ALGORITHMIQUE ET STRUCTURES DE DONNÉES

2. Structure de données linéaires et tris

Yoann Coudert--Osmont

27 Janvier 2026

Définitions

Type abstrait

Un type abstrait est un modèle mathématique d'un ensemble de données pour lequel est spécifié un ensemble d'opérations pouvant être effectuées sur les données.

Définitions

Type abstrait

Un type abstrait est un modèle mathématique d'un ensemble de données pour lequel est spécifié un ensemble d'opérations pouvant être effectuées sur les données.

Structure de données

Une structure de données est une structure qui permet d'implémenter un type abstrait, rendant les opérations du type abstrait plus ou moins efficaces, le tout en utilisant peu d'espace mémoire.

Exemple

Une **file de priorité** est un **type abstrait** maintenant un ensemble de n valeurs comparables entre elles avec les opérations suivantes possibles :

- **Requête :** Obtenir le plus grand élément de l'ensemble.
- **Mise à jour :**
  - Ajouter une nouvelle valeur à l'ensemble.
  - Retirer le plus grand élément de l'ensemble.

Exemple

Une **file de priorité** est un **type abstrait** maintenant un ensemble de n valeurs comparables entre elles avec les opérations suivantes possibles :

- **Requête :** Obtenir le plus grand élément de l'ensemble.
- **Mise à jour :**
  - Ajouter une nouvelle valeur à l'ensemble.
  - Retirer le plus grand élément de l'ensemble.

Un **tas binaire** est une **structure de donnée** permettant d'implémenter une file de priorité avec une complexité spatiale en $\Theta(n)$, et les complexités temporelles suivantes :

- **Requête :** Obtenir le plus grand élément de l'ensemble en $\boldsymbol{\Theta(1)}$.
- **Mise à jour :**
  - Ajouter une nouvelle valeur à l'ensemble en $\boldsymbol{\Theta(\log(n))}$.
  - Retirer le plus grand élément de l'ensemble en $\boldsymbol{\Theta(\log(n))}$.

Plan

1. Structure de données linéaires
   - Tableaux
   - Listes chaînées
2. Algorithmes de tri
   - Tri par insertion
   - Tri rapide
   - Tri par fusion
   - Complexité optimale

Structure de données linéaires

Définition

Structure de données linéaire

Une structure de données linéaire organise les données selon une séquence linéaire. On peut alors numéroter les données ; chaque élément possède un rang.

Définition

Structure de données linéaire

Une structure de données linéaire organise les données selon une séquence linéaire. On peut alors numéroter les données ; chaque élément possède un rang.

On distingue principalement deux structures de données linéaires :

- **Les tableaux** : Les éléments se trouvent les uns à la suite des autres dans la mémoire. Il est ainsi possible d'accéder en $\mathcal{O}(1)$ à n'importe quel élément.

Définition

Structure de données linéaire

Une structure de données linéaire organise les données selon une séquence linéaire. On peut alors numéroter les données ; chaque élément possède un rang.

On distingue principalement deux structures de données linéaires :

- **Les tableaux** : Les éléments se trouvent les uns à la suite des autres dans la mémoire. Il est ainsi possible d'accéder en $\mathcal{O}(1)$ à n'importe quel élément.
- **Les listes chaînées** : Les éléments sont dispersés dans la mémoire. Chaque élément pointe vers le prochain élément dans la structure. L'accès au k-ème élément se fait alors en $\mathcal{O}(k)$.

Structure de données linéaires

Tableaux

Définition

Tableau

Un tableau T est une structure permettant de stocker des valeurs (toutes du même type) $T[i]$ repérés par leurs indices i. Le nombre d'élément n d'un tableau est fixe. Chaque élément peut être accédé et modifier en $\mathcal{O}(1)$.

Comme tous éléments de T ont la même taille s, l'élément $T[i]$ se trouve à l'adresse : $[\text{adresse}(T) + i \cdot s]$.

Dans les langages de programmation

Voici différente façon de créer un tableau d'entiers T de taille n dans différents langages :

Python

Il est possible d'utiliser la classe `array` de `numpy`.

```
import numpy as np
T = np.empty(n, dtype=int)
```

T est alors un tableau de taille n initialisé avec des valeurs aléatoires (celles déjà présentes dans la mémoire de l'ordinateur).

Dans les langages de programmation

Voici différente façon de créer un tableau d'entiers T de taille n dans différents langages :

C++

Il est possible d'utiliser un pointeur `int*`.

```
int *T = new int[n];
```

T est alors un tableau de taille n initialisé avec des valeurs aléatoires (celles déjà présentes dans la mémoire de l'ordinateur). T est **alloué sur le tas**.
Pour **allouer sur la pile**, si n est connu au moment de la compilation, on peut utiliser la classe `std::array`.

```
#include <array>
std::array<int, n> T;
```

Tableaux dynamiques

Tableau dynamique

Un tableau dynamique est une tableau dont la taille est variable. On peut ajouter de nouveaux éléments à ce tableau.

Tableaux dynamiques

Tableau dynamique

Un tableau dynamique est une tableau dont la taille est variable. On peut ajouter de nouveaux éléments à ce tableau.

Lorsque l'on crée un tableau, on demande au système d'exploitation de nous allouer une zone de la mémoire. Mais il est possible que d'autre données soient stockées immédiatement après le tableau.

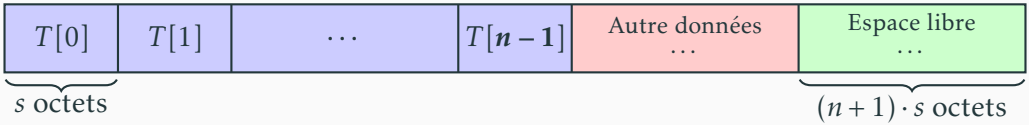
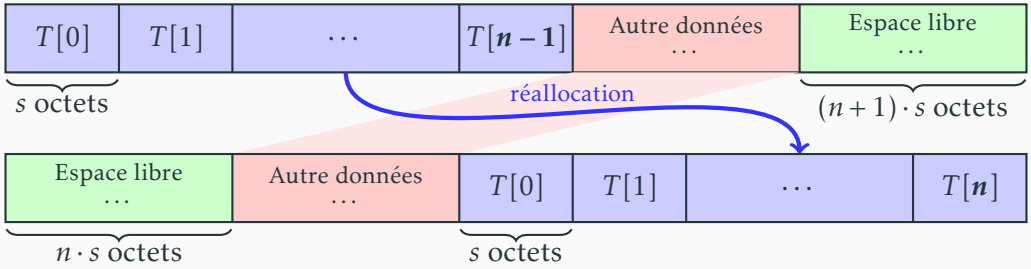


Tableau dynamique

Un tableau dynamique est une tableau dont la taille est variable. On peut ajouter de nouveaux éléments à ce tableau.

Lorsque l'on crée un tableau, on demande au système d'exploitation de nous allouer une zone de la mémoire. Mais il est possible que d'autre données soient stockées immédiatement après le tableau.



