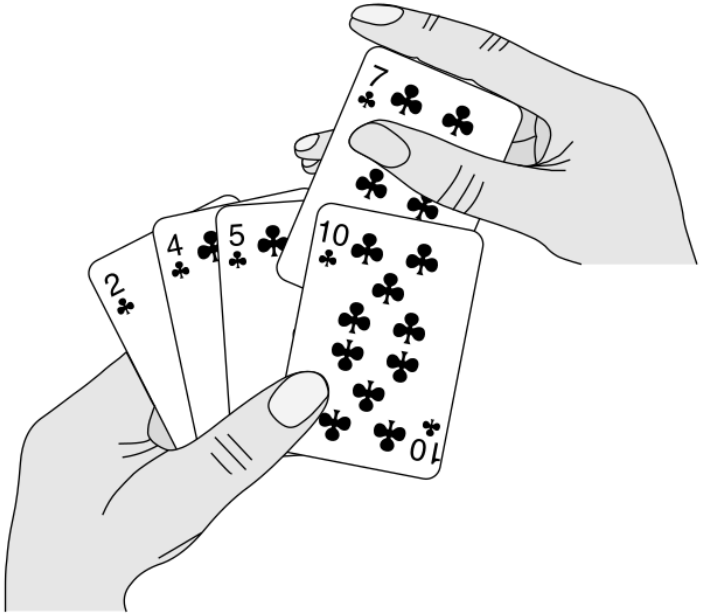
Table des Matières

Le problème du tri

Problème (Le tri)

Entrée : un tableau T d'entiers

Sortie : une permutation de T triée



Une approche naturelle

Un algorithme naturel : le tri par insertion

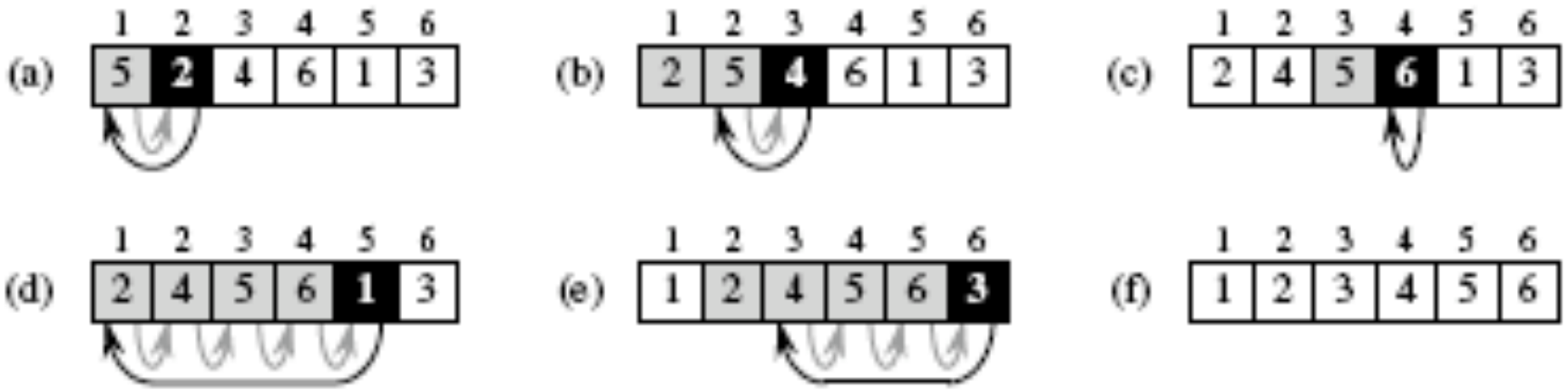


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Un algorithme naturel : le tri par insertion

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3           ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4           i ← j − 1
5           tant que i > 0 et A[i] > clé
6               faire A[i + 1] ← A[i]
7                    i ← i − 1
8           A[i + 1] ← clé
```

Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme est correct. Une fois l'invariant énoncé, il faut montrer trois choses concernant l'invariant de boucle :

- Initialisation : il est vrai avant la première itération de la boucle.
- Conservation : s'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.
- Terminaison : une fois la boucle terminée(!), l'invariant fournit une propriété utile qui aide à montrer la correction de l'algorithme.

Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme est correct. Une fois l'invariant énoncé, il faut montrer trois choses concernant l'invariant de boucle :

- Initialisation : il est vrai avant la première itération de la boucle.
- Conservation : s'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.
- Terminaison : une fois la boucle terminée(!), l'invariant fournit une propriété utile qui aide à montrer la correction de l'algorithme.

Exemple

Pour le tri par insertion :

> *"Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau* $A[1..j-1]$ *se compose des éléments qui occupaient initialement les positions* 1 *à* $j-1$, *mais qui sont maintenant triés."*

Correction du tri par insertion par invariant de boucle

Les invariants de boucle sont des propriétés qui aident à comprendre/expliquer/justifier pourquoi un algorithme est correct. Une fois l'invariant énoncé, il faut montrer trois choses concernant l'invariant de boucle :

- Initialisation : il est vrai avant la première itération de la boucle.
- Conservation : s'il est vrai avant une itération de la boucle, il le reste avant l'itération suivante.
- Terminaison : une fois la boucle terminée(!), l'invariant fournit une propriété utile qui aide à montrer la correction de l'algorithme.

Exemple

Pour le tri par insertion :

> *"Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau* $A[1..j-1]$ *se compose des éléments qui occupaient initialement les positions* 1 *à* $j-1$, *mais qui sont maintenant triés."*

Remarque

Pour les preuves de correction par invariants de boucle, il faut toujours prouver la terminaison à part : ici c'est une boucle for et pour la boucle tant que, c'est la valeur de i qui diminue et finit donc par atteindre la valeur 0.

Invariant de boucle du tri par insertion

Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés.

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

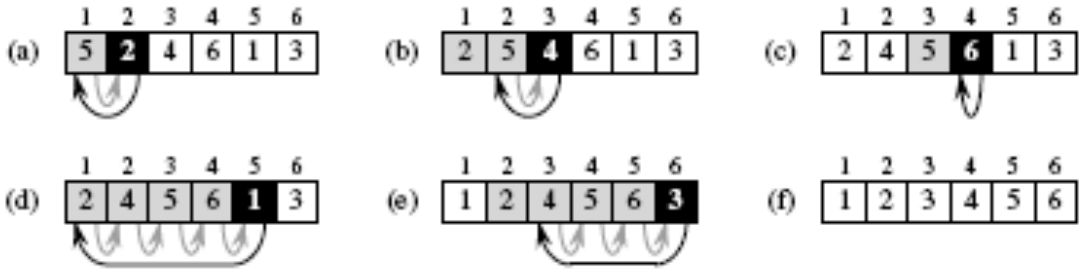


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Invariant de boucle du tri par insertion

Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés.

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

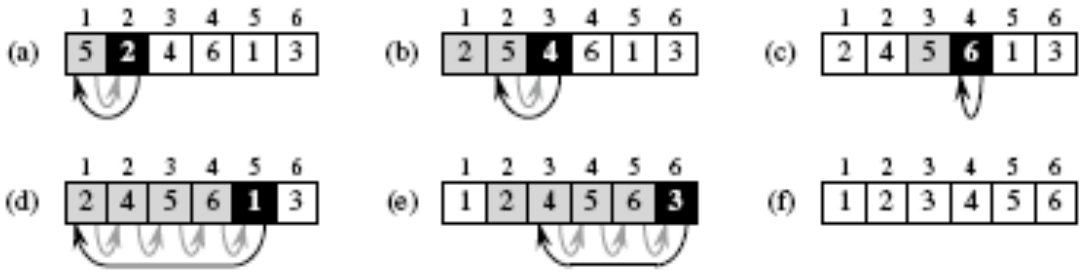


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

À l'initialisation j = 2 et le tableau $A[1..j-1]$ est $A[1]$ qui est trivialement trié.

Invariant de boucle du tri par insertion

Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ *se compose des éléments qui occupaient initialement les positions* $A[1..j-1]$ *mais qui sont maintenant triés.*

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

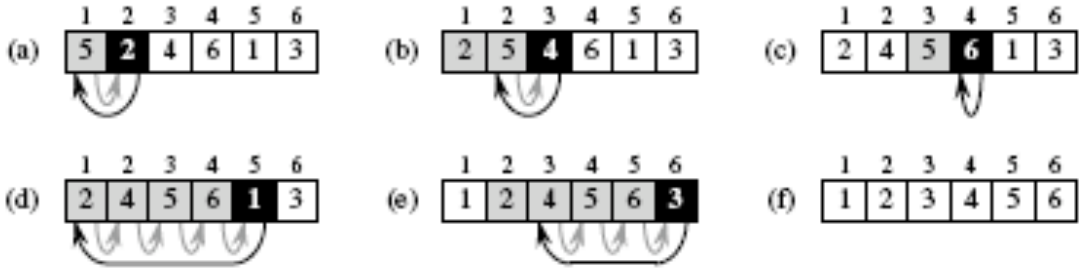


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Conservation (on pourrait être plus formel).
Le corps de la boucle "pour" extérieure fonctionne en déplaçant $A[j-1]$, $A[j-2]$, $A[j-3]$, etc. d'une position vers la droite jusqu'à ce qu'on trouve la bonne position pour A[j] (lignes 4–7).
On insère alors la valeur de A[j] (ligne 8) à la bonne place.

Invariant de boucle du tri par insertion

Au début de chaque itération de la boucle pour des lignes 1–8, le sous-tableau $A[1..j-1]$ se compose des éléments qui occupaient initialement les positions $A[1..j-1]$ mais qui sont maintenant triés.

