

Chapitre 4

Diviser pour régner

HLIN401 : Algorithmique et complexité

L2 Informatique
Université de Montpellier
2020 – 2021

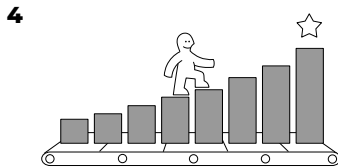
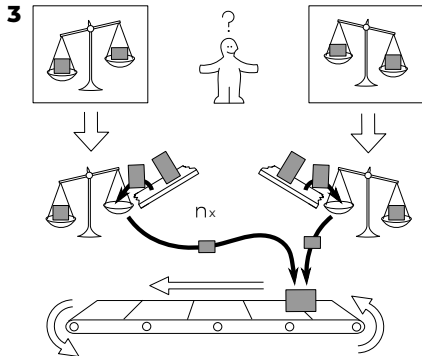
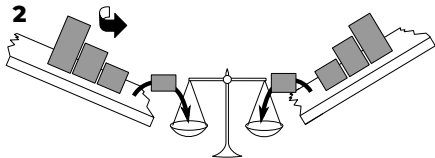
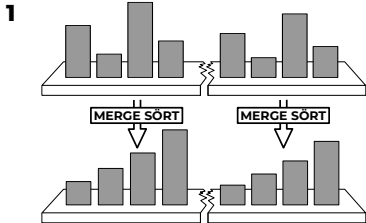
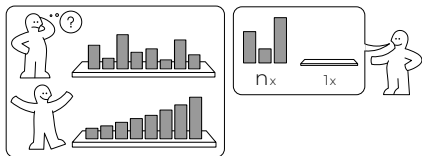
1. Premier exemple : tri fusion

2. Qu'est-ce que « diviser pour régner » ?

3. Deuxième exemple : multiplication d'entiers

4. Exemple spécial : Calcul de rang

MERGE SÖRT



Algorithme du TRIFUSION

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

si $n \leq 1$: **renvoyer** T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

renvoyer FUSION(T_1, T_2)

Algorithme du TRIFUSION

10	5	2	4	3	7	6	4
----	---	---	---	---	---	---	---

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

si $n \leq 1$: **renvoyer** T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

renvoyer FUSION(T_1, T_2)

Algorithme du TRIFUSION

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

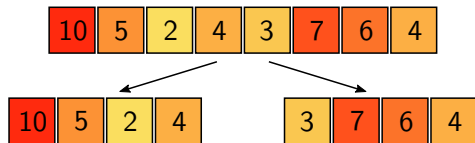
si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

 renvoyer FUSION(T_1, T_2)



Algorithme du TRIFUSION

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

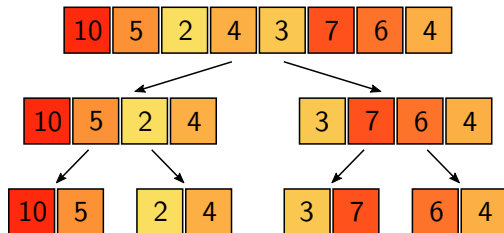
si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

 renvoyer FUSION(T_1, T_2)



Algorithme du TRIFUSION

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

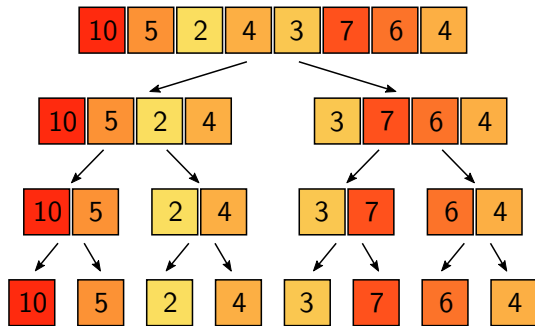
si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

 renvoyer FUSION(T_1, T_2)



Algorithme du TRIFUSION

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

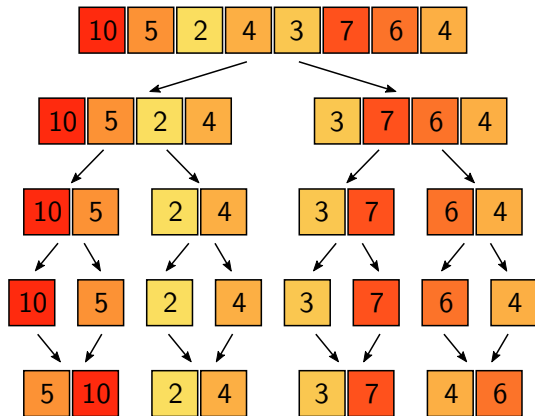
si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

renvoyer FUSION(T_1, T_2)



Algorithme du TRIFUSION

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

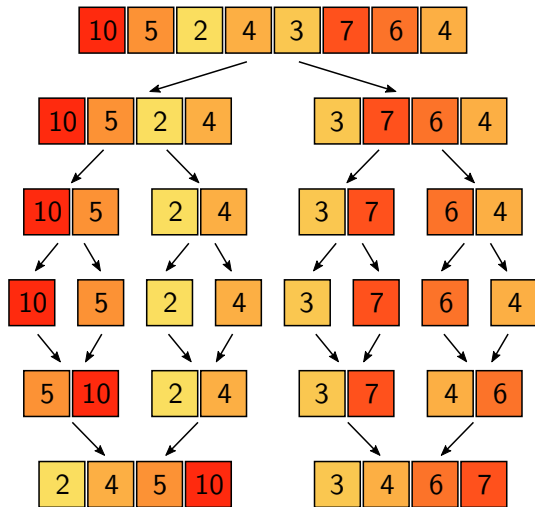
si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

renvoyer FUSION(T_1, T_2)



Algorithme du TRIFUSION

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

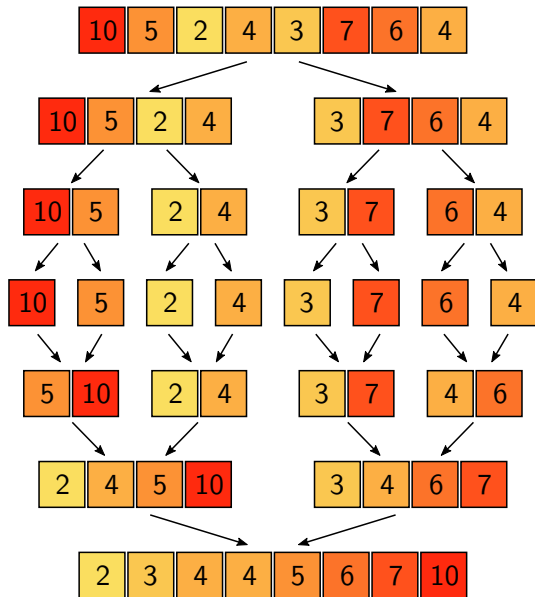
si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

renvoyer FUSION(T_1, T_2)



Algorithme du TRIFUSION

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

 renvoyer FUSION(T_1, T_2)

Lemme

Soit $t(n)$ la complexité de TRIFUSION et $F(n)$ la complexité de FUSION. Alors

$$t(n) = \begin{cases} O(1) & \text{si } n \leq 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + F(n) + O(1) & \text{sinon} \end{cases}$$

Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow$ tableau de taille $n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S

Idée de l'algorithme

- ▶ T_1 et T_2 vus comme des **pires**
- ▶ S vu comme une **file**
- ▶ À chaque itération,
 - ▶ on dépile la plus petite des deux têtes
 - ▶ on l'ajoute à S
- ▶ Quand une des piles est vide, on dépile l'autre

Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow$ tableau de taille $n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

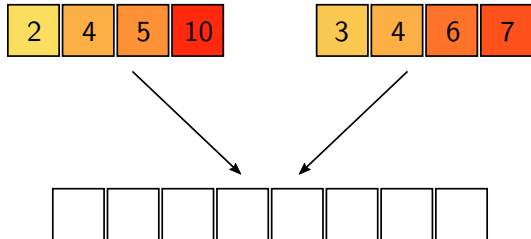
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow \text{tableau de taille } n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

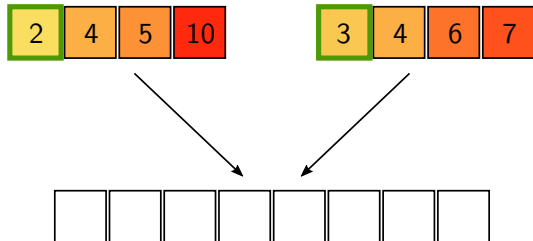
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow \text{tableau de taille } n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

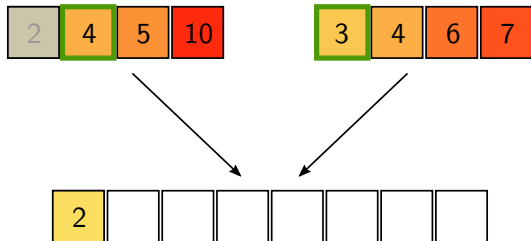
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow \text{tableau de taille } n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

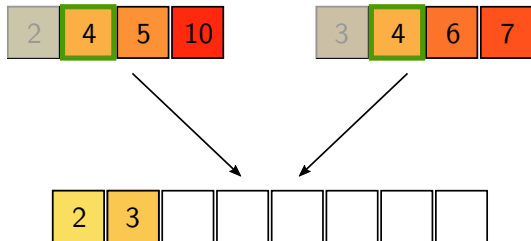
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow \text{tableau de taille } n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

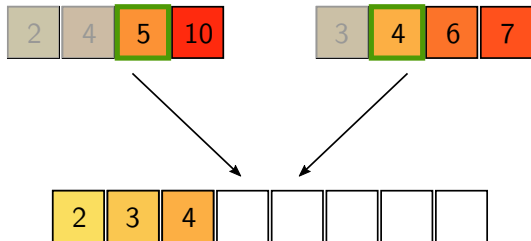
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow \text{tableau de taille } n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

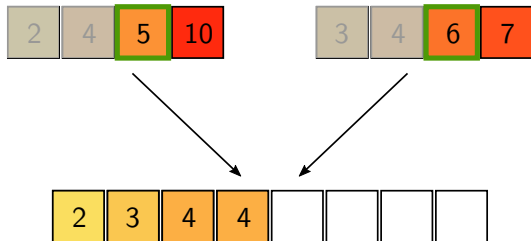
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow \text{tableau de taille } n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

