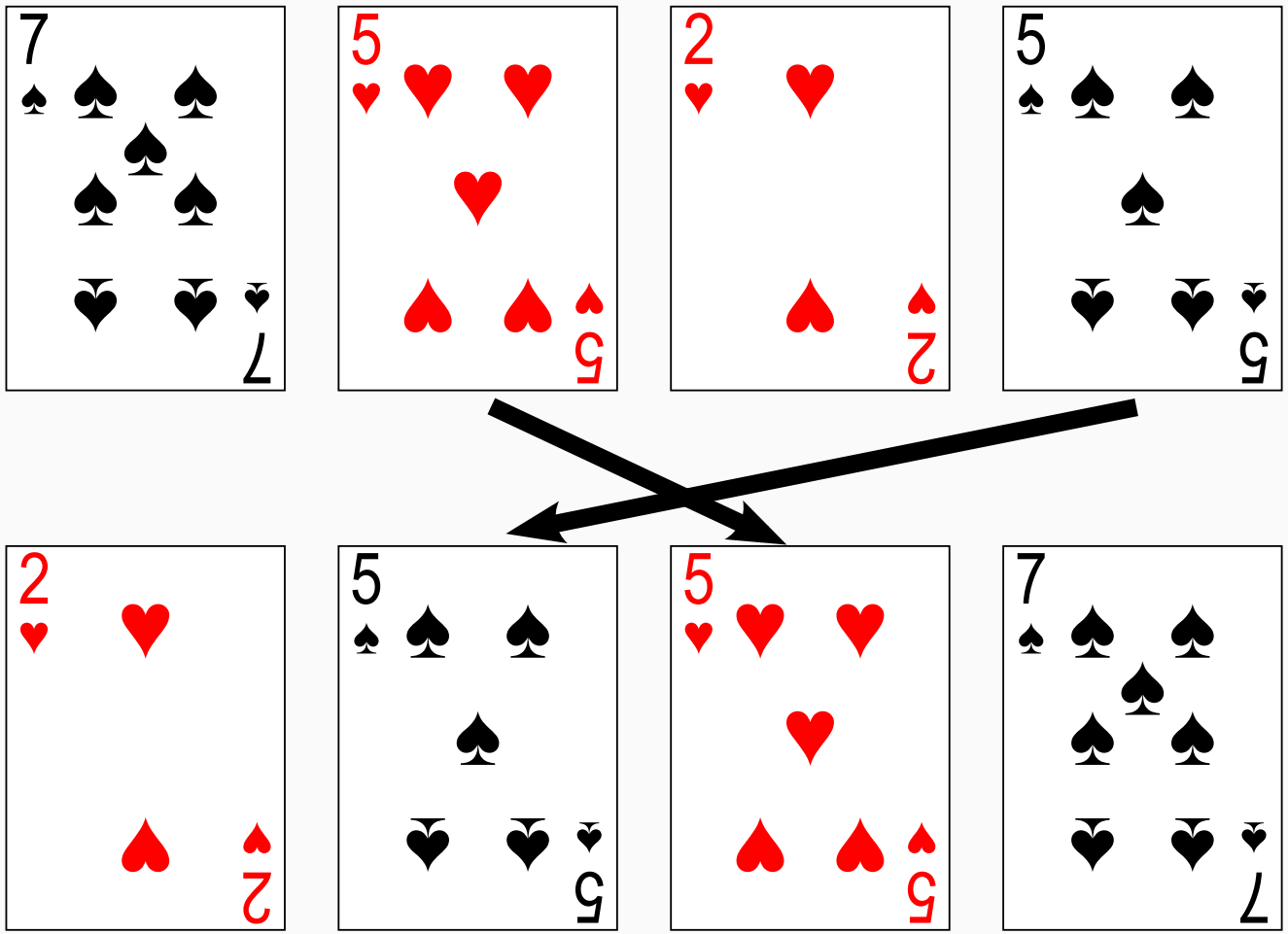
- Prennent un tableau en entrée, retournent la liste (ou rien)
- Tri en place : modifie directement le tableau d'entrée
  - Économe en mémoire, car il n'a alloue pas une nouvelle liste
  - Pas nécessaire de retourner la liste
- Tri stable : préserve l'ordre initial des éléments que sont considérés comme 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