```
TRI-INSERTION (A)
1  pour j ← 2 à longueur[A]
2      faire clé ← A[j]
3          ▷ Insère A[j] dans la suite
                triée A[1 . . j − 1].
4          i ← j − 1
5          tant que i > 0 et A[i] > clé
6              faire A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← clé
```

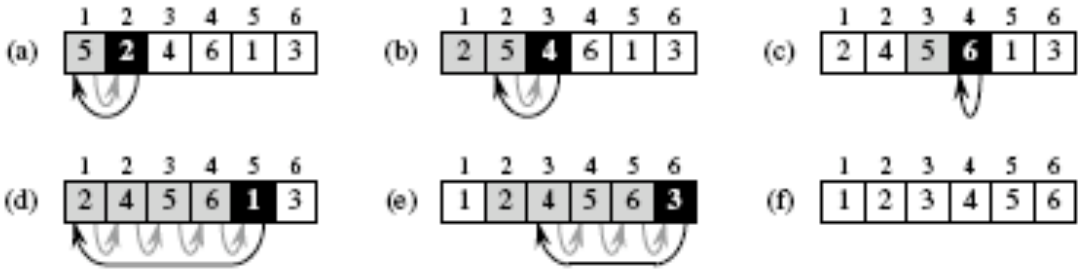


Figure 2.2 Fonctionnement de TRI-INSERTION sur le tableau $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Les indices apparaissent au-dessus des cases, les valeurs du tableau apparaissant dans les cases. (a)–(e) Itérations de la boucle pour des lignes lines 1–8. À chaque itération, la case noire renferme la clé lue dans $A[j]$; cette clé est comparée aux valeurs des cases grises situées à sa gauche (test en ligne 5). Les flèches grises montrent les déplacements des valeurs d'une position vers la droite (ligne 6), alors que les flèches noires indiquent vers où sont déplacées les clés (ligne 8). (f)Tableau trié final.

Terminaison : La boucle pour extérieure prend fin quand j dépasse n, c'est-à-dire quand $j = n + 1$.

En substituant $n + 1$ à j dans la formulation de l'invariant de boucle, on obtient :

Le sous-tableau $A[1..n]$ se compose des éléments qui appartenaient originellement à $A[1..n]$ mais qui sont maintenant triés.

Donc tout le tableau A est trié ! Par conséquent l'algorithme est correct.

Complexité du tri par insertion (dans le pire cas)

TRI-INSERTION (A)		*coût*	*fois*
1	**pour** $j \leftarrow 2$ **à** *longueur*$[A]$	c_1	n
2	**faire** *clé* $\leftarrow A[j]$	c_2	$n-1$
3	▷ Insère $A[j]$ dans la suite triée $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**tant que** $i > 0$ et $A[i] >$ *clé*	c_5	$\sum_{j=2}^{n} t_j$
6	**faire** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow$ *clé*	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Complexité du tri par insertion (dans le pire cas)

TRI-INSERTION (A)		*coût*	*fois*
1	**pour** $j \leftarrow 2$ **à** *longueur*$[A]$	c_1	n
2	**faire** *clé* $\leftarrow A[j]$	c_2	$n-1$
3	▷ Insère $A[j]$ dans la suite triée $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**tant que** $i > 0$ et $A[i] >$ *clé*	c_5	$\sum_{j=2}^{n} t_j$
6	**faire** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow$ *clé*	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) + c_7 \sum_{j=2}^{n}(t_j-1) + c_8(n-1)$$

On se place dans le cas le plus défavorable/le pire cas : le tableau A est trié en ordre décroissant et donc $t_j = j$. Or

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \qquad \text{et} \qquad \sum_{j=2}^{n}(j-1) = \frac{(n-1)n}{2}$$

Exercice

En déduire que $T(n) = O(n^2)$.

Principe de la méthode Diviser-pour-Régner

Elle améliore considérablement le temps d'exécution par "découpe", on s'attend donc à un facteur logarithmique.

Exercice