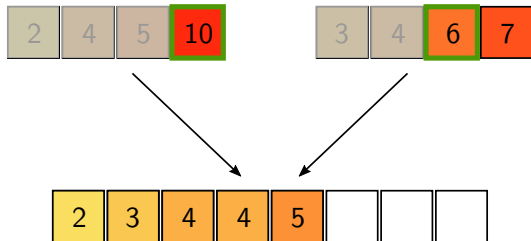
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow \text{tableau de taille } n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

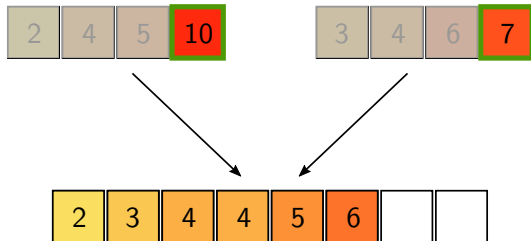
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow$ tableau de taille $n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

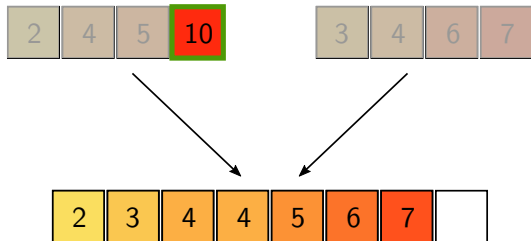
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow$ tableau de taille $n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

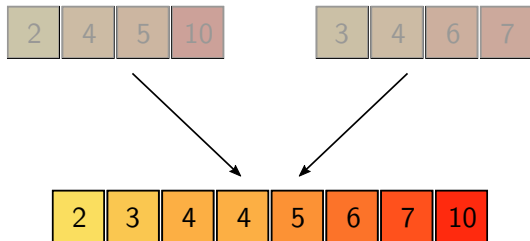
$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S



Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow$ tableau de taille $n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S

Lemme

La complexité $F(n)$ de FUSION est $O(n_1 + n_2)$.

Preuve facile...

Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow$ tableau de taille $n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S

Lemme

Si T_1 et T_2 sont deux tableaux triés (par ordre croissant), FUSION(T_1, T_2) renvoie un tableau trié contenant l'union des éléments de T_1 et T_2 .

Algorithme de FUSION

Algorithme : FUSION(T_1, T_2)

$n_1 \leftarrow \text{taille}(T_1)$; $n_2 \leftarrow \text{taille}(T_2)$

$S \leftarrow$ tableau de taille $n_1 + n_2$

$i_1 \leftarrow 0$; $i_2 \leftarrow 0$

pour $i_S = 0$ à $n_1 + n_2 - 1$:

si $i_2 \geq n_2$ **ou** ($i_1 < n_1$ **et** $T_1[i_1] \leq T_2[i_2]$) :

$S[i_S] \leftarrow T_1[i_1]$

$i_1 \leftarrow i_1 + 1$

sinon : // $i_1 \geq n_1$ ou $T_1[i_1] > T_2[i_2]$

$S[i_S] \leftarrow T_2[i_2]$

$i_2 \leftarrow i_2 + 1$

renvoyer S

Lemme

Si T_1 et T_2 sont deux tableaux triés (par ordre croissant), FUSION(T_1, T_2) renvoie un tableau trié contenant l'union des éléments de T_1 et T_2 .

Preuve rapide. Invariant \mathcal{P}_{i_S} :

à l'entrée de l'itération i_S de la boucle,

1. $S_{[0, i_S[}$ contient les i_S plus petits éléments de $T_1 \cup T_2$ en ordre croissant
2. i_1 est l'indice du plus petit élément de T_1 non présent dans S
3. i_2 est l'indice du plus petit élément de T_2 non présent dans S

Retour sur le TRIFUSION

Théorème

L'algorithme TRIFUSION trie tout tableau de taille n en temps $O(n \log n)$.

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[]})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[]})$

renvoyer FUSION(T_1, T_2)

Retour sur le TRIFUSION

Théorème

L'algorithme TRIFUSION trie tout tableau de taille n en temps $O(n \log n)$.

Preuve de correction par récurrence (facile !)

- Si $n \leq 1$, OK
- Si $n > 1$, $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$, donc T_1 et T_2 triés après appels récursifs. La correction de FUSION suffit à conclure.

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

si $n \leq 1$: renvoyer T

sinon :

$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor]})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lceil n/2 \rceil, n]})$

 renvoyer FUSION(T_1, T_2)

Retour sur le TRIFUSION

Théorème

L'algorithme TRIFUSION trie tout tableau de taille n en temps $O(n \log n)$.

Preuve de correction par récurrence (facile !)

- ▶ Si $n \leq 1$, OK
- ▶ Si $n > 1$, $\lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$, donc T_1 et T_2 triés après appels récursifs. La correction de FUSION suffit à conclure.

Preuve de complexité (étapes)

- ▶ Équation de récurrence : $t(n) \leq 2t(\lceil n/2 \rceil) + O(n)$
- ▶ Arbre de récursion \rightsquigarrow estimation du temps de calcul
- ▶ Preuve par récurrence de l'estimation

Algorithme : TRIFUSION(T)

$n \leftarrow \text{taille}(T)$

si $n \leq 1$: renvoyer T

sinon :

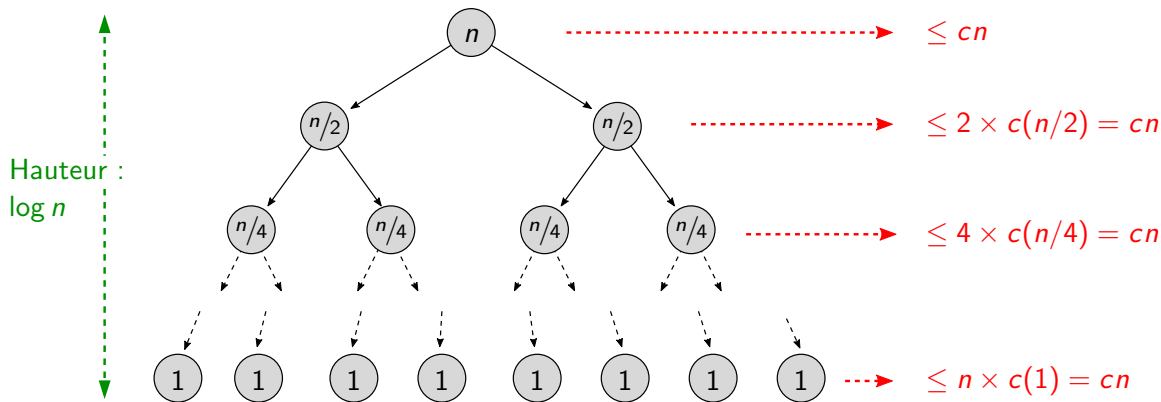
$T_1 \leftarrow \text{TRIFUSION}(T_{[0, \lfloor n/2 \rfloor[})$

$T_2 \leftarrow \text{TRIFUSION}(T_{[\lfloor n/2 \rfloor, n[})$

 renvoyer FUSION(T_1, T_2)

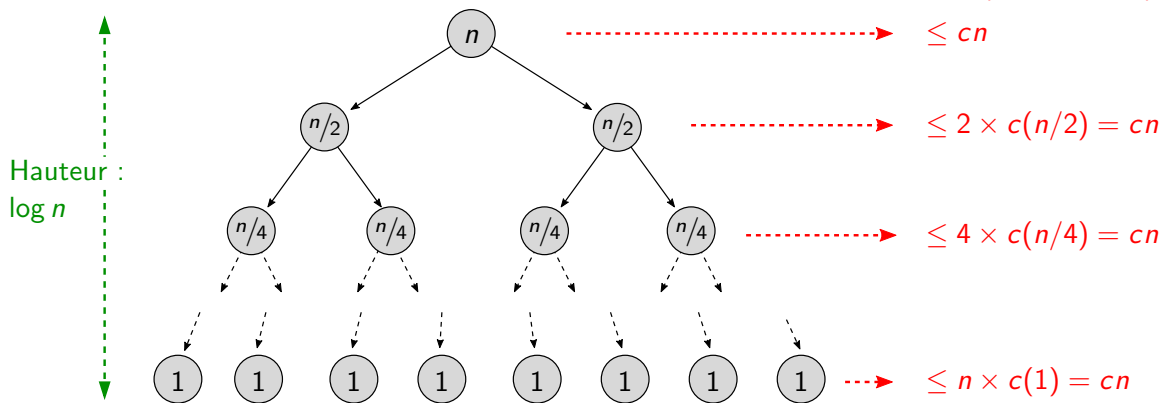
TRIFUSION : idée de la complexité, arbre de récursion

- Soit c une constante telle que $F(n) \leq cn$ (FUSION)



TRIFUSION : idée de la complexité, arbre de récursion

- Soit c une constante telle que $F(n) \leq cn$ (FUSION)



- En tout, l'algo a une complexité approximative de $cn \log n$, c'est-à-dire $O(n \log n)$

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
 - ▶ \mathcal{P}_0 est vraie.
 - ▶ Supposons \mathcal{P}_{k-1} vraie, alors pour k on a :

$$t(2^k) \leq c2^k + 2t(2^{k-1}) \quad (\text{éq. de réc.})$$

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
 - ▶ \mathcal{P}_0 est vraie.
 - ▶ Supposons \mathcal{P}_{k-1} vraie, alors pour k on a :

$$t(2^k) \leq c2^k + 2t(2^{k-1}) \quad (\text{éq. de réc.})$$

$$\leq c2^k + 2(c k 2^{k-1}) \quad (\text{hyp. de réc.})$$

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
 - ▶ \mathcal{P}_0 est vraie.
 - ▶ Supposons \mathcal{P}_{k-1} vraie, alors pour k on a :

$$t(2^k) \leq c2^k + 2t(2^{k-1}) \quad (\text{éq. de réc.})$$

$$\leq c2^k + 2(ck2^{k-1}) \quad (\text{hyp. de réc.})$$

$$\leq c2^k + ck2^k$$

$$\leq c(k+1)2^k$$

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
- ▶ **Étape 2** : on démontre la propriété $\mathcal{Q}_n = \ll t(n) \leq 4cn \log(2n) \gg$ pour $n \geq 1$