La réallocation du tableau nécessite de le recopier entièrement avec une complexité temporelle en $\Theta(n)$.

Question : Combien de copies d'éléments sont effectuées lorsque l'on construit un tableau dynamique de taille n en ajouter des éléments un à un à un tableau initialement vide ?

Tableaux dynamiques

- T de taille 0 : 0 copie(s) à effectuer pour ajouter un élément.

- T de taille 0 : 0 copie(s) à effectuer pour ajouter un élément.
- T de taille 1 : 1 copie(s) à effectuer pour ajouter un élément.

- T de taille 0 : 0 copie(s) à effectuer pour ajouter un élément.
- T de taille 1 : 1 copie(s) à effectuer pour ajouter un élément.
- T de taille 2 : 2 copie(s) à effectuer pour ajouter un élément.

Tableaux dynamiques

- T de taille 0 : 0 copie(s) à effectuer pour ajouter un élément.
- T de taille 1 : 1 copie(s) à effectuer pour ajouter un élément.
- T de taille 2 : 2 copie(s) à effectuer pour ajouter un élément.
- ...
- T de taille $n-1$: $n-1$ copie(s) à effectuer pour ajouter un élément.
- T est de taille n !

Tableaux dynamiques

- T de taille 0 : 0 copie(s) à effectuer pour ajouter un élément.
- T de taille 1 : 1 copie(s) à effectuer pour ajouter un élément.
- T de taille 2 : 2 copie(s) à effectuer pour ajouter un élément.
- ...
- T de taille $n-1$: $n-1$ copie(s) à effectuer pour ajouter un élément.
- T est de taille n !

$$C(n) = 0 + 1 + 2 + \cdots + (n-1)$$

Tableaux dynamiques

- T de taille 0 : 0 copie(s) à effectuer pour ajouter un élément.
- T de taille 1 : 1 copie(s) à effectuer pour ajouter un élément.
- T de taille 2 : 2 copie(s) à effectuer pour ajouter un élément.
- ...
- T de taille $n-1$: $n-1$ copie(s) à effectuer pour ajouter un élément.
- T est de taille n !

$$C(n) = 0 + 1 + 2 + \cdots + (n-1) = n(n-1)/2 = \boldsymbol{\Theta\left(n^2\right)}$$

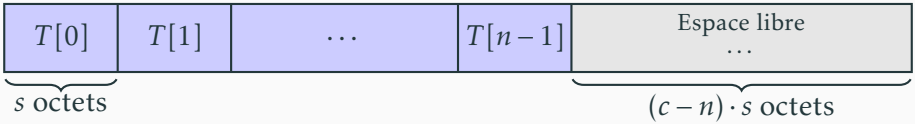
- T de taille 0 : 0 copie(s) à effectuer pour ajouter un élément.
- T de taille 1 : 1 copie(s) à effectuer pour ajouter un élément.
- T de taille 2 : 2 copie(s) à effectuer pour ajouter un élément.
- ...
- T de taille $n-1$: $n-1$ copie(s) à effectuer pour ajouter un élément.
- T est de taille n !

$$C(n) = 0 + 1 + 2 + \cdots + (n-1) = n(n-1)/2 = \boldsymbol{\Theta\left(n^2\right)}$$

Question : Peut-on mieux faire ?

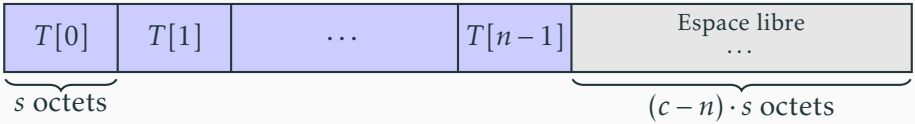
Tableaux dynamiques

On peut allouer une capacité c plus grande que la taille n de T.

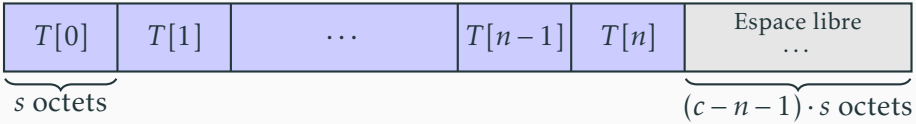


Tableaux dynamiques

On peut allouer une capacité c plus grande que la taille n de T.

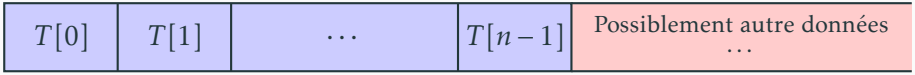


Si $c > n$, ajouter un élément ne nécessite pas de réallocation et donc pas de copie. L'opération se fait en $\Theta(1)$.

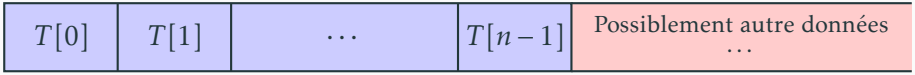


Tableaux dynamiques

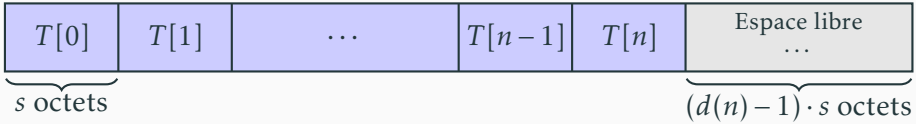
Si $c = n$.



Si $c = n$.



Alors il faut effectuer une réallocation entraînant n copies.
La nouvelle capacité devient alors $[c \leftarrow n + d(n)]$ où $d(n) \geqslant 1$.



Tableaux dynamiques

Supposons que l'on ajoute une capacité fixe $\boldsymbol{d(n) = b}$ à chaque réallocation, avec $b \geqslant 1$. On suppose que l'on construit un tableau de taille $n = k \cdot b$ en partant d'un tableau vide et en ajoutant les éléments un par un.

Supposons que l'on ajoute une capacité fixe $\boldsymbol{d(n) = b}$ à chaque réallocation, avec $b \geqslant 1$. On suppose que l'on construit un tableau de taille $n = k \cdot b$ en partant d'un tableau vide et en ajoutant les éléments un par un.

Étape	Capacité	Nouvelle capacité	#copies
1	0	b	0
2	b	$2b$	b
3	$2b$	$3b$	$2b$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	$(k-1)b$	kb	$(k-1)b$

Tableaux dynamiques

Supposons que l'on ajoute une capacité fixe $\boldsymbol{d(n) = b}$ à chaque réallocation, avec $b \geqslant 1$. On suppose que l'on construit un tableau de taille $n = k \cdot b$ en partant d'un tableau vide et en ajoutant les éléments un par un.