Mais qu'est-ce que le logarithme vient faire là-dedans ?

- On décompose le problème en plusieurs sous-problèmes
- On résoud chaque sous-problèmes récursivement
- On combine les solutions des sous-problèmes en un solution globale

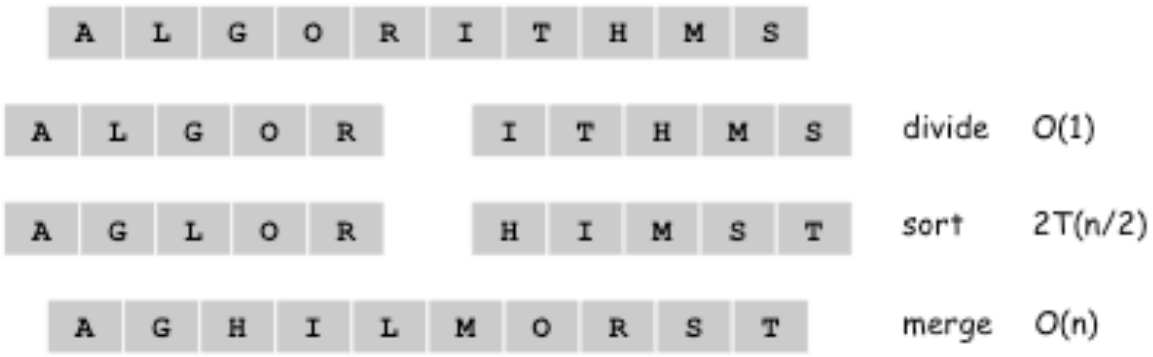
Définition

On considère TRI-FUSION$(A, 1, n)$ avec la définition :

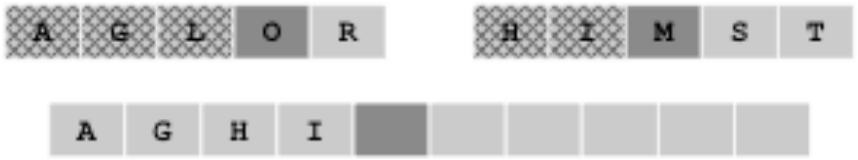
```
TRI-FUSION(A, p, r)
1  si p < r
2     alors q ← ⌊(p + r)/2⌋
3           TRI-FUSION(A, p, q)
4           TRI-FUSION(A, q + 1, r)
5           FUSION(A, p, q, r)
```

Tri fusion : un exemple

- Algorithme principal :



- La fusion :



Lemme

La fusion prend un temps linéaire.

Exercice

Donner un énoncé formel à ce lemme.

Equations de récurrence

Exemple

Pour TRI-FUSION, en supposant que nous avons démontré le lemme, on a :

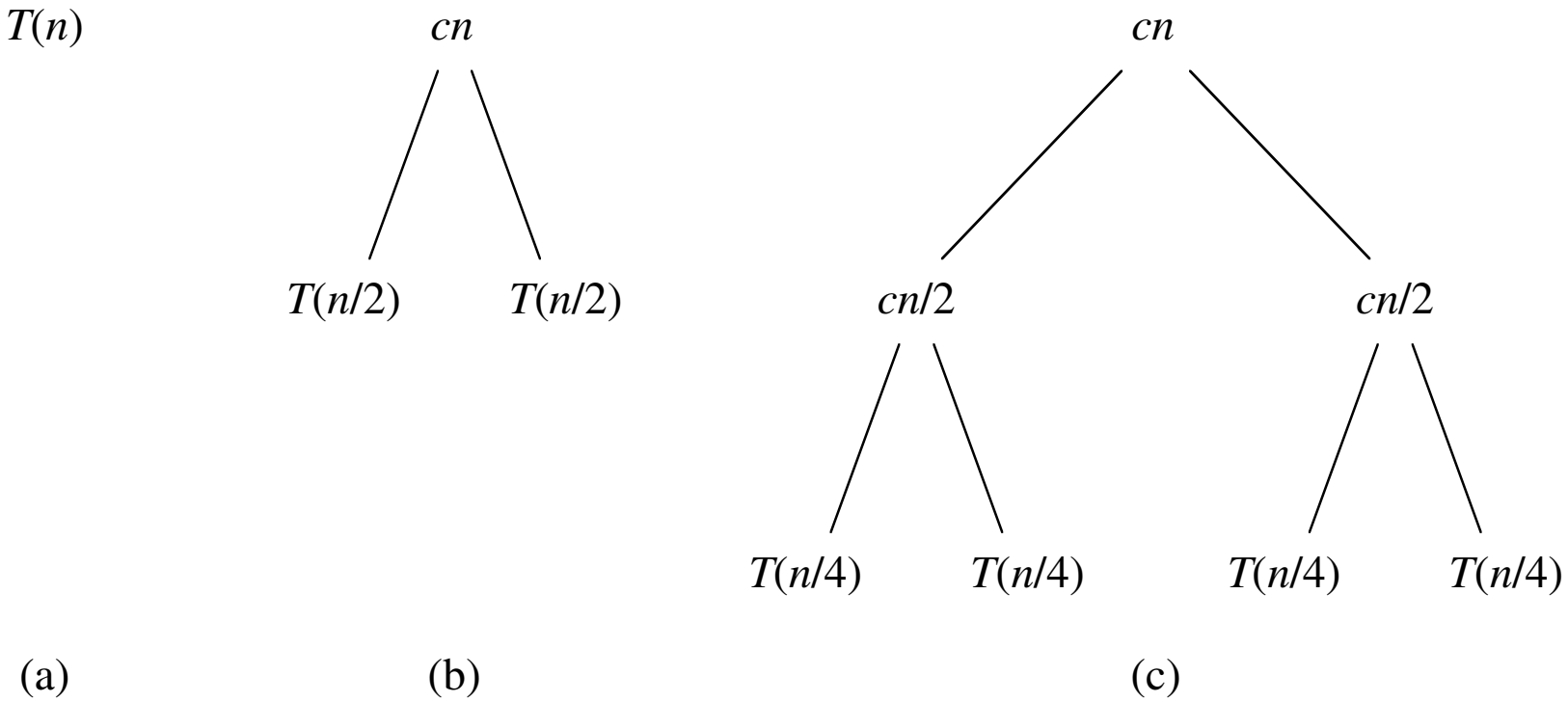
$$T(n) = \begin{cases} c & \text{si } n = 1 \\ 2T(n/2) + cn & \text{si } n > 1 \end{cases}$$

De manière générale pour les algorithmes de type DR on aura une complexité de la forme :

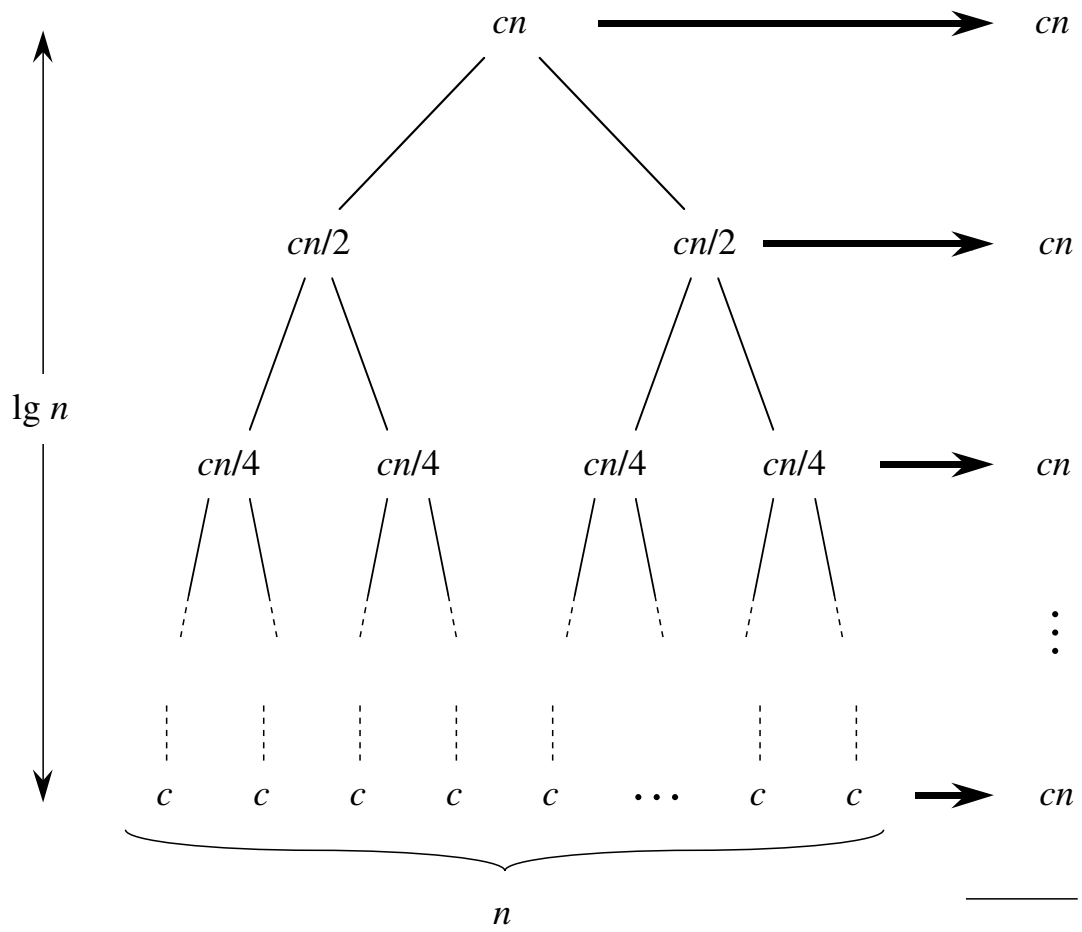
$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ aT(\lceil n/b \rceil) + f(n) & \text{si } n > 1 \end{cases}$$

où $a \geq 1, b > 1$ et $f(n)$ est une fonction donnée.

Complexité de tri fusion (1/2) on suppose $n = 2^k$ (c-à-d. $k = log_2(n)$)



Complexité de tri fusion (2/2) on suppose $n = 2^k$



(d)

Algorithme de FUSION

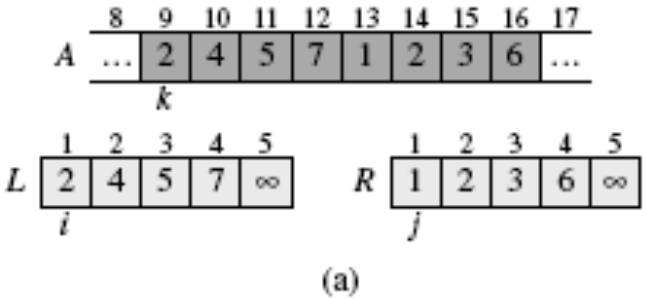
```
FUSION(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  créer tableaux L[1 .. n1 + 1] et R[1 .. n2 + 1]
 4  pour i ← 1 à n1
 5      faire L[i] ← A[p + i − 1]
 6  pour j ← 1 à n2
 7      faire R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  pour k ← p à r
13      faire si L[i] ⩽ R[j]
14                alors A[k] ← L[i]
15                     i ← i + 1
16                sinon A[k] ← R[j]
17                     j ← j + 1
```

Exercice

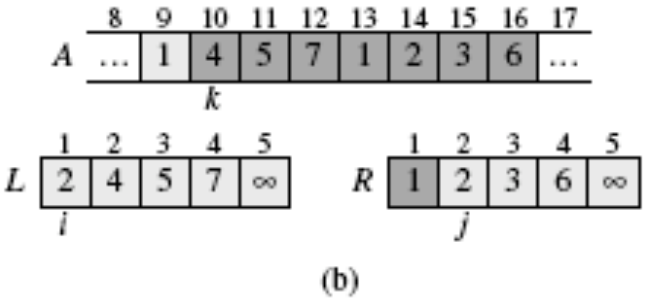
Prouver l'algorithme de FUSION ci-dessus.

Algorithme de FUSION

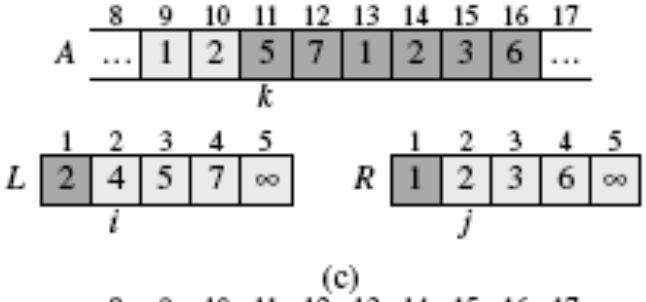
```
FUSION(A, p, q, r)
1  n1 ← q − p + 1
2  n2 ← r − q
3  créer tableaux L[1 . . n1 + 1] et R[1 . . n2 ·
4  pour i ← 1 à n1
5      faire L[i] ← A[p + i − 1]
6  pour j ← 1 à n2
7      faire R[j] ← A[q + j]
8  L[n1 + 1] ← ∞
9  R[n2 + 1] ← ∞
10 i ← 1
11 j ← 1
12 pour k ← p à r
13     faire si L[i] ⩽ R[j]
14             alors A[k] ← L[i]
15                  i ← i + 1
16             sinon A[k] ← R[j]
17                  j ← j + 1
```



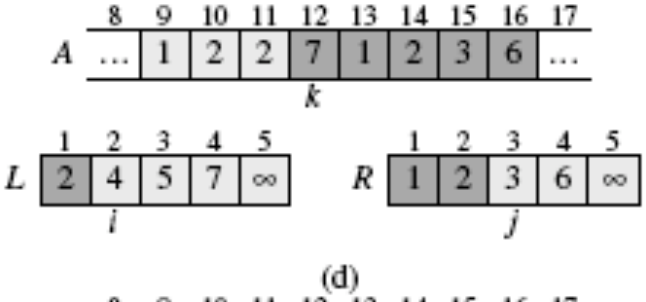
(a)



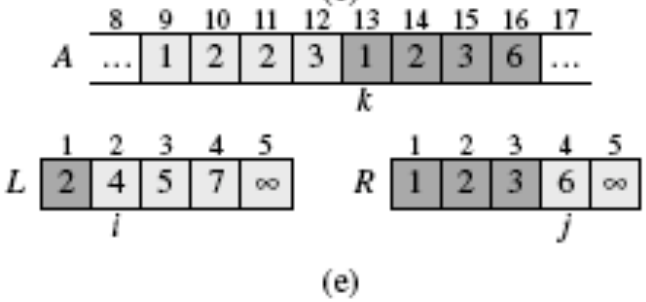
(b)



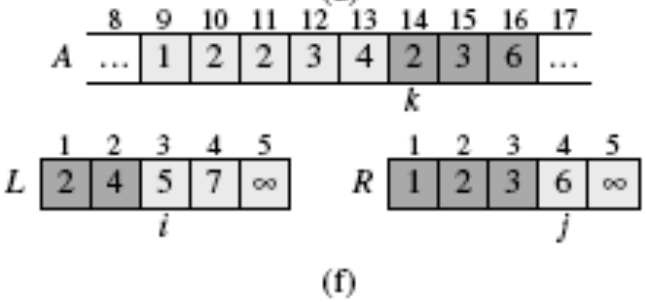
(c)



(d)



(e)



(f)

à continuer ...

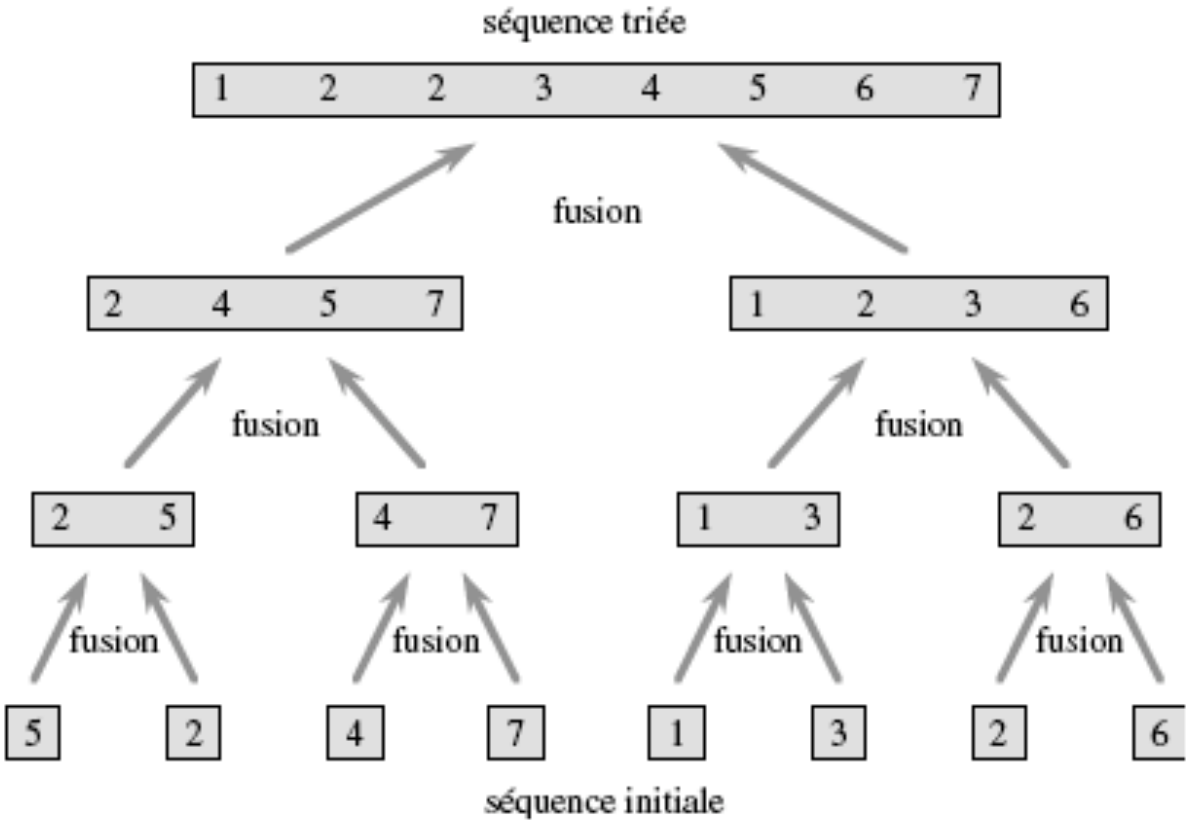