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
- ▶ **Étape 2** : on démontre la propriété $\mathcal{Q}_n = \ll t(n) \leq 4cn \log(2n) \gg$ pour $n \geq 1$
 - ▶ Pour n quelconque, il existe k tel que $n \leq 2^k < 2n$
 - ▶ Donc $t(n) \leq t(2^k) \leq c(k+1)2^k$ par l'étape 1


TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
- ▶ **Étape 2** : on démontre la propriété $\mathcal{Q}_n = \ll t(n) \leq 4cn \log(2n) \gg$ pour $n \geq 1$
 - ▶ Pour n quelconque, il existe k tel que $n \leq 2^k < 2n$
 - ▶ Donc $t(n) \leq t(2^k) \leq c(k+1)2^k$ par l'étape 1
 - ▶ Or $k+1 = \log(2^{k+1}) \leq \log(2 \times 2n) = \log(2n) + 1$

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
- ▶ **Étape 2** : on démontre la propriété $\mathcal{Q}_n = \ll t(n) \leq 4cn \log(2n) \gg$ pour $n \geq 1$
 - ▶ Pour n quelconque, il existe k tel que $n \leq 2^k < 2n$
 - ▶ Donc $t(n) \leq t(2^k) \leq c(k+1)2^k$ par l'étape 1
 - ▶ Or $k+1 = \log(2^{k+1}) \leq \log(2 \times 2n) = \log(2n) + 1$
 - ▶ D'où $c(k+1)2^k \leq c(\log(2n) + 1)(2n) = 2cn(\log(2n) + 1)$

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
- ▶ **Étape 2** : on démontre la propriété $\mathcal{Q}_n = \ll t(n) \leq 4cn \log(2n) \gg$ pour $n \geq 1$
 - ▶ Pour n quelconque, il existe k tel que $n \leq 2^k < 2n$
 - ▶ Donc $t(n) \leq t(2^k) \leq c(k+1)2^k$ par l'étape 1
 - ▶ Or $k+1 = \log(2^{k+1}) \leq \log(2 \times 2n) = \log(2n) + 1$
 - ▶ D'où $c(k+1)2^k \leq c(\log(2n) + 1)(2n) = 2cn(\log(2n) + 1)$
 - ▶ Donc $t(n) \leq 2cn(\log(2n) + 1) \leq 4cn \log(2n)$ 

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
- ▶ **Étape 2** : on démontre la propriété $\mathcal{Q}_n = \ll t(n) \leq 4cn \log(2n) \gg$ pour $n \geq 1$
 - ▶ Pour n quelconque, il existe k tel que $n \leq 2^k < 2n$
 - ▶ Donc $t(n) \leq t(2^k) \leq c(k+1)2^k$ par l'étape 1
 - ▶ Or $k+1 = \log(2^{k+1}) \leq \log(2 \times 2n) = \log(2n) + 1$
 - ▶ D'où $c(k+1)2^k \leq c(\log(2n) + 1)(2n) = 2cn(\log(2n) + 1)$
 - ▶ Donc $t(n) \leq 2cn(\log(2n) + 1) \leq 4cn \log(2n)$ ■
- ▶ Arg... C'est un peu fastidieux...

TRIFUSION : preuve de la complexité, par récurrence

- ▶ On note $F(n) \leq cn$
- ▶ On note $t(n)$ le nombre total d'opérations élémentaires effectuées par TRIFUSION appelé sur un tableau de taille n .
- ▶ **Étape 1** : on démontre la propriété $\mathcal{P}_k = \ll t(2^k) \leq c(k+1)2^k \gg$ pour $k \geq 0$
- ▶ **Étape 2** : on démontre la propriété $\mathcal{Q}_n = \ll t(n) \leq 4cn \log(2n) \gg$ pour $n \geq 1$
 - ▶ Pour n quelconque, il existe k tel que $n \leq 2^k < 2n$
 - ▶ Donc $t(n) \leq t(2^k) \leq c(k+1)2^k$ par l'étape 1
 - ▶ Or $k+1 = \log(2^{k+1}) \leq \log(2 \times 2n) = \log(2n) + 1$
 - ▶ D'où $c(k+1)2^k \leq c(\log(2n) + 1)(2n) = 2cn(\log(2n) + 1)$
 - ▶ Donc $t(n) \leq 2cn(\log(2n) + 1) \leq 4cn \log(2n)$ ■
- ▶ Arg... C'est un peu fastidieux...
- ▶ Remarque : on a utilisé $t(n) \leq t(m)$ pour $n \leq m$.

1. Premier exemple : tri fusion
2. Qu'est-ce que « diviser pour régner » ?
3. Deuxième exemple : multiplication d'entiers
4. Exemple spécial : Calcul de rang

La stratégie « diviser pour régner »

1. **Diviser** le problème en sous-problèmes
2. **Résoudre récursivement** ces sous-problèmes
3. **Combiner** les solutions pour reconstruire la solution du problème original.

La stratégie « diviser pour régner »

1. **Diviser** le problème en sous-problèmes
 2. **Résoudre récursivement** ces sous-problèmes
 3. **Combiner** les solutions pour reconstruire la solution du problème original.
- Stratégie principalement utilisée pour obtenir de meilleures complexités que celles données par un algorithme moins évolué.

La stratégie « diviser pour régner »

1. **Diviser** le problème en sous-problèmes
 2. **Résoudre récursivement** ces sous-problèmes
 3. **Combiner** les solutions pour reconstruire la solution du problème original.
- ▶ Stratégie principalement utilisée pour obtenir de meilleures complexités que celles données par un algorithme moins évolué.
 - ▶ Exemple : la recherche dichotomique

La stratégie « diviser pour régner »

1. **Diviser** le problème en sous-problèmes
 2. **Résoudre récursivement** ces sous-problèmes
 3. **Combiner** les solutions pour reconstruire la solution du problème original.
- ▶ Stratégie principalement utilisée pour obtenir de meilleures complexités que celles données par un algorithme moins évolué.
 - ▶ Exemple : la recherche dichotomique

Exemple du tri fusion

1. **Diviser** le tableau en 2 sous-tableaux de tailles environ égales
2. **Trier récursivement** chaque sous-tableau
3. **Fusionner** les sous-tableaux triés

Analyse d'un algorithme « diviser pour régner »

Récurrance(s) sur la taille du problème

Analyse d'un algorithme « diviser pour régner »

Récurrance(s) sur la taille du problème

- ▶ Preuve de correction
- ▶ Complexité :
 1. Établir **l'équation de récurrence**
 2. Estimer le résultat (arbre de récursion, déroulement de la récurrence, habitude, ...)
 3. Preuve par récurrence

Analyse d'un algorithme « diviser pour régner »

Récurrance(s) sur la taille du problème

- ▶ Preuve de correction
- ▶ Complexité :
 1. Établir **l'équation de récurrence**
 2. Estimer le résultat (arbre de récursion, déroulement de la récurrence, habitude, ...)
 3. Preuve par récurrence

ou

- 2-3. Utiliser le **master theorem** !

Une version du « *master theorem* »

Théorème

S'il existe trois entiers $a \geq 0$, $b > 1$, $d \geq 0$ et $n_0 > 0$ tels que pour tout $n \geq n_0$,

$$T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$$

Alors

$$T(n) = \begin{cases} O(n^d) & \text{si } b^d > a \\ O(n^d \log n) & \text{si } b^d = a \\ O(n^{\frac{\log a}{\log b}}) & \text{si } b^d < a \end{cases}$$

Une version du « *master theorem* »

Théorème

S'il existe trois entiers $a \geq 0$, $b > 1$, $d \geq 0$ et $n_0 > 0$ tels que pour tout $n \geq n_0$,

$$T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$$

Alors

$$T(n) = \begin{cases} O(n^d) & \text{si } b^d > a \\ O(n^d \log n) & \text{si } b^d = a \\ O(n^{\frac{\log a}{\log b}}) & \text{si } b^d < a \end{cases}$$

Exemple du tri fusion

► $T(n) \leq 2T(\lceil n/2 \rceil) + O(n)$

Une version du « *master theorem* »

Théorème

S'il existe trois entiers $a \geq 0$, $b > 1$, $d \geq 0$ et $n_0 > 0$ tels que pour tout $n \geq n_0$,

$$T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$$

Alors

$$T(n) = \begin{cases} O(n^d) & \text{si } b^d > a \\ O(n^d \log n) & \text{si } b^d = a \\ O(n^{\frac{\log a}{\log b}}) & \text{si } b^d < a \end{cases}$$

Exemple du tri fusion

► $T(n) \leq 2T(\lceil n/2 \rceil) + O(n) : a = 2, b = 2, d = 1$

Une version du « *master theorem* »

Théorème

S'il existe trois entiers $a \geq 0$, $b > 1$, $d \geq 0$ et $n_0 > 0$ tels que pour tout $n \geq n_0$,

$$T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$$

Alors

$$T(n) = \begin{cases} O(n^d) & \text{si } b^d > a \\ O(n^d \log n) & \text{si } b^d = a \\ O(n^{\frac{\log a}{\log b}}) & \text{si } b^d < a \end{cases}$$

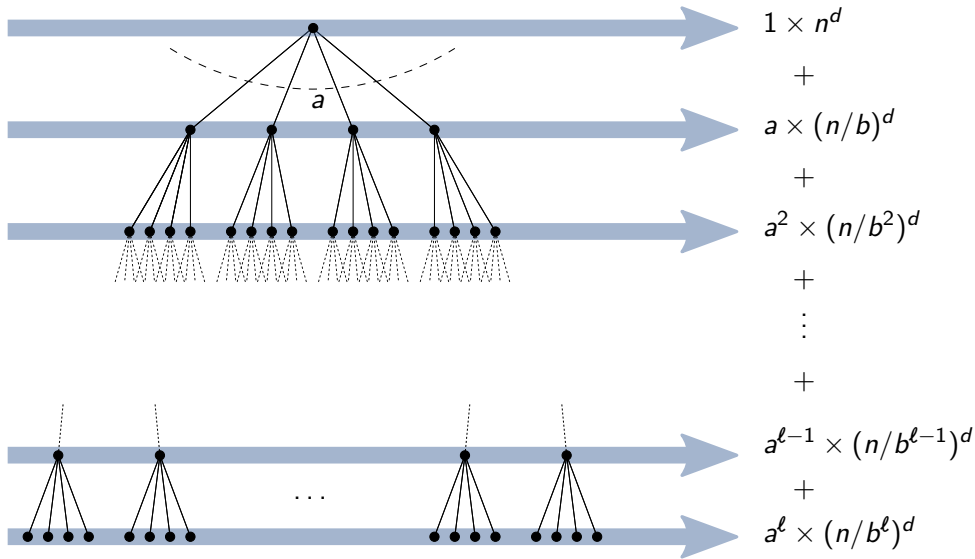
Exemple du tri fusion

- ▶ $T(n) \leq 2T(\lceil n/2 \rceil) + O(n)$: $a = 2$, $b = 2$, $d = 1$
- ▶ $b^d = a \rightsquigarrow T(n) = O(n^d \log n) = O(n \log n)$