Étape	Capacité	Nouvelle capacité	#copies
1	0	b	0
2	b	$2b$	b
3	$2b$	$3b$	$2b$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	$(k-1)b$	kb	$(k-1)b$

$$C(n) = (0 + 1 + 2 + \cdot + (k-1)) \cdot b = k(k-1)b/2$$

Tableaux dynamiques

Supposons que l'on ajoute une capacité fixe $\boldsymbol{d(n) = b}$ à chaque réallocation, avec $b \geqslant 1$. On suppose que l'on construit un tableau de taille $n = k \cdot b$ en partant d'un tableau vide et en ajoutant les éléments un par un.

Étape	Capacité	Nouvelle capacité	#copies
1	0	b	0
2	b	$2b$	b
3	$2b$	$3b$	$2b$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	$(k-1)b$	kb	$(k-1)b$

$$C(n) = (0 + 1 + 2 + \cdot + (k-1)) \cdot b = k(k-1)b/2 = n(n-b)/(2b) = \boldsymbol{\Theta(n^2)}$$

Tableaux dynamiques

Supposons désormais que l'on ajoute une capacité $\boldsymbol{d(n) = (a-1) \cdot n}$ à chaque réallocation, avec $a > 1$. On suppose que l'on construit un tableau de taille $n = a^k$ en partant d'un tableau à un élément et en ajoutant les éléments un par un.

Tableaux dynamiques

Supposons désormais que l'on ajoute une capacité $\boldsymbol{d(n) = (a-1) \cdot n}$ à chaque réallocation, avec $a > 1$. On suppose que l'on construit un tableau de taille $n = a^k$ en partant d'un tableau à un élément et en ajoutant les éléments un par un.

Étape	Capacité	Nouvelle capacité	#copies
1	1	a	1
2	a	a^2	a
3	a^2	a^3	a^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	a^{k-1}	a^k	a^{k-1}

Tableaux dynamiques

Supposons désormais que l'on ajoute une capacité $\boldsymbol{d(n) = (a-1) \cdot n}$ à chaque réallocation, avec $a > 1$. On suppose que l'on construit un tableau de taille $n = a^k$ en partant d'un tableau à un élément et en ajoutant les éléments un par un.

Étape	Capacité	Nouvelle capacité	#copies
1	1	a	1
2	a	a^2	a
3	a^2	a^3	a^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	a^{k-1}	a^k	a^{k-1}

$$C(n) = 1 + a + a^2 + \cdot + a^{k-1}$$

Tableaux dynamiques

Supposons désormais que l'on ajoute une capacité $\boldsymbol{d(n) = (a-1) \cdot n}$ à chaque réallocation, avec $a > 1$. On suppose que l'on construit un tableau de taille $n = a^k$ en partant d'un tableau à un élément et en ajoutant les éléments un par un.

Étape	Capacité	Nouvelle capacité	#copies
1	1	a	1
2	a	a^2	a
3	a^2	a^3	a^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	a^{k-1}	a^k	a^{k-1}

$$C(n) = 1 + a + a^2 + \cdot + a^{k-1} = \frac{a^k - 1}{a - 1} = (n-1)/(a-1) = \boldsymbol{\Theta(n)}$$

$$C(n) = (n-1)/(a-1) = \boldsymbol{\Theta(n)}$$

Remarque : Plus a est grand, moins de copies sont effectuées.
En revanche, plus a est grand, plus de la mémoire est gâchée.
Après une réallocation, la capacité vaut $c = a \cdot n$.
En pratique, a est souvent compris entre 1.1 et 2.

Complexité amortie

L'ajout d'un élément au tableau dynamique peut parfois nécessité n copies si une réallocation a lieu. Sinon, le plus souvent, aucune copie n'est effectuée.
La complexité dans le pire cas est donc un $\Theta(n)$.
En revanche, en moyenne, l'insertion nécessite $C(n)/n \sim 1/(a-1) = \Theta(1)$ copies.

Complexité amortie

L'ajout d'un élément au tableau dynamique peut parfois nécessité n copies si une réallocation a lieu. Sinon, le plus souvent, aucune copie n'est effectuée.
La complexité dans le pire cas est donc un $\Theta(n)$.
En revanche, en moyenne, l'insertion nécessite $C(n)/n \sim 1/(a-1) = \Theta(1)$ copies.

Complexité amortie

La complexité amortie mesure la complexité moyenne d'une suite d'opérations effectuées sur une même structure de données.

Dans les langages de programmation

Voici différente façon de créer un tableau dynamique d'entiers T de taille n dans différents langages :

Python

Il est possible d'utiliser la structure `list`.

⚠ malgré le nom, ce n'est pas une liste chaînée.

```
>>> T = [0]*n
```

T est alors un tableau dynamique de taille n initialisé avec des 0. Le coût amortie de l'insertion est en $\Theta(1)$. L'insertion d'un élément x se fait via :

```
>>> T.append(x)
```

Dans les langages de programmation

Voici différente façon de créer un tableau dynamique d'entiers T de taille n dans différents langages :

C++

Il est possible d'utiliser la structure `std::vector`.

```
#include <vector>
std::vector<int> T(n);
```

T est alors un tableau dynamique de taille n initialisé avec des 0. Le coût amortie de l'insertion est en $\Theta(1)$. L'insertion d'un élément x se fait via :

```
T.push_back(x);
```

Structure de données linéaires

Listes chaînées

Définition

Liste chaînée

Une liste chaînée L est une structure permettant de stocker une séquence de n valeurs $L_0, L_1, \ldots, L_{n-1}$ à l'aide de cellules. Chaque cellule contient une valeur de la séquence ainsi qu'un pointeur vers la cellule suivante formant ainsi une chaîne de cellules. Contrairement au tableau qui permet l'accès direct à n'importe quelle valeur en $\Theta(1)$, la liste requiert de parcourir les cellules les unes après les autres.

La dernière cellule ne pointe vers rien.

Dans les langages de programmation

Les listes sont rarement implémentées dans les librairies standards des langages de programmation.

C++

Il est possible de définir la structure suivante :

```
struct Cellule {
    int valeur;
    Cellule *suivant;
};
using List = Cellule*;
```

Dans les langages de programmation

L'accès à L_k peut être implémenté de la manière suivante :

```
#include <cassert>
int& acces(List &L, int k) {
        while(k > 0) {
                assert(L != nullptr);
                L = L->suivant;
                -- k;
        }
        assert(L != nullptr);
        return L->valeur;
}
```

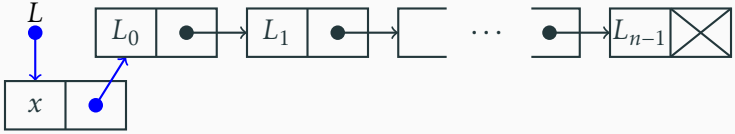
La complexité est en $\Theta(k)$.

Ajout d'éléments

Ajouter un élément x au début d'une liste est possible.

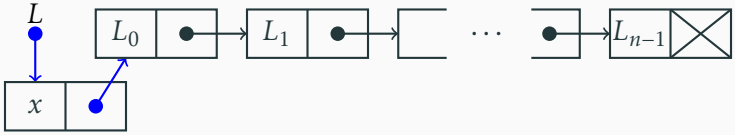
Ajout d'éléments

Ajouter un élément x au début d'une liste est possible.