Tri stable



Stable

Tri stable



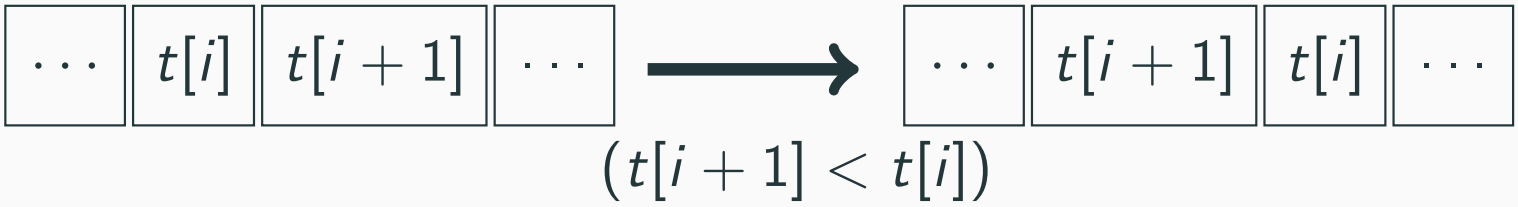
Pas stable

Révision

Algorithmes de tri simples

Tri à bulles

Une opération simple : permutation d'éléments adjacents



Algorithme TriBulleBase

```
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
```

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

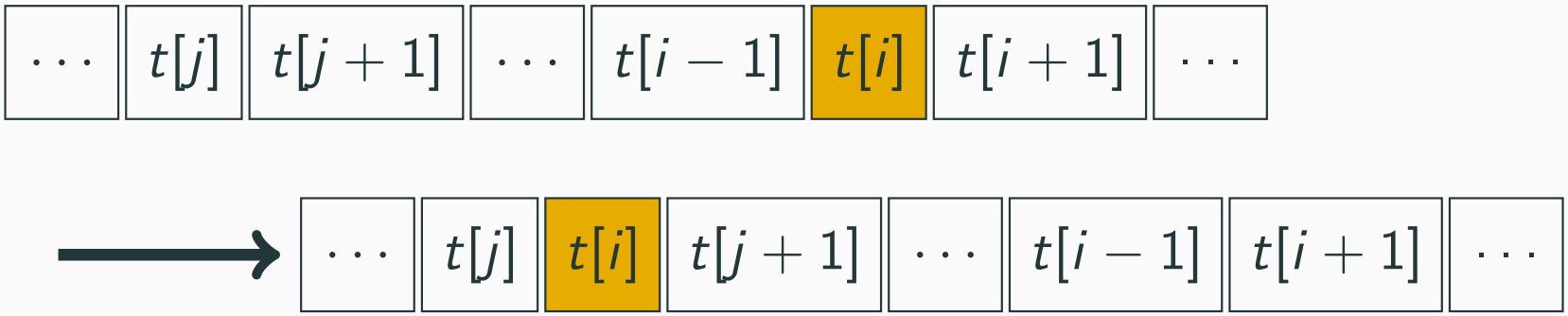
($t[i]$ maximal)

Algorithme TriSelection

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

Tri par insertion

Opération : insérer l'élément $t[i]$ dans $t[0 : i-1]$



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

Tri par insertion

Algorithme TriInsertion

1: **for** $1 \leq i < n$ **do**
2: $\quad x \leftarrow t[i]$
3: $\quad p \leftarrow \texttt{Recherche}(t, i, x)$
4: $\quad \texttt{Decalage}(t, p, i-1)$
5: $\quad t[p] \leftarrow x$
6: **end for**

Complexité

	Meilleur cas	Moyen cas	Pire cas
Tri bulles*	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Tri sélection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Tri insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$

Révision

Algorithmes de tri avancés

Tri rapide

Méthode : Diviser pour régner

Idée : trier récursivement les éléments inférieurs et supérieurs

- Un élément du tableau est choisi (le pivot),
- Les éléments inférieurs au pivot sont déplacés vers sa gauche,
- Et les éléments supérieurs à sa droite,

  - Ensuite, les deux parties sont triées récursivement.