Figure 2.4 Le fonctionnement du tri par fusion sur le tableau $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. Les longueurs des séquences triées en cours de fusion augmentent à mesure que l'algorithme remonte du bas vers le haut.

Méthodes pour résoudre les récurrences

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ aT(\lceil n/b \rceil) + f(n) & \text{si } n > 1 \end{cases}$$

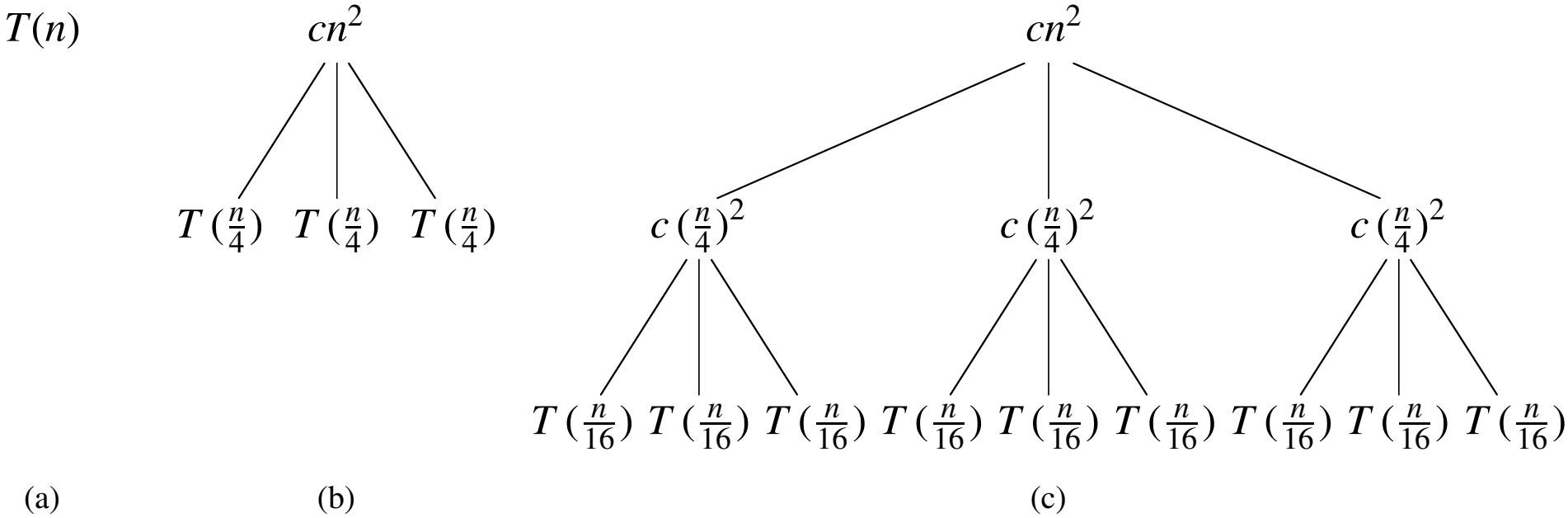
où $a \geq 1, b > 1$ et $f(n)$ est une fonction donnée.

Exercice

Voir le Chapitre 4 de "Introduction à l'algorithmique", T. H. Cormen, C. E. Leiserson, R. L. Rivest, Dunod, 1994.

- Dépliage de la récurrence.
- Méthode par substitution : on devine une solution et on vérifie qu'elle marche.
- Méthode par substitution partielle : on devine une solution mais sans préciser les constantes.

Cas de $T(n) = 3T(\lceil n/4 \rceil) + \Theta(n^2)$



Cas de $T(n) = 3T(\lceil n/4 \rceil) + \Theta(n^2)$

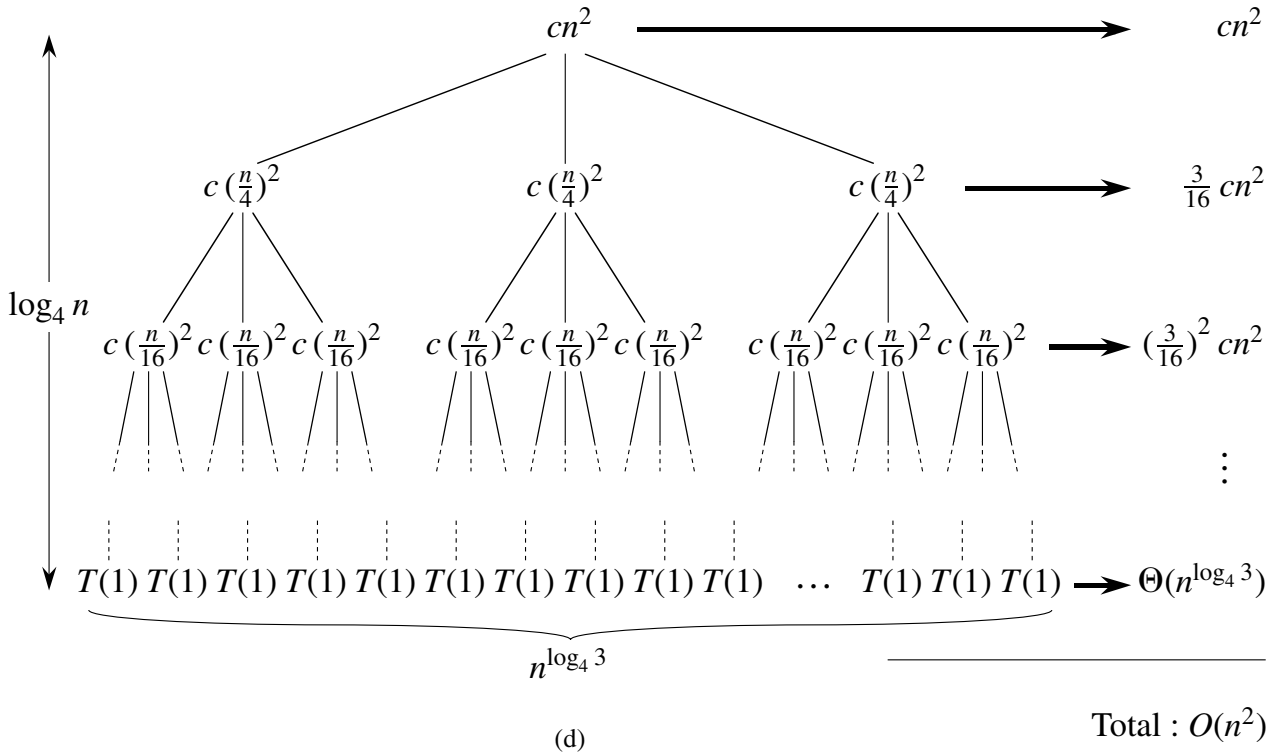


Figure 4.1 La construction d'un arbre récursif pour la récurrence $T(n) = 3T(n/4) + cn^2$. La partie **(a)** montre $T(n)$, progressivement développé dans les parties **(b)–(d)** pour former l'arbre récursif. L'arbre entièrement développé, en partie (d), a une hauteur de $\log_4 n$ (il comprend $\log_4 n + 1$ niveaux).

Le dernier niveau, situé à la profondeur $log_4 n$, a $3^{log_4 n} = n^{log_4 3}$ nœuds, qui ont chacun un coût $T(1)$ et qui donnent un coût total de $n^{log_4 3} T(1)$, qui est $O(n^{log_4 3})$.

Cas de $T(n) = T(\lceil n/3 \rceil) + T(\lceil 2n/3 \rceil) + cn$

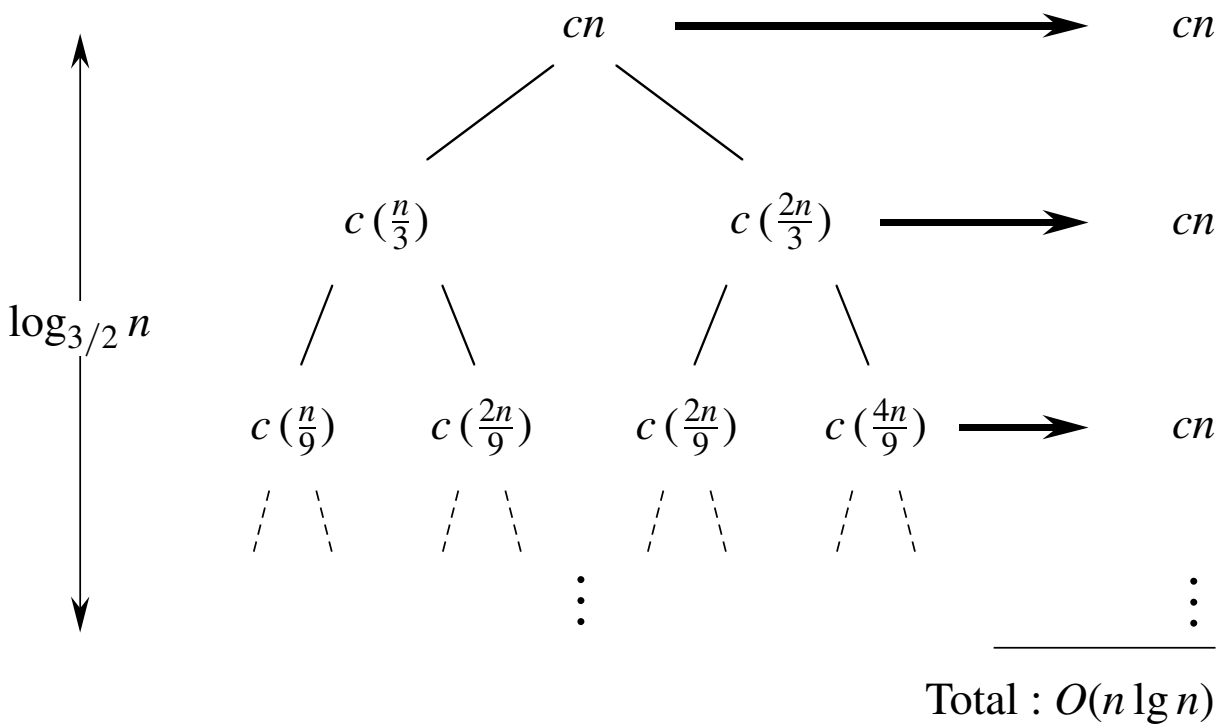
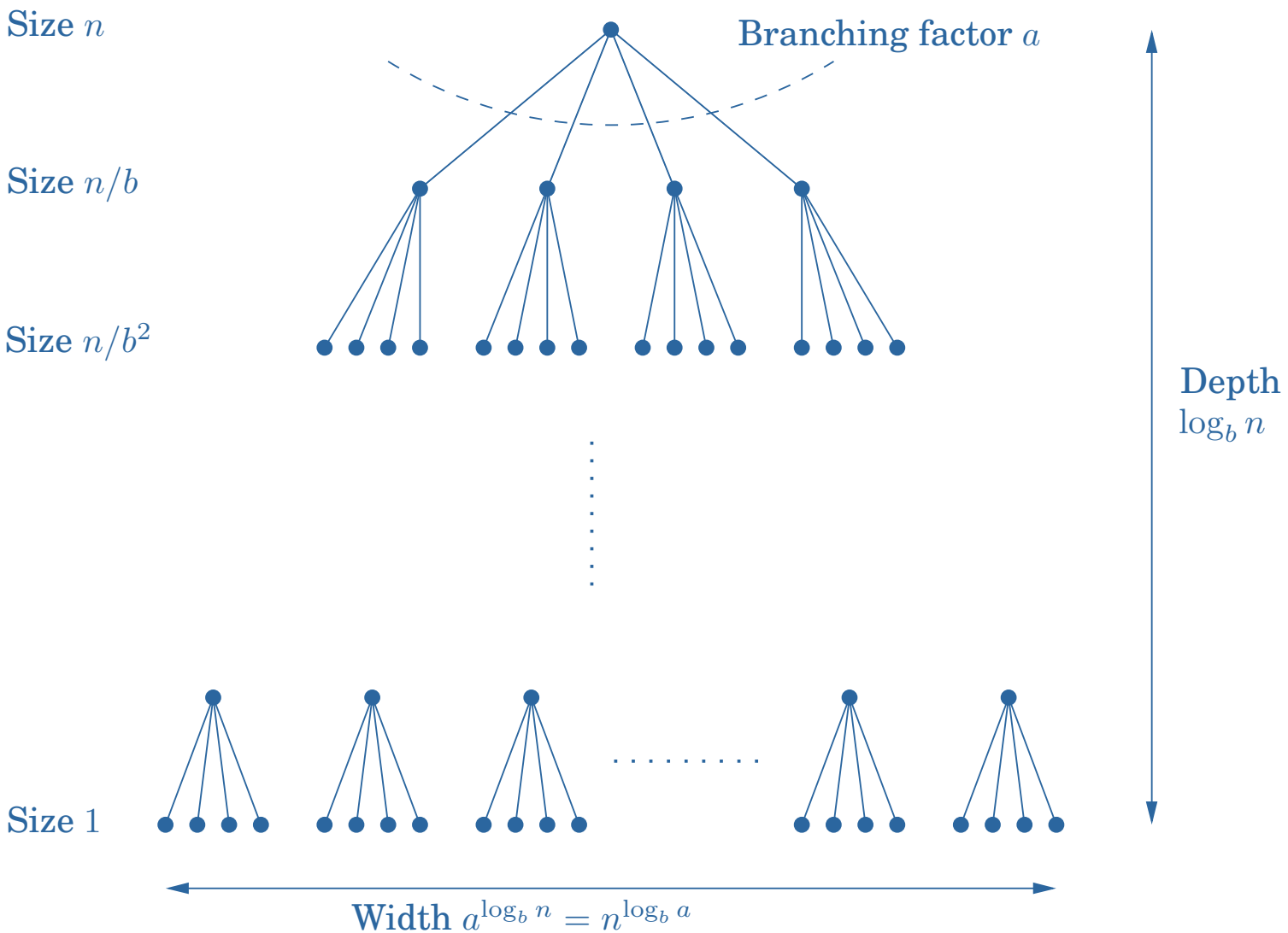


Figure 4.2 Un arbre récursif pour la récurrence $T(n) = T(n/3) + T(2n/3) + cn$.

Cas général $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ avec $a > 0, b > 1, d \geq 0$



Théorème fondamental pour la complexité

Théorème

Si $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ *avec* $a > 0, b > 1, d \geq 0$, *alors*

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

Exercice

Voir la preuve de ce théorème en Section 4.4 de [**?**].

Exercice

Appliquer ce théorème (bien pratique !) aux exemples vus justes avant, et retrouver less résultats établis à la main.

Nous allons voir de nombreux exemples d'utilisation de DR

1. Le tri rapide (Quicksort)
2. Calcul du médian ou problème de sélection
3. Multiplications de nombres à n digits
4. Les deux points les plus rapprochés
5. Multiplications de matrices $n \times n$
6. Le problème des gratte-ciels

mais il y a aussi de nombreuses applications en traitement du signal (l'algorithme de la FFT Fast Fourrier Transformation), d'autres cas en géométrie (enveloppe convexe) menant à de grands progrès en conception et dessins assistés par ordinateur.

Le tri rapide/Quicksort
découvert en 1962 par Tony Hoare (11 Janvier 1934-)

Problème (Le tri)

Entrée : un tableau T d'entiers

Sortie : une permutation de T triée