Ajout d'éléments

Ajouter un élément x au début d'une liste est possible.



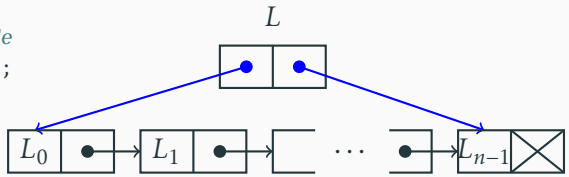

```
void ajouter(List &L, int x) {
    Cellule *c = new Cellule;
    c->valeur = x:
    c->next = L;
    L = c;
}
```

Ajout d'éléments

Pour ajouter un élément à la fin rapidement, il faut modifier la structure de liste pour avoir accès au dernier élément en temps constant.

```
struct List {
    Cellule *debut, *fin;
};

void ajouter_fin(List &L, int x) {
    if(fin == nullptr) { // Liste vide
        L.debut = L.fin = new Cellule;
    } else { // Liste non vide
        L.fin->suivant = new Cellule;
        L.fin = L.fin->suivant;
    }
    L.fin->valeur = x;
    L.fin->suivant = nullptr;
}
```



Suppression d'éléments

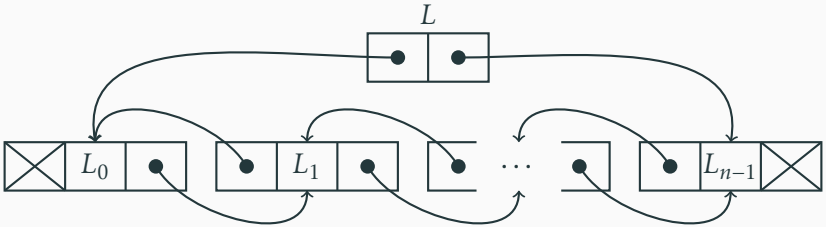
On peut également retirer le premier ou le dernier élément d'une liste en $\Theta(1)$.

```
int retirer_debut(List &L) {
    assert(L.debut != nullptr);
    int x = L.debut->valeur;
    Cellule *c = L.debut->suivant;
    delete L.debut;
    L.debut = c;
    if(c == nullptr) // L est vide
        L.fin = nullptr;
    return x;
}
```

```
int retirer_fin(List &L) {
    assert(L.fin != nullptr);
    int x = L.fin->valeur;
    Cellule *c = /* avant dernière valeur ? */;
    delete L.fin;
    L.fin = c;
    if(c == nullptr) // L est vide
        L.debut = nullptr;
    return x;
}
```

Liste doublement chaînée

Une liste doublement chaînée est une liste dont les cellules stockes à la fois un pointeur vers leur successeur et un pointeur vers leur prédécesseur.



Suppression d'éléments

On peut désormais retirer le dernier élément d'une liste en $\Theta(1)$.

```
struct Cellule {
        int valeur;
        Cellule *precedent, *suivant,
};

int retirer_fin(List &L) {
    assert(L.fin != nullptr);
    int x = L.fin->valeur;
    Cellule *c = L.fin->precedent;
    delete L.fin;
    L.fin = c;
    if(c == nullptr) // L est vide
        L.debut = nullptr;
    return x;
}
```

Pile

Pile

Une pile est un type abstrait dit LIFO (*Last In First Out*) qui supporte deux types d'opérations :

- **Insertion** d'un nouvel élément à la structure.
- Accès et suppression du **dernier élément** ajouté.

Une pile peut être implémentée à l'aide d'une **liste chaînée** et des opérations `ajouter_debut` et `retirer_debut`, toutes deux en $\Theta(1)$.

File

File

Une file est un type abstrait dit FIFO (*First In First Out*) qui supporte deux types d'opérations :

- **Insertion** d'un nouvel élément à la structure.
- Accès et suppression du **premier élément** ajouté.

Une file peut être implémentée à l'aide d'une **liste chaînée** et des opérations `ajouter_fin` et `retirer_debut`, toutes deux en $\Theta(1)$.

File de priorité

Une file est un type abstrait qui supporte deux types d'opérations :

- **Insertion** d'un nouvel élément à la structure.
- Accès et suppression du **plus grand élément**.

Une file de priorité peut être implémentée à l'aide d'une **liste doublement chaînée**. Selon si la liste est maintenue triées on obtient des complexités différentes (voir TP).

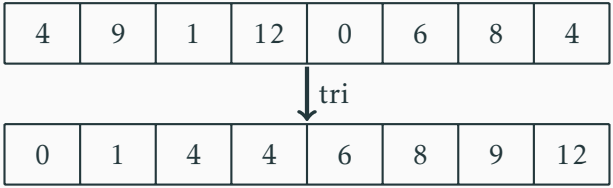
Structure	Insertion	Trouver + grand	Surpression
Liste doubl. chaînée	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Idem + triée	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Algorithmes de tri

Problème

Étant donné un tableau T de taille n, on cherche à trier les valeurs du tableau de sorte à avoir :

$$T[i] \leqslant T[i+1], \quad \forall i < n-1$$

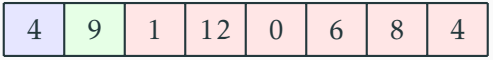


Algorithmes de tri

Tri par insertion

Tri par insertion

Le tri par insertion consiste à itérer sur les éléments du tableau et à les insérer à la bonne position parmi les éléments déjà triés. Ainsi, au moment où on considère un élément, les éléments qui le précèdent sont déjà triés, tandis que les éléments qui le suivent ne sont pas encore triés.



Tri par insertion

Le tri par insertion consiste à itérer sur les éléments du tableau et à les insérer à la bonne position parmi les éléments déjà triés. Ainsi, au moment où on considère un élément, les éléments qui le précèdent sont déjà triés, tandis que les éléments qui le suivent ne sont pas encore triés.

4	9	1	12	0	6	8	4

4	9	1	12	0	6	8	4

Tri par insertion

Le tri par insertion consiste à itérer sur les éléments du tableau et à les insérer à la bonne position parmi les éléments déjà triés. Ainsi, au moment où on considère un élément, les éléments qui le précèdent sont déjà triés, tandis que les éléments qui le suivent ne sont pas encore triés.

4	9	1	12	0	6	8	4
4	9	1	12	0	6	8	4
1	4	9	12	0	6	8	4

Tri par insertion

Le tri par insertion consiste à itérer sur les éléments du tableau et à les insérer à la bonne position parmi les éléments déjà triés. Ainsi, au moment où on considère un élément, les éléments qui le précèdent sont déjà triés, tandis que les éléments qui le suivent ne sont pas encore triés.

