Introduction aux Algorithmes et Programmation en C++

Vincent Chabannes
Bureau 312, UFR math info — vincent.chabannes@math.unistra.fr

Outline

- **1.** Les algorithmes de tri
  - 1.1 Introduction
  - 1.2 Conception, analyse et implémentation en C++
  - 1.3 Applications

1. Les algorithmes de tri

- 1.1 Introduction
- 1.2 Conception, analyse et implémentation en C++
- 1.3 Applications

- **1.1 Introduction**
- 1.2 Conception, analyse et implémentation en C++
- 1.3 Applications

Le classique problème du tri

Il semble que l'être humain préfère le rangement au dérangement, l'ordre au désordre. Le tri est une activité qui est très pratiquée dans notre vie quotidienne.

- Ainsi, est-il vraisemblable que vous préférez habiter un espace où les choses sont rangées à leur place. Il vous est dans ce cas plus facile de les trouver quand vous en aurez besoin.
- Lorsque nous jouons aux cartes, la plupart des personnes ordonnent par couleur et par valeur les cartes, en fonction du jeu auquel ils jouent.
- Mais cette préférence pour l'ordre, le rangement, est loin de concerner la seule vie quotidienne dans sa dimension pratique. Nous la retrouvons dans bien d'autres domaines.



Le problème du tri : avant l'ordinateur

Classer des données est une activité qui a occupé dans le passé de nombreux employés de bureau. Ils réalisaient cette tâche manuellement jusqu'à la naissance de la mécanographie.

Invention de la mécanographie

Hermann Hollerith (1860-1929), ingénieur américain en mécanique, invente (à lui seul) une machine qui trie et additionne des données à partir de cartes perforées pour produire des statistiques. Cette invention va permettre de dépouiller beaucoup plus rapidement le recensement des États-Unis de 1890. Les premiers résultats sont alors produits en quelques mois seulement et l'analyse complète en un peu moins de 3 ans (auparavant, il fallait au moins 8 ans).

Sur chaque carte, divisée en lignes et en colonnes, les perforations représentent, selon un code simple, les caractéristiques d'un individu : âge, sexe, métier, situation familiale, etc. Les cartes sont lues par un lecteur électrique, un trou laisse passer le courant et l'absence de trou l'arrête. En fonction du résultat, une trappe va s'ouvrir.



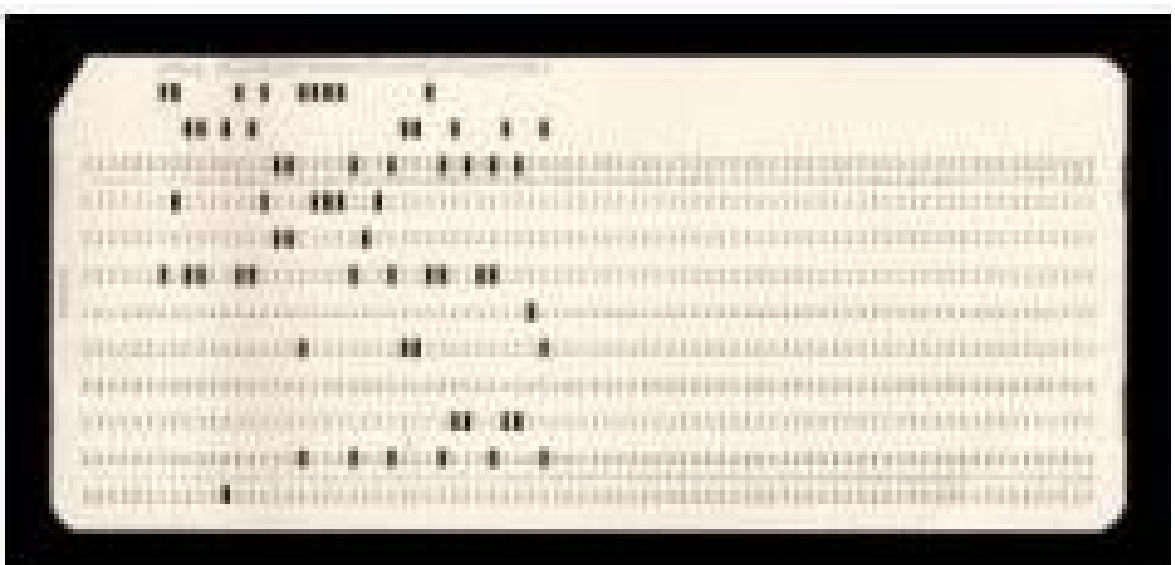
Le problème du tri : avant l'ordinateur

- Les cartes perforées comportent 80 colonnes comportant les chiffres de 0 à 9, ce qui permet de trier un code alphanumérique.
- Les machines se perfectionnent pour pouvoir trier 1000 cartes par minute en 1950 (Tri par base).

IBM 82 Card Sorter (1949)

Chaque fichier de cartes doit être trié, colonne par colonne. Un système de brosse détecte l'emplacement de la perforation et envoie la carte dans le bon casier. Une fois cette colonne triée, un opérateur doit reprendre toutes les piles de cartes dans un ordre déterminé, et les refaire passer dans la trieuse pour traiter la 2e colonne, et ainsi de suite.