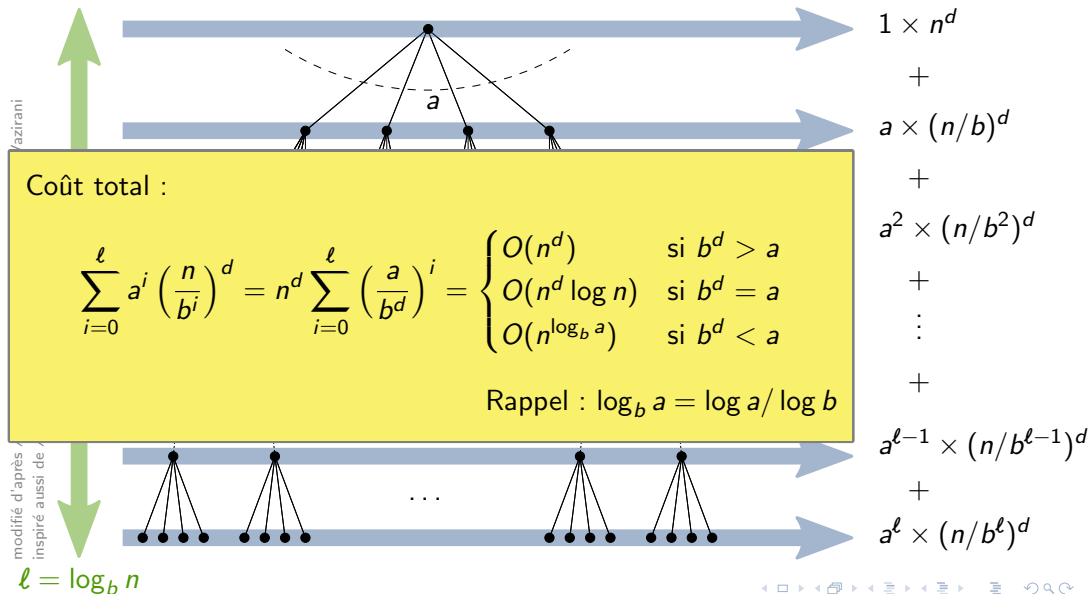
Preuve graphique

modifié d'après *Algorithms* de Dasgupta, Papadimitriou, Vazirani
inspiré aussi de *Algorithms* de J. Erickson

$$\ell = \log_b n$$



Preuve graphique



En pratique

- ▶ Versions plus générales du « *master theorem* »
 - ▶ Récurrences plus générales
 - ▶ Constantes des « grands O »
 - ▶ Termes de plus bas degré

En pratique

- ▶ Versions plus générales du « *master theorem* »
 - ▶ Récurrences plus générales
 - ▶ Constantes des « grands O »
 - ▶ Termes de plus bas degré
- ▶ Dans ce cours : étude de plusieurs exemples
 - ▶ Utilisation autorisée du « *master theorem* »
 - ▶ ... **donc à apprendre !**

En pratique

- ▶ Versions plus générales du « *master theorem* »
 - ▶ Récurrences plus générales
 - ▶ Constantes des « grands O »
 - ▶ Termes de plus bas degré
- ▶ Dans ce cours : étude de plusieurs exemples
 - ▶ Utilisation autorisée du « *master theorem* »
 - ▶ ... **donc à apprendre !**

Objectifs :

- ▶ Savoir tenter un « diviser pour régner »
- ▶ Savoir analyser sa complexité
- ▶ Reconnaître un algo. « diviser pour régner »

1. Premier exemple : tri fusion
2. Qu'est-ce que « diviser pour régner » ?
3. Deuxième exemple : multiplication d'entiers
4. Exemple spécial : Calcul de rang

Retour à l'école primaire

 **On se place dans le modèle RAM**

Multiplication d'entiers

Entrée Deux entiers A et B écrits en base 10

Sortie L'entier $C = A \times B$, en base 10

Retour à l'école primaire

! On se place dans le modèle RAM

Multiplication d'entiers

Entrée Deux entiers A et B écrits en base 10

Sortie L'entier $C = A \times B$, en base 10

$$\begin{array}{r} 1382 \\ \times 7634 \\ \hline 5528 \\ + 4146 \\ + 8292 \\ + 9674 \\ \hline 10550188 \end{array}$$

! On se place dans le modèle RAM

Multiplication d'entiers

Entrée Deux entiers A et B écrits en base 10

Sortie L'entier $C = A \times B$, en base 10

Question

- Combien de **multiplications chiffre à chiffre** sont effectuées ?
- Combien d'**additions chiffre à chiffre** sont effectuées ?

$$\begin{array}{r} 1382 \\ \times 7634 \\ \hline 5528 \\ + 4146 \\ + 8292 \\ + 9674 \\ \hline 10550188 \end{array}$$

! On se place dans le modèle RAM

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

$$A = 1382, B = 7634$$

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0, C_{01} = A_0 \times B_1$
 $C_{10} = A_1 \times B_0, C_{11} = A_1 \times B_1$

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

$$C_{00} = 2788, C_{01} = 6232$$

$$C_{10} = 442, C_{11} = 988$$

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0, C_{01} = A_0 \times B_1$
 $C_{10} = A_1 \times B_0, C_{11} = A_1 \times B_1$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor} (C_{01} + C_{10}) + 10^{2\lfloor n/2 \rfloor} C_{11}$

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

$$C_{00} = 2788, C_{01} = 6232$$

$$C_{10} = 442, C_{11} = 988$$

$$\begin{aligned} C &= 2788 + 100 \cdot (6232 \\ &\quad + 442) + 10000 \cdot 988 \\ &= 10550188 \end{aligned}$$

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0, C_{01} = A_0 \times B_1$
 $C_{10} = A_1 \times B_0, C_{11} = A_1 \times B_1$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor} (C_{01} + C_{10}) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Preuve de correction : $AB = A_0 B_0 + 10^{\lfloor n/2 \rfloor} (A_0 B_1 + A_1 B_0) + 10^{2\lfloor n/2 \rfloor} A_1 B_1$ ■

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

$$C_{00} = 2788, C_{01} = 6232$$

$$C_{10} = 442, C_{11} = 988$$

$$\begin{aligned} C &= 2788 + 100 \cdot (6232 \\ &\quad + 442) + 10000 \cdot 988 \\ &= 10550188 \end{aligned}$$

Première tentative

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0, C_{01} = A_0 \times B_1$
 $C_{10} = A_1 \times B_0, C_{11} = A_1 \times B_1$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor} (C_{01} + C_{10}) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Preuve de correction : $AB = A_0 B_0 + 10^{\lfloor n/2 \rfloor} (A_0 B_1 + A_1 B_0) + 10^{2\lfloor n/2 \rfloor} A_1 B_1$ ■

Preuve de complexité : $T(n) \leq 4T(\lceil n/2 \rceil) + O(n)$

► $a = 4, b = 2, d = 1 : b^d < a$

► $T(n) = O(n^{\log a / \log b}) = O(n^{\log 4 / \log 2}) = O(n^2) \dots$ ■

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

$$C_{00} = 2788, C_{01} = 6232$$

$$C_{10} = 442, C_{11} = 988$$

$$\begin{aligned} C &= 2788 + 100 \cdot (6232 \\ &\quad + 442) + 10000 \cdot 988 \\ &= 10550188 \end{aligned}$$

Idée de Karatsuba¹ (version Knuth)

$$A_0B_1 + A_1B_0 = A_0B_0 + A_1B_1 - (A_0 - A_1)(B_0 - B_1)$$

Idée de Karatsuba¹ (version Knuth)

$$A_0B_1 + A_1B_0 = A_0B_0 + A_1B_1 - (A_0 - A_1)(B_0 - B_1)$$

- A_0B_0 et A_1B_1 sont calculés de toute façon \rightsquigarrow un seul produit en plus !

Idée de Karatsuba¹ (version Knuth)

$$A_0B_1 + A_1B_0 = A_0B_0 + A_1B_1 - (A_0 - A_1)(B_0 - B_1)$$

- ▶ A_0B_0 et A_1B_1 sont calculés de toute façon \rightsquigarrow un seul produit en plus !
- ▶ $A_0 - A_1$ possède (env.) $n/2$ chiffres... mais peut être négatif \rightsquigarrow règle des signes

×	+	-
+	+	-
-	-	+

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

$$A = 1382, B = 7634$$

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0$, $C_{11} = A_1 \times B_1$
 $D = (A_0 - A_1) \times (B_0 - B_1)$

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

$$C_{00} = 2788, C_{11} = 988$$

$$D = 69 \times (-42) = -2898$$

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0$, $C_{11} = A_1 \times B_1$
 $D = (A_0 - A_1) \times (B_0 - B_1)$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor} (C_{00} + C_{11} - D) + 10^{2\lfloor n/2 \rfloor} C_{11}$

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

$$C_{00} = 2788, C_{11} = 988$$

$$D = 69 \times (-42) = -2898$$

$$\begin{aligned} C &= 2788 + 100 \cdot (2788 \\ &\quad + 988 + 2898) + 10000 \cdot 988 \\ &= 10550188 \end{aligned}$$

Algorithme de Karatsuba (1962)

Entrée $A = \sum_{i=0}^{n-1} a_i 10^i$ et $B = \sum_{i=0}^{n-1} b_i 10^i$

Diviser $A = A_0 + 10^{\lfloor n/2 \rfloor} A_1$
 $B = B_0 + 10^{\lfloor n/2 \rfloor} B_1$