4	9	1	12	0	6	8	4
4	9	1	12	0	6	8	4
1	4	9	12	0	6	8	4
1	4	9	12	0	6	8	4

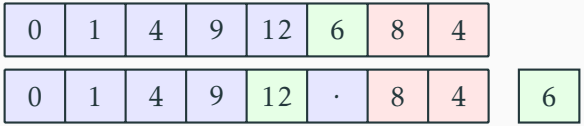
0	1	4	9	12	6	8	4
0	1	4	6	9	12	8	4
0	1	4	6	8	9	12	4
0	1	4	4	6	8	9	12

Complexité

L'insertion du i-ème élément se fait un tableau trié contenant i valeurs. Coût de l'insertion dans le pire cas en $\mathcal{O}(i)$.

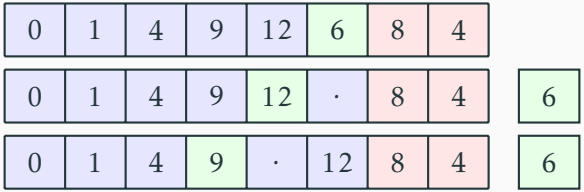
Complexité

L'insertion du i-ème élément se fait un tableau trié contenant i valeurs. Coût de l'insertion dans le pire cas en $\mathcal{O}(i)$.



Complexité

L'insertion du i-ème élément se fait un tableau trié contenant i valeurs. Coût de l'insertion dans le pire cas en $\mathcal{O}(i)$.



Complexité

L'insertion du i-ème élément se fait un tableau trié contenant i valeurs. Coût de l'insertion dans le pire cas en $\mathcal{O}(i)$.

Complexité

L'insertion du i-ème élément se fait un tableau trié contenant i valeurs. Coût de l'insertion dans le pire cas en $\mathcal{O}(i)$.

Pire cas : La complexité du tri par insertion est alors :

$$C(n) = \sum_{i=0}^{n-1} i = n(n-1)/2 = \mathbf{\Theta(n^2)}$$

Pire cas : La complexité du tri par insertion est alors :

$$C(n) = \sum_{i=0}^{n-1} i = n(n-1)/2 = \mathbf{\Theta(n^2)}$$

Meilleur cas : Si le tableau est déjà trié, aucune valeur ne doit être déplacé. La complexité est en $\mathbf{\Theta(n)}$ (Chaque élément est comparé à son prédécesseur).

Pire cas : La complexité du tri par insertion est alors :

$$C(n) = \sum_{i=0}^{n-1} i = n(n-1)/2 = \boldsymbol{\Theta(n^2)}$$

Meilleur cas : Si le tableau est déjà trié, aucune valeur ne doit être déplacé. La complexité est en $\boldsymbol{\Theta(n)}$ (Chaque élément est comparé à son prédécesseur).

En moyenne : l'insertion du i-ème élément nécessitent de déplacer $i/2$ valeurs.

$$C(n) = \sum_{i=0}^{n-1} i/2 = n(n-1)/4 = \boldsymbol{\Theta(n^2)}$$

Algorithmes de tri

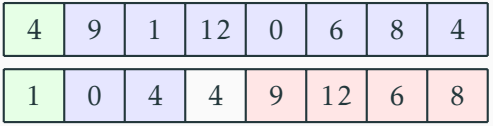
Tri rapide

Tri rapide

Le tri rapide consiste à choisir un élément du tableau, appelé **pivot**, et à répartir les autres éléments de part et d'autre du pivot : à gauche ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont strictement supérieurs. On fait ensuite un appel récursif de la méthode dans chaque sous-ensemble gauche et droite.

Tri rapide

Le tri rapide consiste à choisir un élément du tableau, appelé **pivot**, et à répartir les autres éléments de part et d'autre du pivot : à gauche ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont strictement supérieurs. On fait ensuite un appel récursif de la méthode dans chaque sous-ensemble gauche et droite.



Tri rapide

Le tri rapide consiste à choisir un élément du tableau, appelé **pivot**, et à répartir les autres éléments de part et d'autre du pivot : à gauche ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont strictement supérieurs. On fait ensuite un appel récursif de la méthode dans chaque sous-ensemble gauche et droite.

4	9	1	12	0	6	8	4
1	0	4	4	9	12	6	8
0	1	4	4	9	12	6	8

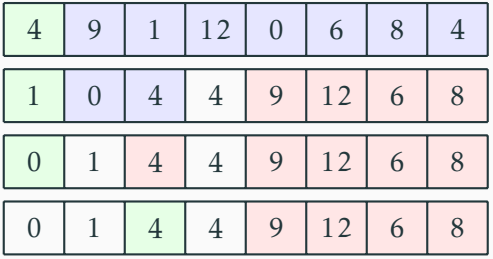
Tri rapide

Le tri rapide consiste à choisir un élément du tableau, appelé **pivot**, et à répartir les autres éléments de part et d'autre du pivot : à gauche ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont strictement supérieurs. On fait ensuite un appel récursif de la méthode dans chaque sous-ensemble gauche et droite.

4	9	1	12	0	6	8	4
1	0	4	4	9	12	6	8
0	1	4	4	9	12	6	8
0	1	4	4	9	12	6	8

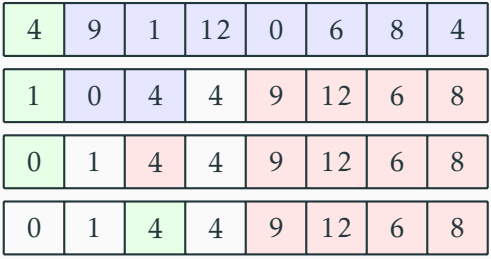
Tri rapide

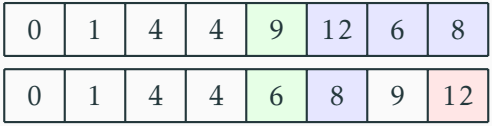
Le tri rapide consiste à choisir un élément du tableau, appelé **pivot**, et à répartir les autres éléments de part et d'autre du pivot : à gauche ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont strictement supérieurs. On fait ensuite un appel récursif de la méthode dans chaque sous-ensemble gauche et droite.



Tri rapide

Le tri rapide consiste à choisir un élément du tableau, appelé **pivot**, et à répartir les autres éléments de part et d'autre du pivot : à gauche ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont strictement supérieurs. On fait ensuite un appel récursif de la méthode dans chaque sous-ensemble gauche et droite.





Tri rapide

Le tri rapide consiste à choisir un élément du tableau, appelé **pivot**, et à répartir les autres éléments de part et d'autre du pivot : à gauche ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont strictement supérieurs. On fait ensuite un appel récursif de la méthode dans chaque sous-ensemble gauche et droite.

4	9	1	12	0	6	8	4
1	0	4	4	9	12	6	8
0	1	4	4	9	12	6	8
0	1	4	4	9	12	6	8

0	1	4	4	9	12	6	8
0	1	4	4	6	8	9	12
0	1	4	4	6	8	9	12

Tri rapide

Le tri rapide consiste à choisir un élément du tableau, appelé **pivot**, et à répartir les autres éléments de part et d'autre du pivot : à gauche ceux qui sont inférieurs ou égaux au pivot, et à droite ceux qui sont strictement supérieurs. On fait ensuite un appel récursif de la méthode dans chaque sous-ensemble gauche et droite.