```
TRI-RAPIDE(A, p, r)
1  si p < r
2     alors q ← PARTITION(A, p, r)
3           TRI-RAPIDE(A, p, q − 1)
4           TRI-RAPIDE(A, q + 1, r)
```

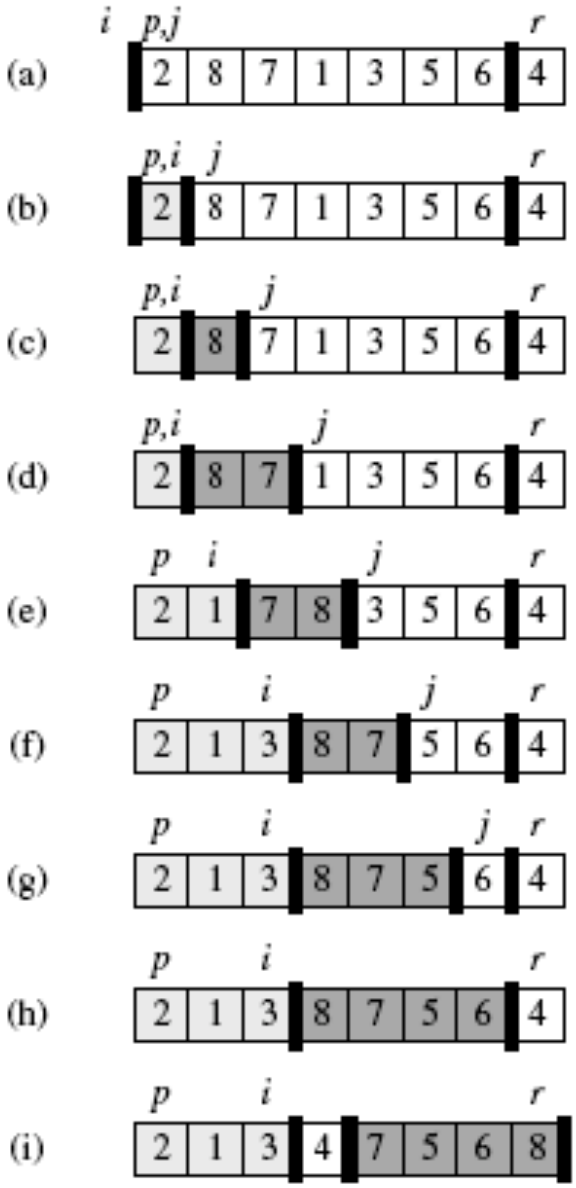
Pour trier un tableau A entier, l'appel initial est TRI-RAPIDE(A, 1, *longueur*[A]).

avec après l'appel de partition :

$$\forall k, p \leq k < q, A[k] \leq A[q] \text{ et } \forall k, q < k \leq r, A[q] < A[k]$$

Algorithme de PARTITION

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  pour j ← p à r − 1
4      faire si A[j] ⩽ x
5          alors i ← i + 1
6                permuter A[i] ↔ A[j]
7  permuter A[i + 1] ↔ A[r]
8  retourner i + 1
```