Récursion $C_{00} = A_0 \times B_0$, $C_{11} = A_1 \times B_1$
 $D = (A_0 - A_1) \times (B_0 - B_1)$

Combiner $C = C_{00} + 10^{\lfloor n/2 \rfloor} (C_{00} + C_{11} - D) + 10^{2\lfloor n/2 \rfloor} C_{11}$

$$A = 1382, B = 7634$$

$$A_1 = 13, A_0 = 82$$

$$B_1 = 76, B_0 = 34$$

$$C_{00} = 2788, C_{11} = 988$$

$$D = 69 \times (-42) = -2898$$

$$\begin{aligned} C &= 2788 + 100 \cdot (2788 \\ &\quad + 988 + 2898) + 10000 \cdot 988 \\ &= 10550188 \end{aligned}$$

Algorithme : KARATSUBA(A, B)

si A et B n'ont qu'un chiffre : **renvoyer** $a_0 b_0$

Écrire A sous la forme $A_0 + 10^{\lfloor n/2 \rfloor} A_1$

Écrire B sous la forme $B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$C_{00} \leftarrow \text{KARATSUBA}(A_0, B_0)$

$C_{11} \leftarrow \text{KARATSUBA}(A_1, B_1)$

$D \leftarrow \text{KARATSUBA}(|A_0 - A_1|, |B_0 - B_1|)$

$s \leftarrow \text{signe}(A_0 - A_1) \times \text{signe}(B_0 - B_1)$

renvoyer $C_{00} + 10^{\lfloor n/2 \rfloor} (C_{00} + C_{11} - sD) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Propriétés de l'algorithme de Karatsuba

Algorithme : KARATSUBA(A, B)

si A et B n'ont qu'un chiffre : **renvoyer** $a_0 b_0$

Écrire A sous la forme $A_0 + 10^{\lfloor n/2 \rfloor} A_1$

Écrire B sous la forme $B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$C_{00} \leftarrow \text{KARATSUBA}(A_0, B_0)$

$C_{11} \leftarrow \text{KARATSUBA}(A_1, B_1)$

$D \leftarrow \text{KARATSUBA}(|A_0 - A_1|, |B_0 - B_1|)$

$s \leftarrow \text{signe}(A_0 - A_1) \times \text{signe}(B_0 - B_1)$

renvoyer $C_{00} + 10^{\lfloor n/2 \rfloor} (C_{00} + C_{11} - sD) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Lemme (correction)

$$A \times B = C_{00} + 10^{\lfloor n/2 \rfloor} (C_{00} + C_{11} - sD) + 10^{2\lfloor n/2 \rfloor} C_{11}$$

Preuve : $C_{00} + C_{11} - sD = A_0 B_1 + A_1 B_0$

Corollaire

L'algorithme KARATSUBA est correct.

Propriétés de l'algorithme de Karatsuba

Algorithme : KARATSUBA(A, B)

si A et B n'ont qu'un chiffre : **renvoyer** $a_0 b_0$

Écrire A sous la forme $A_0 + 10^{\lfloor n/2 \rfloor} A_1$

Écrire B sous la forme $B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$C_{00} \leftarrow \text{KARATSUBA}(A_0, B_0)$

$C_{11} \leftarrow \text{KARATSUBA}(A_1, B_1)$

$D \leftarrow \text{KARATSUBA}(|A_0 - A_1|, |B_0 - B_1|)$

$s \leftarrow \text{signe}(A_0 - A_1) \times \text{signe}(B_0 - B_1)$

renvoyer $C_{00} + 10^{\lfloor n/2 \rfloor} (C_{00} + C_{11} - sD) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Lemme (terminaison)

Pour $n > 1$, $|A_0 - A_1|$ et $|B_0 - B_1|$ ont $< n$ chiffres.

Preuve : $-10^{\lceil n/2 \rceil} \leq A_0 - A_1 \leq 10^{\lfloor n/2 \rfloor}$ et $\lceil n/2 \rceil < n$ pour $n > 1$.

Corollaire

L'algorithme KARATSUBA termine.

Preuve : appels récursifs sur des entiers **strictement plus petits**

Propriétés de l'algorithme de Karatsuba

Algorithme : KARATSUBA(A, B)

si A et B n'ont qu'un chiffre : **renvoyer** $a_0 b_0$

Écrire A sous la forme $A_0 + 10^{\lfloor n/2 \rfloor} A_1$

Écrire B sous la forme $B_0 + 10^{\lfloor n/2 \rfloor} B_1$

$C_{00} \leftarrow \text{KARATSUBA}(A_0, B_0)$

$C_{11} \leftarrow \text{KARATSUBA}(A_1, B_1)$

$D \leftarrow \text{KARATSUBA}(|A_0 - A_1|, |B_0 - B_1|)$

$s \leftarrow \text{signe}(A_0 - A_1) \times \text{signe}(B_0 - B_1)$

renvoyer $C_{00} + 10^{\lfloor n/2 \rfloor} (C_{00} + C_{11} - sD) + 10^{2\lfloor n/2 \rfloor} C_{11}$

Lemme (complexité)

Soit $K(n)$ le temps de calcul de KARATSUBA pour des entrées de taille n . Alors
 $K(n) \leq 3K(\lceil n/2 \rceil) + O(n)$

Corollaire (*master theorem*)

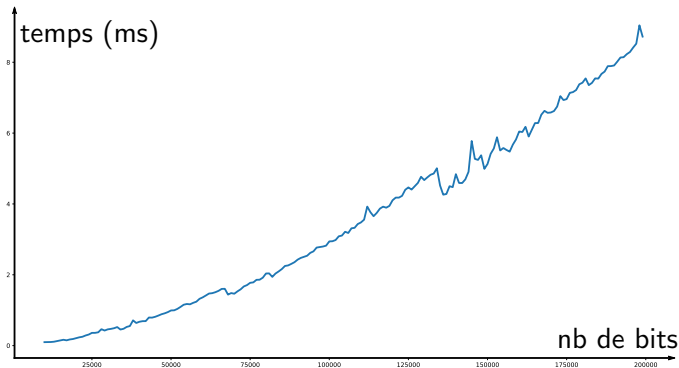
$$K(n) = O(n^{\log 3 / \log 2}) = \mathbf{O}(n^{\log 3}) \simeq O(n^{1,58})$$

Dans la *vraie* vie

- ▶ Base 10 \rightsquigarrow bases 2^{32} , 2^{64} , ...
 - ▶ Grands entiers représentés comme liste d'entiers de taille k bits \iff entiers écrits en base 2^k !
 - ▶ Exemple : `gmp` (C/C++), `BigInteger` (Java), `int` (Python)

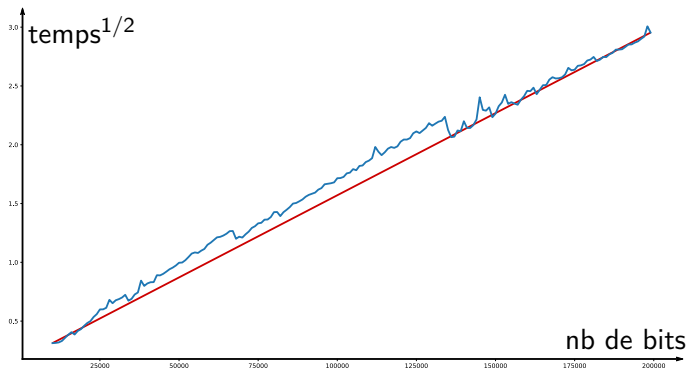
Dans la vraie vie

- ▶ Base 10 \rightsquigarrow bases 2^{32} , 2^{64} , ...
 - ▶ Grands entiers représentés comme liste d'entiers de taille k bits \iff entiers écrits en base 2^k !
 - ▶ Exemple : gmp (C/C++), BigInteger (Java), int (Python)



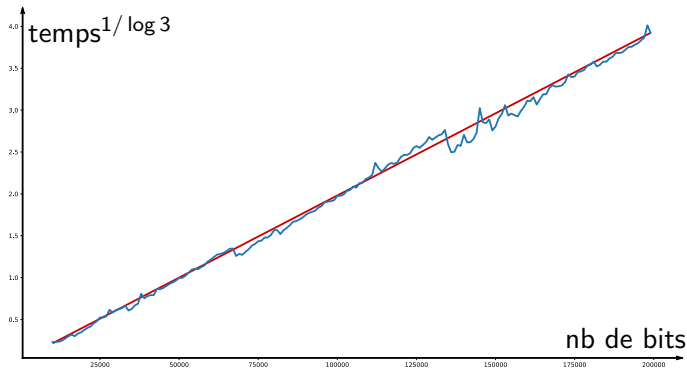
Dans la vraie vie

- ▶ Base 10 \rightsquigarrow bases 2^{32} , 2^{64} , ...
 - ▶ Grands entiers représentés comme liste d'entiers de taille k bits \iff entiers écrits en base 2^k !
 - ▶ Exemple : gmp (C/C++), BigInteger (Java), int (Python)



Dans la vraie vie

- ▶ Base 10 \rightsquigarrow bases 2^{32} , 2^{64} , ...
 - ▶ Grands entiers représentés comme liste d'entiers de taille k bits \iff entiers écrits en base 2^k !
 - ▶ Exemple : gmp (C/C++), BigInteger (Java), int (Python)



Dans la *vraie* vie

- ▶ Base 10 \rightsquigarrow bases 2^{32} , 2^{64} , ...
 - ▶ Grands entiers représentés comme liste d'entiers de taille k bits \iff entiers écrits en base 2^k !
 - ▶ Exemple : gmp (C/C++), BigInteger (Java), int (Python)
- ▶ Remarque par rapport au modèle
 - ▶ Modèle du cours (word RAM) : multiplication en temps $O(1)$
 - ▶ Irréaliste pour de grands entiers
 - ▶ Modèle souvent utilisé : taille d'un registre = $O(\log n)$

Dans la *vraie* vie

- ▶ Base 10 \rightsquigarrow bases 2^{32} , 2^{64} , ...
 - ▶ Grands entiers représentés comme liste d'entiers de taille k bits \iff entiers écrits en base 2^k !
 - ▶ Exemple : gmp (C/C++), BigInteger (Java), int (Python)
- ▶ Remarque par rapport au modèle
 - ▶ Modèle du cours (word RAM) : multiplication en temps $O(1)$
 - ▶ Irréaliste pour de grands entiers
 - ▶ Modèle souvent utilisé : taille d'un registre = $O(\log n)$
- ▶ Autre utilisation : polynômes