4	9	1	12	0	6	8	4
1	0	4	4	9	12	6	8
0	1	4	4	9	12	6	8
0	1	4	4	9	12	6	8

0	1	4	4	9	12	6	8
0	1	4	4	6	8	9	12
0	1	4	4	6	8	9	12
0	1	4	4	6	8	9	12

Complexité

La répartition des éléments de part et d'autre du pivot est en $\Theta(n)$.
Si g éléments sont placés à gauche, et d à droite du pivot (avec $d + g = n - 1$), alors, la complexité $C(n)$ vérifie :

$$C(n) = C(g) + C(d) + \Theta(n)$$

Complexité

La répartition des éléments de part et d'autre du pivot est en $\Theta(n)$.
Si g éléments sont placés à gauche, et d à droite du pivot (avec $d + g = n - 1$), alors, la complexité $C(n)$ vérifie :

$$C(n) = C(g) + C(d) + \Theta(n)$$

Pire cas : Le tableau est déjà trié, tous les éléments sont placés à droite ($g = 0$ et $d = n - 1$).

$$C(n) = C(n-1) + n - 1 = \sum_{i=0}^{n-1} i = n(n-1)/2 = \mathbf{\Theta(n^2)}$$

Complexité

La répartition des éléments de part et d'autre du pivot est en $\Theta(n)$.
Si g éléments sont placés à gauche, et d à droite du pivot (avec $d + g = n - 1$), alors, la complexité $C(n)$ vérifie :

$$C(n) = C(g) + C(d) + \Theta(n)$$

Pire cas : Le tableau est déjà trié, tous les éléments sont placés à droite ($g = 0$ et $d = n - 1$).

$$C(n) = C(n-1) + n - 1 = \sum_{i=0}^{n-1} i = n(n-1)/2 = \mathbf{\Theta(n^2)}$$

Meilleur cas : Le pivot est la médiane, $g = d = (n-1)/2$.

$$C(n) = 2C(n/2) + \Theta(n)$$

Le théorème maître donne $C(n) = \mathbf{\Theta(n \log(n))}$.

Complexité moyenne

- Chaque élément est choisi un fois en tant que pivot $\implies \Theta(n)$.

Complexité moyenne

- Chaque élément est choisi un fois en tant que pivot $\implies \Theta(n)$.
- On note z_i la i-ème valeur, une fois le tableau trié. Lorsque z_i et z_j sont comparés, l'un d'eux est pivot et ne sera ensuite plus comparé.
  Donc z_i et z_j sont comparés au plus une fois !
  Soit $X_{ij} \in \{0, 1\}$ le nombre de fois que z_i et z_j sont comparés.

Complexité moyenne

- Chaque élément est choisi un fois en tant que pivot $\implies \Theta(n)$.
- On note z_i la i-ème valeur, une fois le tableau trié. Lorsque z_i et z_j sont comparés, l'un d'eux est pivot et ne sera ensuite plus comparé.
  Donc z_i et z_j sont comparés au plus une fois !
  Soit $X_{ij} \in \{0, 1\}$ le nombre de fois que z_i et z_j sont comparés.

Le nombre de comparaisons total est noté $X^{(n)} = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} X_{ij}$. La complexité moyenne est :

$$C(n) = \mathbb{E}\left[X^{(n)}\right] + \Theta(n)$$

Complexité moyenne

Soit k tel que $i < k < j$.
Si z_k est choisi comme pivot avant z_i et z_k, alors z_i se retrouve à droite de z_k et z_j à gauche. z_i et z_j ne seront alors jamais comparés.

Complexité moyenne

Soit k tel que $i < k < j$.
Si z_k est choisi comme pivot avant z_i et z_k, alors z_i se retrouve à droite de z_k et z_j à gauche. z_i et z_j ne seront alors jamais comparés.
z_i et z_j sont comparés *ssi* le premier pivot choisi parmi $\{z_i, z_{i+1}, \ldots, z_j\}$ est z_i ou z_j.

Complexité moyenne

Soit k tel que $i < k < j$.

Si z_k est choisi comme pivot avant z_i et z_k, alors z_i se retrouve à droite de z_k et z_j à gauche. z_i et z_j ne seront alors jamais comparés.

z_i et z_j sont comparés *ssi* le premier pivot choisi parmi $\{z_i, z_{i+1}, \ldots, z_j\}$ est z_i ou z_j.

$$E\left[X_{ij}\right] = \frac{2}{j-i+1}$$

$$\mathbb{E}\left[X^{(n)}\right] = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} = \sum_{i=0}^{n-2} \sum_{k=1}^{n-i-1} \frac{2}{k+1}$$

Complexité moyenne

Soit k tel que $i < k < j$.
Si z_k est choisi comme pivot avant z_i et z_k, alors z_i se retrouve à droite de z_k et z_j à gauche. z_i et z_j ne seront alors jamais comparés.
z_i et z_j sont comparés *ssi* le premier pivot choisi parmi $\{z_i, z_{i+1}, \ldots, z_j\}$ est z_i ou z_j.

$$E\left[X_{ij}\right] = \frac{2}{j-i+1}$$

$$\mathbb{E}\left[X^{(n)}\right] = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} = \sum_{i=0}^{n-2} \sum_{k=1}^{n-i-1} \frac{2}{k+1}$$

La série harmonique $\sum_{k=1}^{\infty} \frac{1}{k} = \mathcal{O}(\log(n))$. Donc :

$$\mathbb{E}\left[X^{(n)}\right] = \sum_{i=0}^{n-2} \mathcal{O}(\log(n)) = \mathcal{O}(n \log(n))$$

La complexité moyenne est donc un $\boldsymbol{\mathcal{O}(n \log(n))}$.

Algorithmes de tri

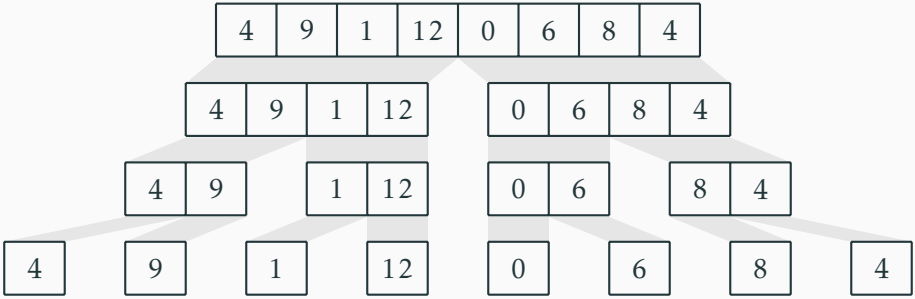
Tri par fusion

Tri fusion

Le tri fusion consiste à séparer le tableau en deux parties de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$, à trier récursivement les deux parties et à les fusionner.