Fonctionnement de PARTITION sur un exemple

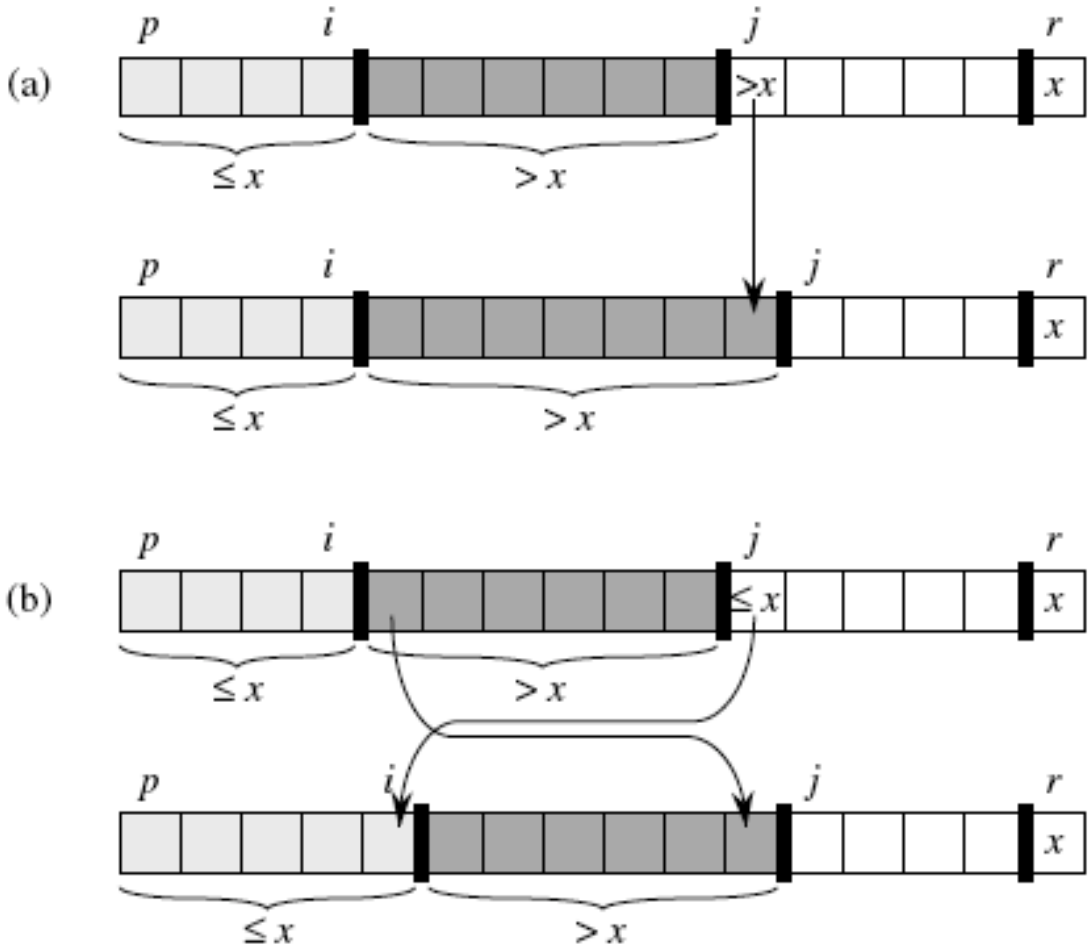


Figure 7.3 Les deux cas pour une itération de la procédure PARTITION. **(a)** Si $A[j] > x$, l'unique action est d'incrémenter j, ce qui conserve l'invariant de boucle. **(b)** Si $A[j] \leqslant x$, l'indice i est incrémenté, $A[i]$ et $A[j]$ sont échangés, puis j est incrémenté. Ici aussi, l'invariant de boucle est conservé.

Complexité du tri rapide

- Le cas le plus défavorable intervient pour le tri rapide quand la routine de partitionnement produit un sous-problème à $n-1$ éléments et un autre avec 0 élément. C'est le cas si la séquence est déjà triée (ou en sens inverse) alors l'algorithme s'éxecute en $\Omega(n^2)$ car la récurrence devient

$$T(n) = T(n-1) + \Theta(1)$$

d'où $T(n) = \Theta(n^2)$.

- Le cas le plus favorable s'obtient lorsque le pivot vient toujours se placer au milieu.

$$T(n) = 2T(n/2) + \Theta(n) \rightsquigarrow T(n) = O(nlogn)$$

Remarque

On peut montrer que c'est ce qu'on obtient en moyenne avec une version "randomizée" (dans le choix du pivot) du tri rapide (Chapitre 7 "du Cormen").