Dans la *vraie* vie

- ▶ Base 10 \rightsquigarrow bases 2^{32} , 2^{64} , ...
 - ▶ Grands entiers représentés comme liste d'entiers de taille k bits \iff entiers écrits en base 2^k !
 - ▶ Exemple : gmp (C/C++), BigInteger (Java), int (Python)
- ▶ Remarque par rapport au modèle
 - ▶ Modèle du cours (word RAM) : multiplication en temps $O(1)$
 - ▶ Irréaliste pour de grands entiers
 - ▶ Modèle souvent utilisé : taille d'un registre = $O(\log n)$
- ▶ Autre utilisation : polynômes
- ▶ Algorithmes plus rapides (1960's)
 - ▶ Toom-3 : découpe en 3 morceaux $O(n^{1,465})$
 - ▶ Toom-Cook : découpe en r morceaux $O(n^{1+\epsilon})$
 - ▶ Algorithmes basés sur la FFT $O(n \log n \log \log n)$

Dans la vraie vie

- ▶ Base 10 \rightsquigarrow bases 2^{32} , 2^{64} , ...
 - ▶ Grands entiers représentés comme liste d'entiers de taille k bits \iff entiers écrits en base 2^k !
 - ▶ Exemple : gmp (C/C++), BigInteger (Java), int (Python)
- ▶ Remarque par rapport au modèle
 - ▶ Modèle du cours (word RAM) : multiplication en temps $O(1)$
 - ▶ Irréaliste pour de grands entiers
 - ▶ Modèle souvent utilisé : taille d'un registre = $O(\log n)$
- ▶ Autre utilisation : polynômes
- ▶ Algorithmes plus rapides (1960's)
 - ▶ Toom-3 : découpe en 3 morceaux $O(n^{1,465})$
 - ▶ Toom-Cook : découpe en r morceaux $O(n^{1+\epsilon})$
 - ▶ Algorithmes basés sur la FFT $O(n \log n \log \log n)$
 - ▶ **2021 : dernier mot ?** (utilise la FFT) $O(n \log n)$

1. Premier exemple : tri fusion
2. Qu'est-ce que « diviser pour régner » ?
3. Deuxième exemple : multiplication d'entiers
4. Exemple spécial : Calcul de rang

Définition et algorithmes \pm naïfs

Entrée Un tableau T de n nombres, et un entier $k \in \{1, \dots, n\}$

Sortie le $k^{\text{ème}}$ plus petit élément de T , noté **rang(k, T)**

Ex. : $k = 1 \rightsquigarrow \min$, $k = n \rightsquigarrow \max$, $k = n/2 \rightsquigarrow \text{médiane}$

Définition et algorithmes \pm naïfs

Entrée Un tableau T de n nombres, et un entier $k \in \{1, \dots, n\}$

Sortie le $k^{\text{ème}}$ plus petit élément de T , noté **rang**(k, T)

Ex. : $k = 1 \rightsquigarrow \min$, $k = n \rightsquigarrow \max$, $k = n/2 \rightsquigarrow \text{médiane}$

Algorithme en $O(n^2)$:

```
pour  $i = 0$  à  $n - 1$  :  
   $c = 0$   
  pour  $j = 0$  à  $n - 1$  :  
    si  $T[j] \leq T[i]$  :  $c \leftarrow c + 1$   
  si  $c = k$  : renvoyer  $T[i]$ 
```

Définition et algorithmes \pm naïfs

Entrée Un tableau T de n nombres, et un entier $k \in \{1, \dots, n\}$

Sortie le $k^{\text{ème}}$ plus petit élément de T , noté **rang**(k, T)

Ex. : $k = 1 \rightsquigarrow \min$, $k = n \rightsquigarrow \max$, $k = n/2 \rightsquigarrow \text{médiane}$

Algorithme en $O(n^2)$:

```
pour  $i = 0$  à  $n - 1$  :  
   $c = 0$   
  pour  $j = 0$  à  $n - 1$  :  
    si  $T[j] \leq T[i]$  :  $c \leftarrow c + 1$   
  si  $c = k$  : renvoyer  $T[i]$ 
```

Algorithme en $O(n \log n)$:

```
Trier  $T$   
renvoyer  $T_{[k-1]}$ 
```


Définition et algorithmes \pm naïfs

Entrée Un tableau T de n nombres, et un entier $k \in \{1, \dots, n\}$

Sortie le $k^{\text{ème}}$ plus petit élément de T , noté **rang**(k, T)

Ex. : $k = 1 \rightsquigarrow \min$, $k = n \rightsquigarrow \max$, $k = n/2 \rightsquigarrow \text{médiane}$

Algorithme en $O(n^2)$:

```
pour  $i = 0$  à  $n - 1$  :  
   $c = 0$   
  pour  $j = 0$  à  $n - 1$  :  
    si  $T[j] \leq T[i]$  :  $c \leftarrow c + 1$   
  si  $c = k$  : renvoyer  $T[i]$ 
```

Algorithme en $O(n \log n)$:

```
Trier  $T$   
renvoyer  $T_{[k-1]}$ 
```

Algorithme en $O(n)$?

Stratégie « diviser pour régner »

Diviser Choisir un **pivot** $p \in T$ pour séparer T en trois :

- ▶ T_{inf} contient les éléments x de T vérifiant $x < p$,
- ▶ T_{eq} contient les éléments x de T vérifiant $x = p$,
- ▶ T_{sup} contient les éléments x de T vérifiant $x > p$,

$$n_{\text{inf}} = |T_{\text{inf}}|$$

$$n_{\text{eq}} = |T_{\text{eq}}|$$

$$n_{\text{sup}} = |T_{\text{sup}}|$$

Stratégie « diviser pour régner »

Diviser Choisir un **pivot** $p \in T$ pour séparer T en trois :

- ▶ T_{inf} contient les éléments x de T vérifiant $x < p$,
- ▶ T_{eq} contient les éléments x de T vérifiant $x = p$,
- ▶ T_{sup} contient les éléments x de T vérifiant $x > p$,

$$n_{\text{inf}} = |T_{\text{inf}}|$$

$$n_{\text{eq}} = |T_{\text{eq}}|$$

$$n_{\text{sup}} = |T_{\text{sup}}|$$

Exemple

$T =$

3	7	21	9	12	16	7	4	1	2
---	---	----	---	----	----	---	---	---	---

pivot : $T_{[1]} = 7$

Stratégie « diviser pour régner »

Diviser Choisir un **pivot** $p \in T$ pour séparer T en trois :

- ▶ T_{inf} contient les éléments x de T vérifiant $x < p$,
- ▶ T_{eq} contient les éléments x de T vérifiant $x = p$,
- ▶ T_{sup} contient les éléments x de T vérifiant $x > p$,

$$\begin{aligned}n_{\text{inf}} &= |T_{\text{inf}}| \\n_{\text{eq}} &= |T_{\text{eq}}| \\n_{\text{sup}} &= |T_{\text{sup}}|\end{aligned}$$

Exemple

$T =$

3	7	21	9	12	16	7	4	1	2
---	---	----	---	----	----	---	---	---	---

pivot : $T_{[1]} = 7$

Stratégie « diviser pour régner »

Diviser Choisir un **pivot** $p \in T$ pour séparer T en trois :

- ▶ T_{inf} contient les éléments x de T vérifiant $x < p$,
- ▶ T_{eq} contient les éléments x de T vérifiant $x = p$,
- ▶ T_{sup} contient les éléments x de T vérifiant $x > p$,

$$\begin{aligned}n_{\text{inf}} &= |T_{\text{inf}}| \\n_{\text{eq}} &= |T_{\text{eq}}| \\n_{\text{sup}} &= |T_{\text{sup}}|\end{aligned}$$

Exemple

$T =$

3	7	21	9	12	16	7	4	1	2
---	---	----	---	----	----	---	---	---	---

- ▶ $T_{\text{inf}} =$

3	4	1	2
---	---	---	---
- ▶ $T_{\text{eq}} =$

7	7
---	---
- ▶ $T_{\text{sup}} =$

21	9	12	16
----	---	----	----

pivot : $T_{[1]} = 7$

$$n_{\text{inf}} = 4$$

$$n_{\text{eq}} = 2$$

$$n_{\text{sup}} = 4$$

Du coup :

- ▶ $\text{rang}(2, T) = \text{rang}(2, T_{\text{inf}})$
- ▶ $\text{rang}(5, T) = T_{[1]}$ (pivot)
- ▶ $\text{rang}(9, T) = \text{rang}(3, T_{\text{sup}})$

Stratégie « diviser pour régner »

Diviser Choisir un **pivot** $p \in T$ pour séparer T en trois :

- ▶ T_{inf} contient les éléments x de T vérifiant $x < p$,
- ▶ T_{eq} contient les éléments x de T vérifiant $x = p$,
- ▶ T_{sup} contient les éléments x de T vérifiant $x > p$,

$$\begin{aligned}n_{\text{inf}} &= |T_{\text{inf}}| \\n_{\text{eq}} &= |T_{\text{eq}}| \\n_{\text{sup}} &= |T_{\text{sup}}|\end{aligned}$$

Récursion Trouver $\text{rang}(k, T)$ dans T_{inf} , T_{eq} ou T_{sup}

$$\text{rang}(k, T) = \begin{cases} \text{rang}(k, T_{\text{inf}}) & \text{si } k \leq n_{\text{inf}} \\ p & \text{si } n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}} \\ \text{rang}(k - n_{\text{inf}} - n_{\text{eq}}, T_{\text{sup}}) & \text{si } n_{\text{inf}} + n_{\text{eq}} < k \end{cases}$$

Combiner Rien à faire...

Stratégie « diviser pour régner »

Diviser Choisir un **pivot** $p \in T$ pour séparer T en trois :

- ▶ T_{inf} contient les éléments x de T vérifiant $x < p$,
- ▶ T_{eq} contient les éléments x de T vérifiant $x = p$,
- ▶ T_{sup} contient les éléments x de T vérifiant $x > p$,

$$\begin{aligned}n_{\text{inf}} &= |T_{\text{inf}}| \\n_{\text{eq}} &= |T_{\text{eq}}| \\n_{\text{sup}} &= |T_{\text{sup}}|\end{aligned}$$

Récursion Trouver $\text{rang}(k, T)$ dans T_{inf} , T_{eq} ou T_{sup}

$$\text{rang}(k, T) = \begin{cases} \text{rang}(k, T_{\text{inf}}) & \text{si } k \leq n_{\text{inf}} \\ p & \text{si } n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}} \\ \text{rang}(k - n_{\text{inf}} - n_{\text{eq}}, T_{\text{sup}}) & \text{si } n_{\text{inf}} + n_{\text{eq}} < k \end{cases}$$

Combiner Rien à faire...