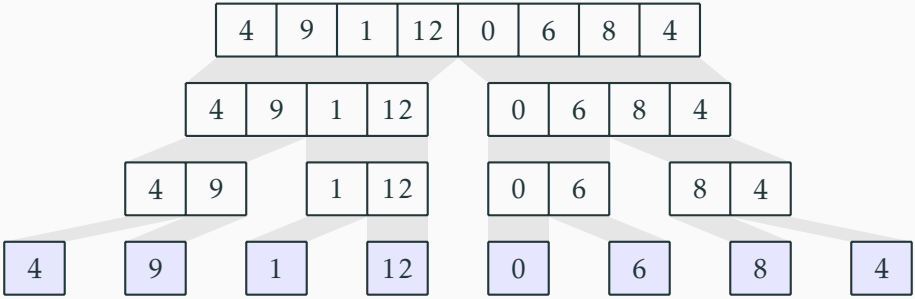
Tri fusion

Le tri fusion consiste à séparer le tableau en deux parties de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$, à trier récursivement les deux parties et à les fusionner.



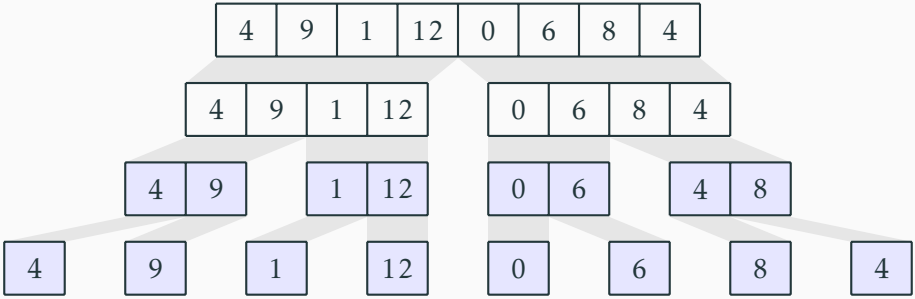
Tri fusion

Le tri fusion consiste à séparer le tableau en deux parties de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$, à trier récursivement les deux parties et à les fusionner.



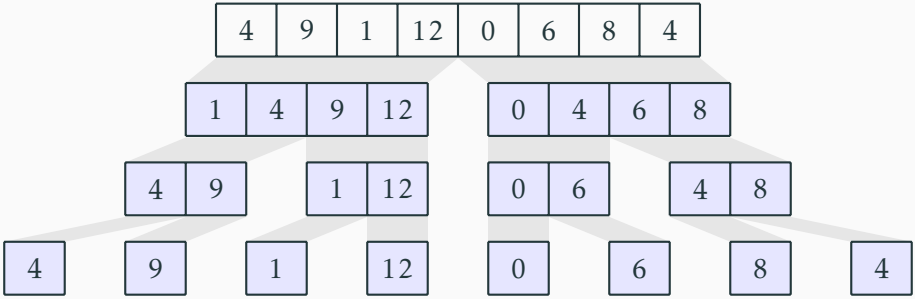
Tri fusion

Le tri fusion consiste à séparer le tableau en deux parties de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$, à trier récursivement les deux parties et à les fusionner.



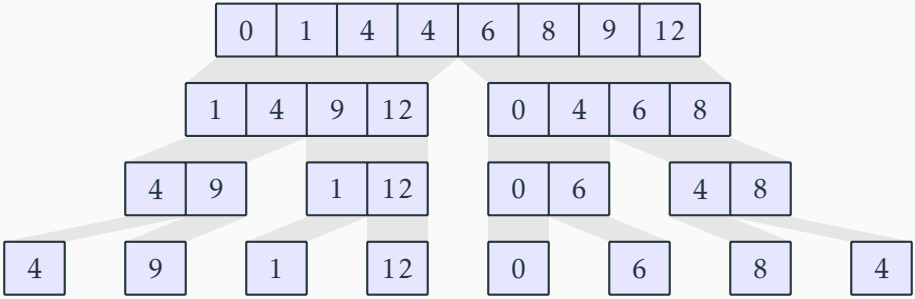
Tri fusion

Le tri fusion consiste à séparer le tableau en deux parties de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$, à trier récursivement les deux parties et à les fusionner.



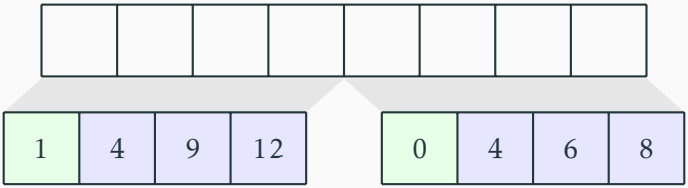
Tri fusion

Le tri fusion consiste à séparer le tableau en deux parties de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$, à trier récursivement les deux parties et à les fusionner.



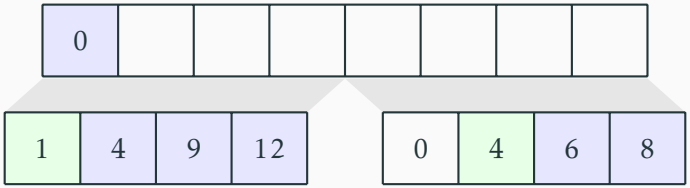
Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



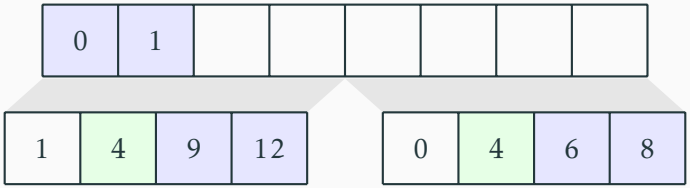
Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



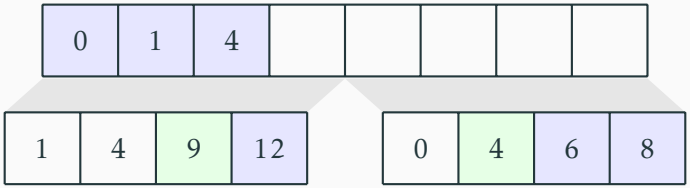
Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



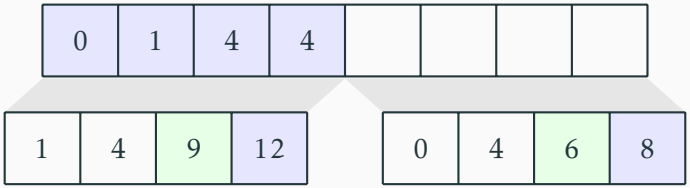
Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



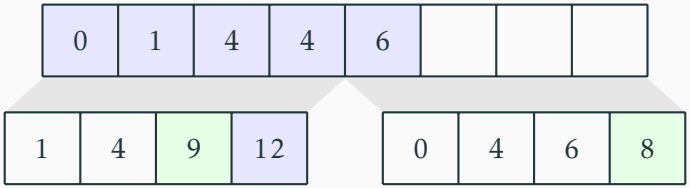
Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