Très populaire, car il fonctionne bien en pratique

Tri rapide

Algorithme TriRapide

Données: tableau t de type s, début d et fin f

1: **if** $f - d > 1$ **then**
2: $p = \textsc{Partition}(t, d, f)$
// Sélectionne le pivot, regroupe éléments inférieurs / supérieurs
3: $\textsc{TriRapide}(t, d, p)$
4: $\textsc{TriRapide}(t, p + 1, f)$
5: **end if**

Le tri initial est appelé avec $\textsc{TriRapide}(t, 0, n)$

Partition (annexe tri rapide)

Algorithme Partition

Données: tableau t de type s, début d et fin f

Résultat: position du pivot p

1: $p \leftarrow d$, $d \leftarrow d + 1$
2: **while** $d < f$ **do**
3: **if** $t[d] \leq t[p]$ **then**
4: $d \leftarrow d + 1$
5: **else**
6: $\textsc{Permute}(t[d], t[f - 1])$
7: $f \leftarrow f - 1$
8: **end if**
9: **end while**
10: $\textsc{Permute}(t[p], t[d - 1])$
11: **return** $d - 1$

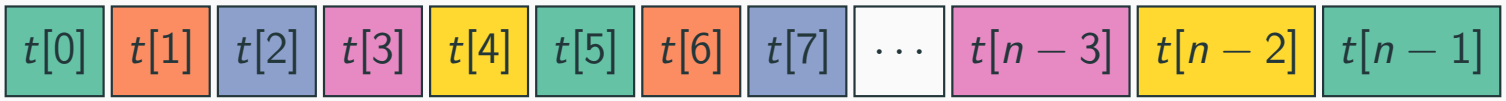
Tri Shell

Optimisation du tri par insertion

- Permet l'échange d'éléments très éloignés les uns des autres

Idéé : trier d'abord les séquences de chaque h-ème élément

- Trier chaque couleur (par insertion), $h = 5$



- Permet aux éléments de se déplacer sur de longues distances
- Laisse moins de travail au tri final

Effectue plusieurs tris sur une séquence décroissante de h

- Termine toujours par un tri $h = 1$ (tri par insertion)

Tri Shell

Algorithme TriShell

Données: tableau t de type s, taille du tableau n

1: $h \leftarrow int(n/2)$
2: **while** $h > 0$ **do**
3: **for** $0 \leq i < h$ **do**
4: TriSousTableau(t, n, i, h)
5: **end for**
6: $h \leftarrow int(h/2)$
7: **end while**

Tri Shell (TriSousTableau)

Algorithme TriSousTableau

Données: tableau t de type s, entiers n, i, h

```
1:  i ← i + h
2:  while i < n do
3:      x ← t[i]
4:      j ← i
5:      while j − h ≥ 0 ∧ t[j − h] > x do
6:          t[j] ← t[j − h]
7:          j ← j − h
8:      end while
9:      t[j] ← x
10:     i ← i + h
11: end while
```

Tri par tas

Idée :

- Donner au tableau une structure spécifique
- Ce qui permet de extraire l'élément maximal rapidement
- Un peu comme une amélioration du tri par sélection
    - Mais beaucoup plus rapide

Tri par tas

Algorithme TriTas

Données: tableau t de type s, taille du tableau n

```
1: ConstruireTas(t, n)
2: while n > 1 do
3:     Permuter(t[0], t[n − 1])
4:     n ← n − 1
5:     RetablirTas(t, n, 0)
6: end while
```

Complexité

	Meilleur cas	Moyen cas	Pire cas
Tri rapide	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$
Tri Shell	$\Theta(n \log n)$	?	$\Theta(n^2)$
(Pratt)	$\Theta(n \log^2 n)$	$\Theta(n \log^2 n)$	$\Theta(n \log^2 n)$
Tri par tas	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Tri fusion	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$