Question importante : quel choix pour le pivot ?

L'algorithme

Algorithme : RANG(T, k)

si $|T| = 1$: renvoyer $T_{[0]}$

$$p \leftarrow \text{CHOIXPIVOT}(T)$$
$$n_{\text{inf}} \leftarrow 0, n_{\text{eq}} \leftarrow 0$$

pour $i = 0$ à $n - 1$:

```
// Calcul de  $n_{\text{inf}}$  et  $n_{\text{eq}}$ 
```

$$\mathbf{si} \ T_{[i]} < p : n_{\text{inf}} \leftarrow n_{\text{inf}} + 1$$

sinon si $T_{[i]} = p : n_{\text{eq}} \leftarrow n_{\text{eq}} + 1$

si $k \leq n_{inf}$: Calculer T_{inf} et **renvoyer** $RANG(T_{inf}, k)$

sinon si $n_{inf} < k \leq n_{inf} + n_{eq}$: renvoyer p

sinon : Calculer T_{sup} et **renvoyer** $RANG(T_{sup}, k - n_{inf} - n_{eq})$

Correction de l'algorithme

Lemme

$$\text{rang}(k, T) = \begin{cases} \text{rang}(k, T_{inf}) & \text{si } k \leq n_{inf} \\ p & \text{si } n_{inf} < k \leq n_{inf} + n_{eq} \\ \text{rang}(k - n_{inf} - n_{eq}, T_{sup}) & \text{si } n_{inf} + n_{eq} < k \end{cases}$$

Correction de l'algorithme

Lemme

$$\text{rang}(k, T) = \begin{cases} \text{rang}(k, T_{\text{inf}}) & \text{si } k \leq n_{\text{inf}} \\ p & \text{si } n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}} \\ \text{rang}(k - n_{\text{inf}} - n_{\text{eq}}, T_{\text{sup}}) & \text{si } n_{\text{inf}} + n_{\text{eq}} < k \end{cases}$$

Preuve

Cas 1 ($k \leq n_{\text{inf}}$) : soit $r = \text{rang}(k, T_{\text{inf}})$

Si $x \in T_{\text{eq}} \cup T_{\text{sup}}$, $x > r$; et il y a k éléments $\leq r$ dans T_{inf} ; donc il y a k éléments $\leq r$ dans T

Correction de l'algorithme

Lemme

$$\text{rang}(k, T) = \begin{cases} \text{rang}(k, T_{\text{inf}}) & \text{si } k \leq n_{\text{inf}} \\ p & \text{si } n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}} \\ \text{rang}(k - n_{\text{inf}} - n_{\text{eq}}, T_{\text{sup}}) & \text{si } n_{\text{inf}} + n_{\text{eq}} < k \end{cases}$$

Preuve

Cas 1 ($k \leq n_{\text{inf}}$) : soit $r = \text{rang}(k, T_{\text{inf}})$

Si $x \in T_{\text{eq}} \cup T_{\text{sup}}$, $x > r$; et il y a k éléments $\leq r$ dans T_{inf} ; donc il y a k éléments $\leq r$ dans T

Cas 2 ($n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}}$) : soit $r = p$

Si $x \in T_{\text{sup}}$, $x > r$; et il y a $n_{\text{inf}} < k$ éléments $< r$ dans T_{inf} , et n_{eq} éléments égaux à r dans T_{eq} ; donc $\text{rang}(k, T) \in T_{\text{eq}}$

Correction de l'algorithme

Lemme

$$\text{rang}(k, T) = \begin{cases} \text{rang}(k, T_{\text{inf}}) & \text{si } k \leq n_{\text{inf}} \\ p & \text{si } n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}} \\ \text{rang}(k - n_{\text{inf}} - n_{\text{eq}}, T_{\text{sup}}) & \text{si } n_{\text{inf}} + n_{\text{eq}} < k \end{cases}$$

Preuve

Cas 1 ($k \leq n_{\text{inf}}$) : soit $r = \text{rang}(k, T_{\text{inf}})$

Si $x \in T_{\text{eq}} \cup T_{\text{sup}}$, $x > r$; et il y a k éléments $\leq r$ dans T_{inf} ; donc il y a k éléments $\leq r$ dans T

Cas 2 ($n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}}$) : soit $r = p$

Si $x \in T_{\text{sup}}$, $x > r$; et il y a $n_{\text{inf}} < k$ éléments $< r$ dans T_{inf} , et n_{eq} éléments égaux à r dans T_{eq} ; donc $\text{rang}(k, T) \in T_{\text{eq}}$

Cas 3 ($n_{\text{inf}} + n_{\text{eq}} < k$) soit $r = \text{rang}(k - n_{\text{inf}} - n_{\text{eq}}, T_{\text{sup}})$

il y a $n_{\text{inf}} + n_{\text{eq}} < k$ éléments $< r$ dans $T_{\text{inf}} \cup T_{\text{eq}}$; il y a $k - n_{\text{inf}} - n_{\text{eq}}$ éléments $\leq r$ dans T_{sup} ; donc $k - n_{\text{inf}} - n_{\text{eq}} + (n_{\text{inf}} + n_{\text{eq}}) = k$ éléments $\leq r$ au total

Complexité

Algorithme : $\text{RANG}(T, k)$

si $k = 1$: renvoyer $T_{[0]}$

$$p \leftarrow \text{CHOIXPIVOT}(T)$$

Calculer n_{inf} et n_{eq}

si $k \leq n_{inf}$: renvoyer $RANG(T_{inf}, k)$

si $n_{inf} < k \leq n_{inf} + n_{eq}$: renvoyer p

sinon : renvoyer $RANG(T_{sup}, k - n_{inf} - n_{eq})$

Complexité

Algorithme : $\text{RANG}(T, k)$

si $k = 1$: **renvoyer** $T_{[0]}$

$p \leftarrow \text{CHOIXPIVOT}(T)$

Calculer n_{inf} et n_{eq}

si $k \leq n_{\text{inf}}$: **renvoyer** $\text{RANG}(T_{\text{inf}}, k)$

si $n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}}$: **renvoyer** p

sinon : **renvoyer** $\text{RANG}(T_{\text{sup}}, k - n_{\text{inf}} - n_{\text{eq}})$

► Calculer n_{inf} , n_{eq} , T_{inf} , T_{sup} : $O(n)$

Complexité

Algorithme : $\text{RANG}(T, k)$

si $k = 1$: **renvoyer** $T_{[0]}$

$p \leftarrow \text{CHOIXPIVOT}(T)$

Calculer n_{inf} et n_{eq}

si $k \leq n_{\text{inf}}$: **renvoyer** $\text{RANG}(T_{\text{inf}}, k)$

si $n_{\text{inf}} < k \leq n_{\text{inf}} + n_{\text{eq}}$: **renvoyer** p

sinon : **renvoyer** $\text{RANG}(T_{\text{sup}}, k - n_{\text{inf}} - n_{\text{eq}})$

- Calculer n_{inf} , n_{eq} , T_{inf} , T_{sup} : $O(n)$
- Un appel récursif (au pire) sur T_{inf} ou T_{sup} , de taille n' :

$$t(n) = t(n') + O(n)$$

Complexité

Algorithme : RANG(T, k)

si $k = 1$: renvoyer $T_{[0]}$

$$p \leftarrow \text{CHOIXPIVOT}(T)$$

Calculer n_{inf} et n_{eq}

si $k \leq n_{inf}$: renvoyer $RANG(T_{inf}, k)$

si $n_{inf} < k \leq n_{inf} + n_{eq}$: renvoyer p

sinon : renvoyer $\text{RANG}(T_{sup}, k - n_{inf} - n_{eq})$

- Calculer $n_{\text{inf}}, n_{\text{eq}}, T_{\text{inf}}, T_{\text{sup}} : O(n)$
- Un appel récursif (au pire) sur T_{inf} ou T_{sup} , de taille n' :

$$t(n) = t(n') + O(n)$$

- ▶ Cas idéal : $n' = n/2 \rightsquigarrow t(n) = O(n)$ (master theorem)
- ▶ Pire cas : $n' = n - 1 \rightsquigarrow t(n) = O(n^2)$ (à la main)

Complexité

Algorithme : RANG(T, k)

si $k = 1$: renvoyer $T_{[0]}$

$$p \leftarrow \text{CHOIXPIVOT}(T)$$

Calculer n_{inf} et n_{eq}

si $k \leq n_{inf}$: renvoyer $RANG(T_{inf}, k)$

si $n_{inf} < k \leq n_{inf} + n_{eq}$: renvoyer p

sinon : renvoyer $\text{RANG}(T_{sup}, k - n_{inf} - n_{eq})$

- Calculer $n_{\text{inf}}, n_{\text{eq}}, T_{\text{inf}}, T_{\text{sup}} : O(n)$
- Un appel récursif (au pire) sur T_{inf} ou T_{sup} , de taille n' :

$$t(n) = t(n') + O(n)$$

- ▶ Cas idéal : $n' = n/2 \rightsquigarrow t(n) = O(n)$ (master theorem)
- ▶ Pire cas : $n' = n - 1 \rightsquigarrow t(n) = O(n^2)$ (à la main)

But : choix de pivot pour minimiser n' !

Choix du pivot (1)

Algorithme : CHOIXPIVOT(T)

renvoyer $T_{[0]}$

Choix du pivot (1)

Algorithme : CHOIXPIVOT(T)

renvoyer $T_{[0]}$

Complexité

► Cas le pire : tableau initialement trié

↪ $n' = n - 1$, donc $t(n) = O(n^2)$

Choix du pivot (1)

Algorithme : CHOIXPIVOT(T)

renvoyer $T_{[0]}$

Complexité

► Cas le pire : tableau initialement trié

↪ $n' = n - 1$, donc $t(n) = O(n^2)$

► Si tableau aléatoire : on peut montrer que $\mathbb{E}[n'] = n/2$

↪ $t(n) = O(n)$ « en moyenne »

Choix du pivot (1)

Algorithme : CHOIXPIVOT(T)

renvoyer $T_{[0]}$

Complexité

► Cas le pire : tableau initialement trié

↪ $n' = n - 1$, donc $t(n) = O(n^2)$

► Si tableau aléatoire : on peut montrer que $\mathbb{E}[n'] = n/2$

↪ $t(n) = O(n)$ « en moyenne »

Choix correct si les tableaux sont aléatoires, mais en pratique ce n'est rarement le cas !