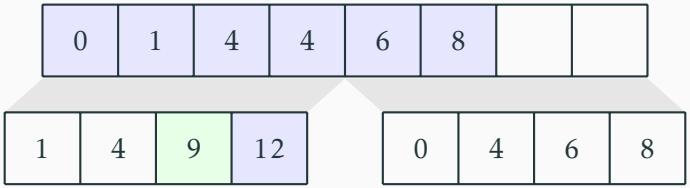
Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



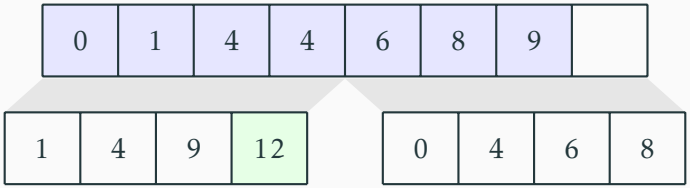
Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



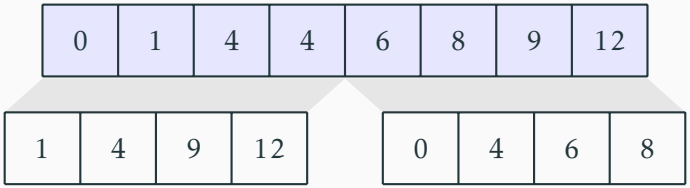
Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



Fusion

La principale opération de ce tri est la fusion de deux tableaux triés.



La fusion s'effectue en $\Theta(n)$. On obtient donc la relation :

$$C(n) = 2 \cdot C\left(\frac{n}{2}\right) + \Theta(n)$$

Avec le théorème maître nous obtenons une complexité en $\boldsymbol{\Theta(n \log(n))}$.

Récapitulatif

Algorithme	Pire cas	Moyen	Meilleur cas
Insertion	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Rapide	$\Theta(n^2)$	$\Theta(n\log(n))$	$\Theta(n\log(n))$
Fusion	$\Theta(n\log(n))$	$\Theta(n\log(n))$	$\Theta(n\log(n))$

Dans les langages de programmation

Python

Si L est de type `list`, alors `L.sort()` effectue en tri en place.

```
>>> L = [4, 2, 1, 3]
>>> L.sort()
>>> print(L)
[1, 2, 3, 4]
```

Si a est un itérable, alors `sorted(a)` renvoie un objet `list` trié.

```
>>> a = (4, 2, 1, 3)
>>> sorted(a)
[1, 2, 3, 4]
```

Dans les langages de programmation

C++

La fonction `std::sort` de la librairie standard trie les valeurs en place entre deux itérateurs.

```cpp
#include <algorithm>
#include <vector>

std::vector<int> v = {4, 2, 1, 3};
std::sort(v.begin(), v.end()); // désormais v = {1, 2, 3, 4}

int a[] = {4, 2, 1, 3};
std::sort(a, a+4); // désormais a = {1, 2, 3, 4}
```

Algorithmes de tri

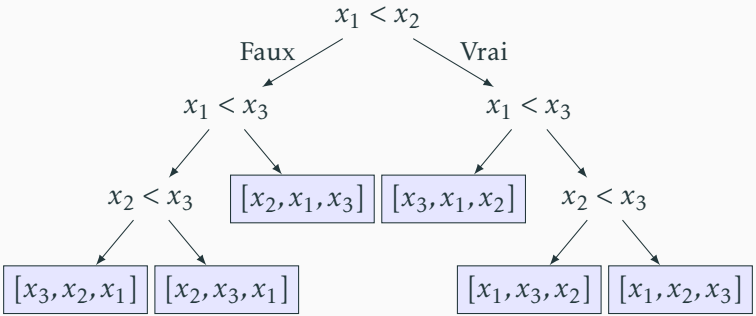
Complexité optimale

Complexité optimale

Question : Peut on faire mieux que $\Theta(n \log(n))$ dans le pire cas ?

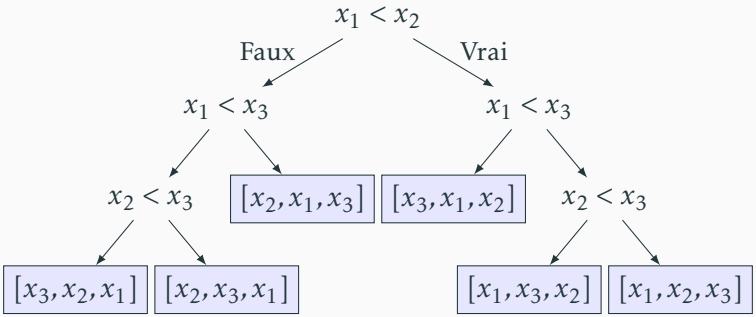
Complexité optimale

Question : Peut on faire mieux que $\Theta(n \log(n))$ dans le pire cas ?
On représente un algorithme de tri par un arbre de comparaison.



Complexité optimale

Question : Peut on faire mieux que $\Theta(n \log(n))$ dans le pire cas ?
On représente un algorithme de tri par un arbre de comparaison.



Les feuilles sont les résultats de l'algorithme. Il y en a $n!$, une par permutation.

Complexité optimale

- La longueur d'un chemin entre la racine et une feuille de l'arbre est le nombre de comparaisons effectuées pour trier le tableau.

Complexité optimale

- La longueur d'un chemin entre la racine et une feuille de l'arbre est le nombre de comparaisons effectuées pour trier le tableau.
- La **hauteur** H d'un arbre est la longueur du plus long chemin entre la racine et une feuille.
  La complexité dans le pire cas d'un algorithme de tri est donc la hauteur de l'arbre.
  Un arbre binaire de hauteur H possède au plus 2^H feuilles.
  Un arbre possédant F feuilles a donc une hauteur supérieur ou égale à $\lceil \log_2(F) \rceil$.

Complexité optimale

- La longueur d'un chemin entre la racine et une feuille de l'arbre est le nombre de comparaisons effectuées pour trier le tableau.
- La **hauteur** H d'un arbre est la longueur du plus long chemin entre la racine et une feuille.
  La complexité dans le pire cas d'un algorithme de tri est donc la hauteur de l'arbre.
  Un arbre binaire de hauteur H possède au plus 2^H feuilles.
  Un arbre possédant F feuilles a donc une hauteur supérieur ou égale à $\lceil \log_2(F) \rceil$.

On a donc :

$$C_{\text{opt}}(n) = H = \lceil \log_2(F) \rceil = \lceil \log_2(n!) \rceil$$

Complexité optimale

Formule de Stirling

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

En utilisant la formule de Stirling on obtient :

$$\log_2(n!) = \frac{1}{2} \cdot (1 + \log_2(\pi) + \log_2(n)) + n \cdot (\log_2(n) - \log_2(e)) = \Theta(n \log(n))$$

Conclusion : Un algorithme de tri ne peut donc pas faire mieux que $\boldsymbol{\Theta(n \log(n))}$ dans le pire cas !