Complexité du tri rapide

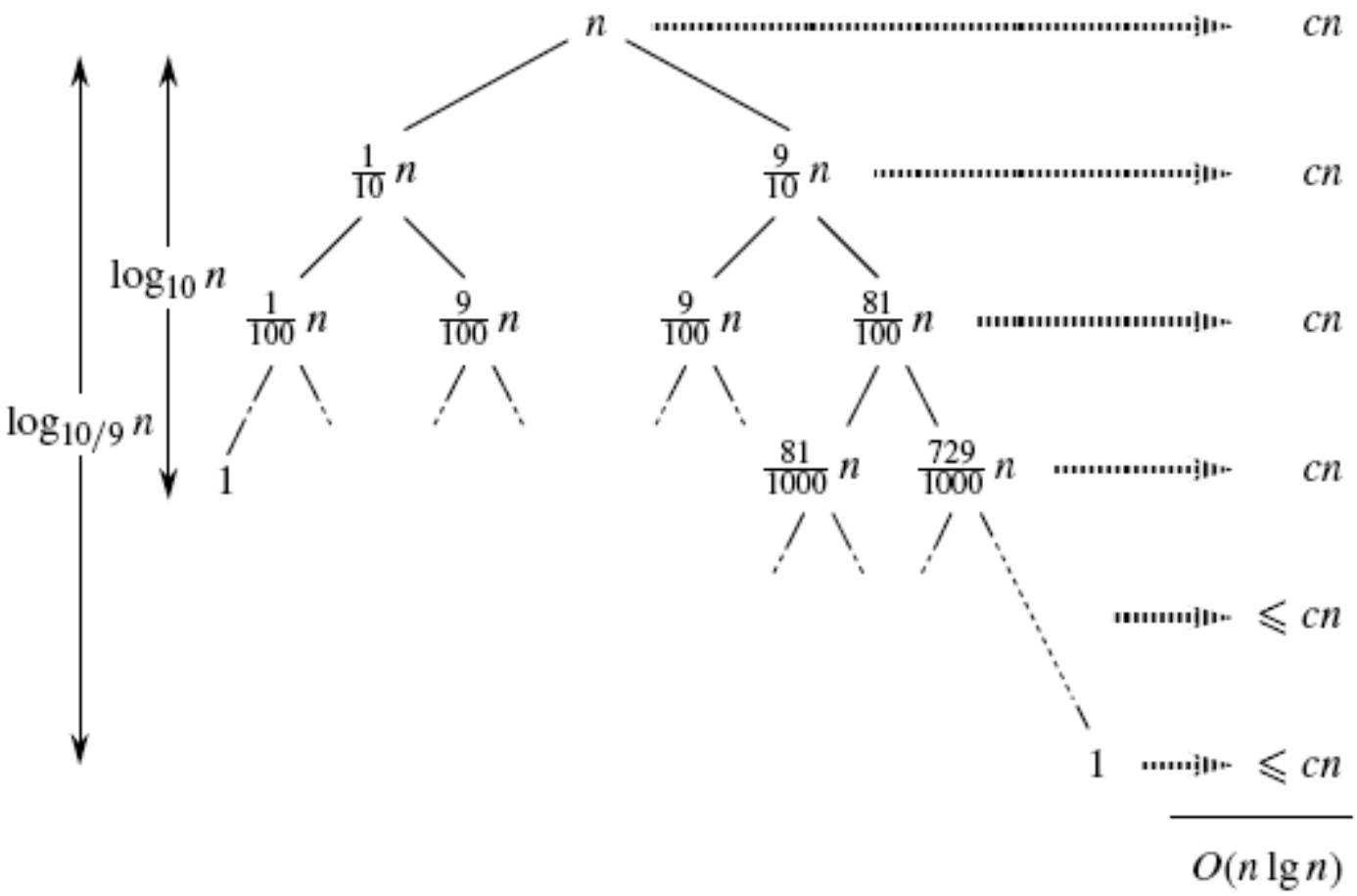
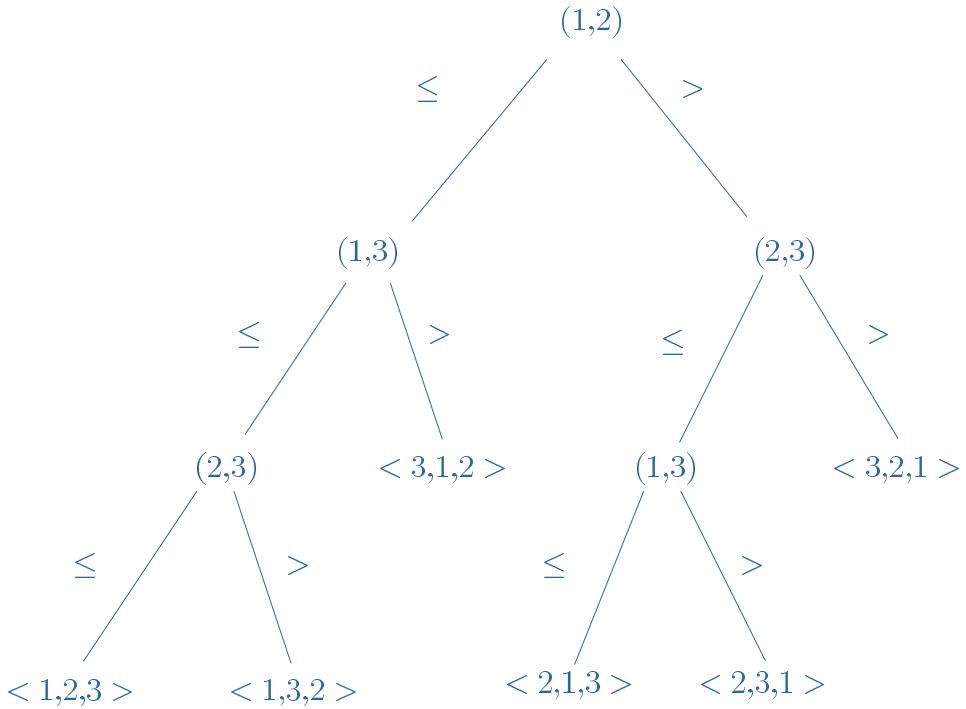


Figure 7.4 Arbre récursif de TRI-RAPIDE dans lequel PARTITION produit toujours une décomposition 9-1, donnant ainsi un temps d'exécution $O(n \lg n)$. Les nœuds montrent les tailles de sous problème, les coûts par niveau figurant à droite. Les coûts par niveau incluent la constante c implicitement contenue dans le terme $\Theta(n)$

Complexité intrinsèque du tri par comparaison

On considère les arbres de décision.

Exemple L'arbre de décision du tri par sélection ordinaire pour une liste de 3 éléments est représenté ci-après.



Théorème

Tout arbre de décision d'un tri pour n éléments a une hauteur d'ordre supérieur $O(nlog(n))$.

Le problème de Sélection

Problème (Le problème de Sélection)

Entrée : une liste d'entiers S et un entier i

Sortie : l'élément de rang i

Définitions

Définition

Le i-ème rang d'un ensemble à n éléments est le i-ème plus petit élément.

Exemple

Le minimum d'un ensemble (à n éléments) est l'élément de rang 1, et le maximum est l'élément de rang n.

Définition

Un médian d'un ensemble à n éléments est un élément "au milieu" :
Si n est impair, il est unique et c'est celui de rang $(n+1)/2$.
Si n est pair, il existe deux médians de rangs respectifs

- $n/2$ (médian inférieur)
- $n/2+1$ (médian supérieur)

On convient de prendre toujours $\lfloor (n+1)/2 \rfloor$.

Calcul de l'élément de rang i

- Solution 1 : Il suffit de trier, donc $O(nlogn)$, mais on fait plus que demandé.

  Peut-on faire en $O(n)$?

Calcul de l'élément de rang i

- Solution 1 : Il suffit de trier, donc $O(nlogn)$, mais on fait plus que demandé.

  Peut-on faire en $O(n)$? Oui !

- Solution 2 : Diviser-pour-Régner.

Un algorithme Diviser-pour-Régner pour le problème de sélection

Si on se fixe une valeur v donnée, imaginons que S soit séparée en 3 sous-listes d'éléments : ceux plus petits que v (S_L), ceux égaux à v (S_v) et ceux plus grands que v (S_R).

Exemple

Soit S : | 2 | 36 | 5 | 21 | 8 | 13 | 11 | 20 | 5 | 4 | 1 | et $v = 5$. Alors

S_L : | 2 | 4 | 1 | S_v : | 5 | 5 | S_R : | 36 | 21 | 8 | 13 | 11 | 20 |.

On voit que $Selection(S, 8) = Selection(S_R, 3)$

Plus généralement :

$$Selection(S, i) = \begin{cases} Selection(S_L, i) & \text{if } i \leq |S_L| \\ v & \text{if } |S_L| < i \leq |S_L| + |S_v| \\ Selection(S_R, i - |S_L| - |S_v|) & \text{if } i > |S_L| + |S_v| \end{cases}$$

On cherche $Selection(S, \lfloor (n+1)/2 \rfloor)$

Complexité du calcul de $Selection(S, \lfloor (n+1)/2 \rfloor)$

$$Selection(S,i) = \begin{cases} Selection(S_L, i) & \text{if } i \leq |S_L| \\ v & \text{if } |S_L| < i \leq |S_L| + |S_v| \\ Selection(S_R, i - |S_L| - |S_v|) & \text{if } i > |S_L| + |S_v| \end{cases}$$

- À v fixé le calcul de S_L, S_v et S_R se fait en $O(n)$ (et on peut même faire ce calcul *en place*)
- Imaginons que l'on choisisse "bien" la valeur v à chaque appel récursif de sorte que $|S_L|, |S_R| \approx \frac{1}{2}|S|$. Alors le calcul de $Selection(S, \lfloor (n+1)/2 \rfloor)$ a une complexité $T(n)$ telle que

$$T(n) = T(n/2) + O(n)$$

Théorème

Si $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ *avec* $a > 0, b > 1, d \geq 0$, *alors*

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

On a un algorithme en $O(n)$!