Choix du pivot (2)

Algorithme : CHOIXPIVOT(T)

$$j \leftarrow \text{entier aléatoire entre } 0 \text{ et } n-1$$
renvoyer $T_{[j]}$

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
             // guaranteed to be random.
}
```

<https://xkcd.com/221/>

Choix du pivot (2)

Algorithme : CHOIXPIVOT(T)

$$j \leftarrow \text{entier aléatoire entre } 0 \text{ et } n - 1$$
renvoyer $T_{[j]}$

```
int getRandomNumber()
{
    return 4; // chosen by fair dice roll.
              // guaranteed to be random.
}
```

<https://xkcd.com/221/>

Complexité

- ▶ Cas le pire : si on manque de chance

$\rightsquigarrow n' = n - 1$, donc $t(n) = O(n^2)$

Choix du pivot (2)

Algorithme : CHOIXPIVOT(T)

$j \leftarrow$ entier aléatoire entre 0 et $n - 1$

renvoyer $T[j]$

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

<https://xkcd.com/221/>

Complexité

► Cas le pire : si on manque de chance

↪ $n' = n - 1$, donc $t(n) = O(n^2)$

► Mais avec probabilité $1/2$: $n' \geq n/4$

↪ $t(n) = O(n)$ avec « bonne probabilité »



Choix du pivot (2)

Algorithme : CHOIXPIVOT(T)

$j \leftarrow$ entier aléatoire entre 0 et $n - 1$

renvoyer $T[j]$

```
int getRandomNumber()  
{  
    return 4; // chosen by fair dice roll.  
             // guaranteed to be random.  
}
```

<https://xkcd.com/221/>

Complexité

► Cas le pire : si on manque de chance

↪ $n' = n - 1$, donc $t(n) = O(n^2)$

► Mais avec probabilité $1/2$: $n' \geq n/4$

↪ $t(n) = O(n)$ avec « bonne probabilité »



Bon choix, quelque soit le tableau, mais difficile à analyser

Choix du pivot (3)

Algorithme : CHOIXPIVOT(T)

$j \leftarrow$ entier aléatoire entre 0 et $n - 1$

Calculer n_{inf} et n_{sup} avec pivot $T[j]$

si $n_{\text{inf}} \leq 3n/4$ et $n_{\text{sup}} \leq 3n/4$: **renvoyer** $T[j]$

sinon : **renvoyer** CHOIXPIVOT(T)

Choix du pivot (3)

Algorithme : CHOIXPIVOT(T)

$j \leftarrow$ entier aléatoire entre 0 et $n - 1$

Calculer n_{inf} et n_{sup} avec pivot $T[j]$

si $n_{\text{inf}} \leq 3n/4$ et $n_{\text{sup}} \leq 3n/4$: **renvoyer** $T[j]$

sinon : **renvoyer** CHOIXPIVOT(T)

Complexité

- ▶ Cas le pire : $n' = 3n/4$
- ▶ Coût de CHOIXPIVOT : $O(n)$ fois le nombre d'essais
- ▶ En moyenne 2 tentatives pour j car réussite avec proba. $1/2$

Choix du pivot (3)

Algorithme : CHOIXPIVOT(T)

$j \leftarrow$ entier aléatoire entre 0 et $n - 1$

Calculer n_{inf} et n_{sup} avec pivot $T[j]$

si $n_{\text{inf}} \leq 3n/4$ et $n_{\text{sup}} \leq 3n/4$: **renvoyer** $T[j]$

sinon : **renvoyer** CHOIXPIVOT(T)

Complexité

- ▶ Cas le pire : $n' = 3n/4$
- ▶ Coût de CHOIXPIVOT : $O(n)$ fois le nombre d'essais
- ▶ En moyenne 2 tentatives pour j car réussite avec proba. $1/2$
- ▶ $t(n) \leq t(3n/4) + O(n) \rightsquigarrow t(n) = O(n)$

(*master thm*)

Choix du pivot (3)

Algorithme : CHOIXPIVOT(T)

$j \leftarrow$ entier aléatoire entre 0 et $n - 1$

Calculer n_{inf} et n_{sup} avec pivot $T[j]$

si $n_{\text{inf}} \leq 3n/4$ et $n_{\text{sup}} \leq 3n/4$: **renvoyer** $T[j]$

sinon : **renvoyer** CHOIXPIVOT(T)

Complexité

- ▶ Cas le pire : $n' = 3n/4$
- ▶ Coût de CHOIXPIVOT : $O(n)$ fois le nombre d'essais
- ▶ En moyenne 2 tentatives pour j car réussite avec proba. $1/2$
- ▶ $t(n) \leq t(3n/4) + O(n) \rightsquigarrow t(n) = O(n)$ (master thm)

Bon choix en théorie, *facile* à analyser. En pratique, préférer le précédent !

Choix du pivot (bonus!)

Algorithme : CHOIXPIVOT(T)

pour $i = 0$ à $\lceil n/5 \rceil - 1$:

$m_i \leftarrow \text{MEDIANE}(T_{[5i, 5i+5[})$

renvoyer $\text{MEDIANE}([m_0, \dots, m_{\lceil n/5 \rceil - 1}])$

$(\text{MEDIANE}(T) = \text{RANG}(T, \lfloor n/2 \rfloor))$

Choix du pivot (bonus!)

Algorithme : CHOIXPIVOT(T)

pour $i = 0$ à $\lceil n/5 \rceil - 1$:

$m_i \leftarrow \text{MEDIANE}(T_{[5i, 5i+5[})$

renvoyer $\text{MEDIANE}([m_0, \dots, m_{\lceil n/5 \rceil - 1}])$

$(\text{MEDIANE}(T) = \text{RANG}(T, \lfloor n/2 \rfloor))$

Complexité

- ▶ Cas le pire : on peut montrer que $n' \leq 7n/10 + 6$
- ▶ Coût de CHOIXPIVOT : $O(n) + t(\lceil n/5 \rceil)$

(pas si facile!)

Choix du pivot (bonus!)

Algorithme : CHOIXPIVOT(T)

pour $i = 0$ à $\lceil n/5 \rceil - 1$:

$m_i \leftarrow \text{MEDIANE}(T_{[5i, 5i+5[})$

renvoyer $\text{MEDIANE}([m_0, \dots, m_{\lceil n/5 \rceil - 1}])$

($\text{MEDIANE}(T) = \text{RANG}(T, \lfloor n/2 \rfloor)$)

Complexité

- ▶ Cas le pire : on peut montrer que $n' \leq 7n/10 + 6$ *(pas si facile!)*
- ▶ Coût de CHOIXPIVOT : $O(n) + t(\lceil n/5 \rceil)$
- ▶ $t(n) \leq t(7n/10 + 6) + t(\lceil n/5 \rceil) + O(n) \rightsquigarrow t(n) = O(n)$ *(récurrence pas si facile!)*

Choix du pivot (bonus!)

Algorithme : CHOIXPIVOT(T)

pour $i = 0$ à $\lceil n/5 \rceil - 1$:

$m_i \leftarrow \text{MEDIANE}(T_{[5i, 5i+5[})$

renvoyer $\text{MEDIANE}([m_0, \dots, m_{\lceil n/5 \rceil - 1}])$

($\text{MEDIANE}(T) = \text{RANG}(T, \lfloor n/2 \rfloor)$)

Complexité

- ▶ Cas le pire : on peut montrer que $n' \leq 7n/10 + 6$ *(pas si facile!)*
- ▶ Coût de CHOIXPIVOT : $O(n) + t(\lceil n/5 \rceil)$
- ▶ $t(n) \leq t(7n/10 + 6) + t(\lceil n/5 \rceil) + O(n) \rightsquigarrow t(n) = O(n)$ *(récurrence pas si facile!)*

Algorithme déterministe : optimal en théorie, moins bien en pratique !

Récapitulatif

Théorème

$\text{RANG}(T, k)$ retourne le $k^{\text{ème}}$ élément de T .

En fonction de CHOIXPIVOT, sa complexité peut être

- ▶ $O(n^2)$ dans le pire des cas mais $O(n)$ « en moyenne »
- ▶ $O(n)$ avec bonne probabilité, pour tout tableau
- ▶ $O(n)$ de manière déterministe, pour tout tableau

Récapitulatif

Théorème

$\text{RANG}(T, k)$ retourne le $k^{\text{ème}}$ élément de T .

En fonction de CHOIXPIVOT, sa complexité peut être

- ▶ $O(n^2)$ dans le pire des cas mais $O(n)$ « en moyenne »
- ▶ $O(n)$ avec bonne probabilité, pour tout tableau
- ▶ $O(n)$ de manière déterministe, pour tout tableau

Remarques

- ▶ Le choix de pivot $T_{[0]}$ fonctionne avec bonne probabilité si on mélange aléatoirement T au début
- ▶ Version plus complexe de cet algorithme : tri rapide (QUICKSORT)

Conclusion

Diviser-pour-régner

- ▶ Trois étapes : **diviser**, **résoudre**, **combiner**
- ▶ Analyse de complexité : équation de récurrence + *master theorem*
- ▶ Très nombreux algorithmes ! tableaux, nombres, arbres, géométrie, ...

Conclusion

Diviser-pour-régner

- ▶ Trois étapes : **diviser**, **résoudre**, **combiner**
- ▶ Analyse de complexité : équation de récurrence + *master theorem*
- ▶ Très nombreux algorithmes ! tableaux, nombres, arbres, géométrie, ...

Aperçu dans ce cours

- ▶ Analyse **en moyenne** d'algorithmes \rightsquigarrow comportement sur une entrée aléatoire
 - ▶ Algorithme **probabiliste** \rightsquigarrow choix aléatoires au cours de l'algorithme
- Voir le cours d'algorithmique de L3*