- Il suffit de choisir v aléatoirement (hors cours, voir [?] page 65)

Multiplications de nombres à n digits

Problème (Multiplications de nombres à n digits)

Entrée : deux entiers codés sur n digits

Sortie : leur produit

La multiplication à l'école :

$$
\begin{array}{r}
31415962 \\
\times\ 27182818 \\
\hline
251327696 \\
31415962 \\
251327696 \\
62831924 \\
251327696 \\
31415962 \\
219911734 \\
62831924 \\
\hline
853974377340916
\end{array}
$$

Un algorithme en $O(n^2)$.

On peut faire mieux avec DR !

Si on découpe les nombres

Or on remarque que

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

```
MULTIPLY(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      e ← MULTIPLY(a, c, m)
      f ← MULTIPLY(b, d, m)
      g ← MULTIPLY(b, c, m)
      h ← MULTIPLY(a, d, m)
      return 10^{2m} e + 10^m (g + h) + f
```

D'où un algorithme qui en découle :

$$T(n) = 4\,T(\lceil n/2 \rceil) + O(n) \text{ et } T(1) = 1$$

d'où toujours un algorithme en $O(n^2)$, ZUT !

Anatolii Karatsuba in 1962 (à la Gauss 1800) + Donald Knuth

Or on remarque que

$$bc + ad = ac + bd - (a-b)(c-d)$$

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^{2m} e + 10^m (e + f − g) + f
```

D'où l'algorithme :

$$T(n) = 3T(\lceil n/2 \rceil) + O(n) \text{ et } T(1) = 1$$

d'où un algorithme en $O(n^{log_2 3})$, or $log_2 3 \approx 1.585 < 2$.

Le problème des deux points les plus rapprochés

Problème (Les deux points les plus rapprochés)

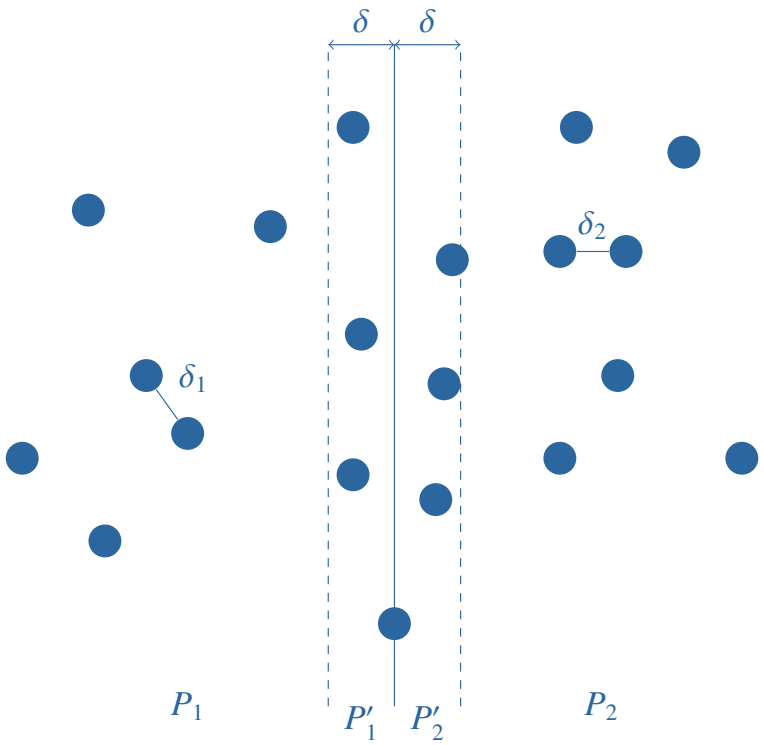
Entrée : un tableau P de pointsdans le plan

Sortie : $min_{a,b \in T, a \neq b} d(a,b)$

où $d(a,b) = \sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$, noté classiquement $\|a - b\|_2$

Diviser pour Régner : on coupe le plan en deux et on calcule δ_1 et δ_2 pour les deux sous-ensembles de points.

$\delta = min\{\delta_1, \delta_2\}$

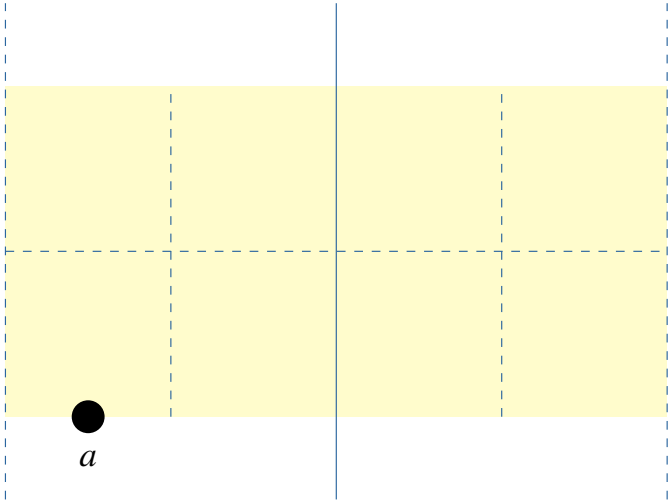


On cherche la distance minimale entre deux points distincts de la bande verticale $P' = P'_1 \cup P'_2$.

Calcul efficace dans la bande P'

On trie les points par ordonnées croissantes et on montre que pour un point a, il suffit de calculer les distances entre a et b où b est parmi les 7 points qui succèdent à a dans P'.

Si un point est distant de moins de δ de a alors il est dans le rectangle jaune de taille $2\delta \times \delta$.



Lemme

Il ne peut y avoir plus de 8 points distants d'au moins δ dans ce rectangle de dimension $2\delta \times \delta$.

Donc il suffira de considérer a et les 7 autres points suivants pour l'ordre d'ordonnées croissantes.

Algorithme *plusRapproches*(P)

Entrées : Un tableau de points P du plan
Sorties : La distance minimale, i.e. $min_{a,b \in T | a \neq b} ||a - b||_2$

```
1 X := tri(P, par abscisses croissantes) ;
2 Y := tri(P, par ordonnées croissantes) ;
3 plusRapprochesRec(X, Y)
```

Algorithme *plusRapprochesRec*(X, Y)

Entrées : Deux tableaux X, Y qui contiennent les même points, X est trié par absisse croissante et Y est trié par ordonnée croissante

Sorties : La distance minimale, i.e. $min_{a,b \in X | a \neq b} \|a - b\|_2$

1 **si** card$(P) \leq 3$ **alors**
2 **retourner** *méthode naïve*
3 **sinon**
4 $X_G := X[1, \lfloor \frac{|X|}{2} \rfloor]$
5 $X_D := X[\lfloor \frac{|X|}{2} \rfloor + 1, |X|]$
6 $x_{sep} := X_G[\frac{|X|}{2}].x$
7 $Y_G :=$ extraire les éléments de Y d'abcsisse $\leq x_{sep}$
8 $Y_D :=$ extraire les éléments de Y d'abcsisse $> x_{sep}$
9 $\delta_G := plusRapprochesRec(X_G, Y_G)$
10 $\delta_D := plusRapprochesRec(X_D, Y_D)$
11 $\delta := min(\delta_G, \delta_D)$
12 $Y' :=$ extraire les éléments de Y qui sont dans la bande verticale d'absisse x_{sep} et de largeur 2δ ;
13 **retourner** $min(\delta, plusRapprochesBande(Y', \delta))$

Les fonctions non données ici sont laissés en exercice.

Complexité de l'algorithme *plusRapproches*(P)

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$
$$T(3) = T(2) = T(1) = \Theta(1)$$

donc en $O(nlogn)$

Multiplications de matrices $n \times n$

Problème (Multiplications de matrices $n \times n$)

Entrée : deux matrices carrées X et Y de dimension n

Sortie : $X * Y$

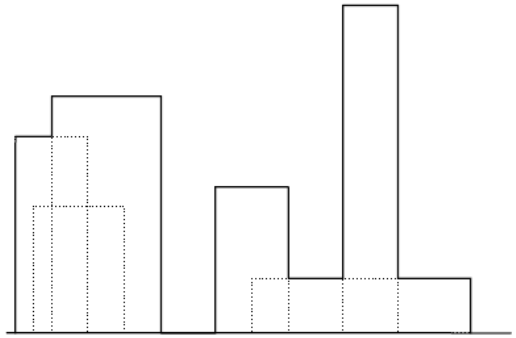
Au tableau

Le problème des gratte-ciels

Problème (Le problème des gratte-ciels)

Entrée : La description de n gratte-ciels dans un tableau I, chaque immeuble est représenté par un intervalle de points $[a, b]$ désignant sa ligne d'horizon (donc $a.y = b.y$).

Sortie : la ligne qui décrit les toits des gratte-ciels pour un observateur qui regarde le paysage en face urbain ; certaines parties d'immeubles sont cachées.



Exercice

Il vous est fortement recommandé d'essayer de le résoudre.