À partir des années 1960, ces outils mécaniques sont progressivement remplacés par l'usage de l'ordinateur. Les cartes perforées ont d'ailleurs été les seuls systèmes d'entrée-sortie des premiers ordinateurs, avant l'apparition des claviers et écrans dans les années 1970.





Le problème du tri : récurrent en informatique

Problème du tri

Consiste à organiser une collection d'éléments selon une relation d'ordre déterminée. Les objets à trier doivent donc être comparables.

- Applications innombrables :
- Tri des mails selon leur ancienneté
- Tri des résultats de requête dans un moteur de recherche
- Tri des facettes des objets pour l'affichage dans les jeux 3D
- Gestion des opérations bancaires
- Le tri sert aussi de brique de base / pré-traitement pour d'autres algos
- Recherche par dichotomie
- Recherche des éléments dupliqués dans un tableau
- Recherche du $k^{\text{ème}}$ élément le plus grand dans une liste
- Calcul de la médiane d'une série statistique



Définition 1

Un algorithme de tri est dit en place s'il modifie directement la structure qu'il est en train de trier et qu'il ne construit aucun objet proportionnel à la taille de la séquence.

$\Longrightarrow$ Nécessite que l'objet trié soit modifiable $\Longrightarrow$ Intérêt principal : complexité en mémoire (ou spatiale) réduite !

Définition 2

Un algorithme de tri d'une séquence est dit stable s'il conserve la position relative dans la liste des quantités qui sont égales pour la relation d'ordre.

$\Longrightarrow$ Exemple : on veut trier la liste de cartes $T = [6\clubsuit, 5\spadesuit, 4\heartsuit, 5\diamondsuit]$

Si le tri laisse le 5♠ avant le 5♦, il s'agit d'un tri stable, sinon il est instable :

- si une fois trié, on obtient: $T = [4\heartsuit, 5\spadesuit, 5\diamondsuit, 6\clubsuit]$, le tri est stable,
- si une fois trié, on obtient: $T = [4\heartsuit, 5\diamondsuit, 5\spadesuit, 6\clubsuit]$, le tri est instable.

Les algorithmes de tri

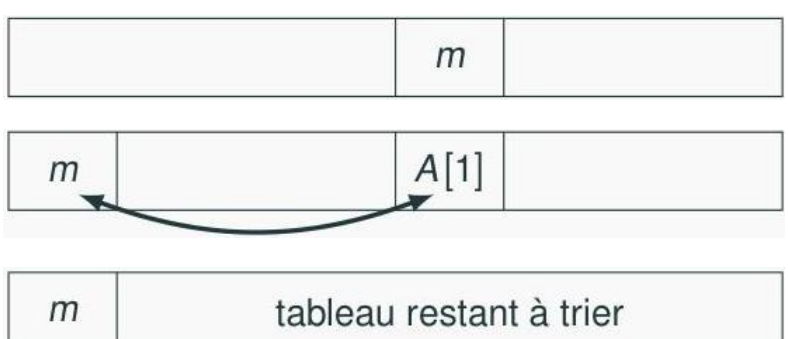
- Tris par comparaison
- Tri par sélection (selection sort)
- Tri par insertion (insertion sort)
- Tri à bulle (bubble sort)
- Tri par fusion (merge sort)
- Tri rapide (quick sort)
- Tri par tas (heap sort)
- Tris utilisant la structure des données
- Tri comptage ou Tri par dénombrement (counting sort)
- Tri par base (radix sort)
- Tri par paquets (bucket sort)

- 1.1 Introduction
- **1.2 Conception, analyse et implémentation en C++**
- 1.3 Applications

Le tri par sélection : le principe

Soit A un tableau. Le tri par sélection consiste simplement à sélectionner l'élément le plus petit du tableau à trier, à l'enlever, et à répéter itérativement le processus tant qu'il reste des éléments dans la séquence.

- recherche (et sélection) d'un élément minimal :
- permutation avec le premier :
- tri du tableau restant :



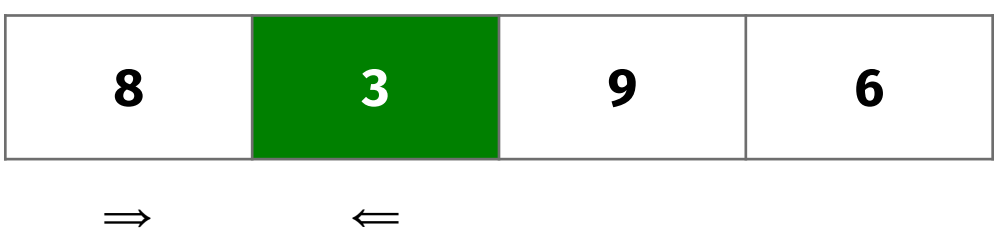
Le tri par sélection : le principe

text(size: 20pt, weight: “bold”) [8]	text(size: 20pt, weight: “bold”) [3]	text(size: 20pt, weight: “bold”) [9]	text(size: 20pt, weight: “bold”) [6]

Le tri par sélection : le principe

Itération $k = 1$

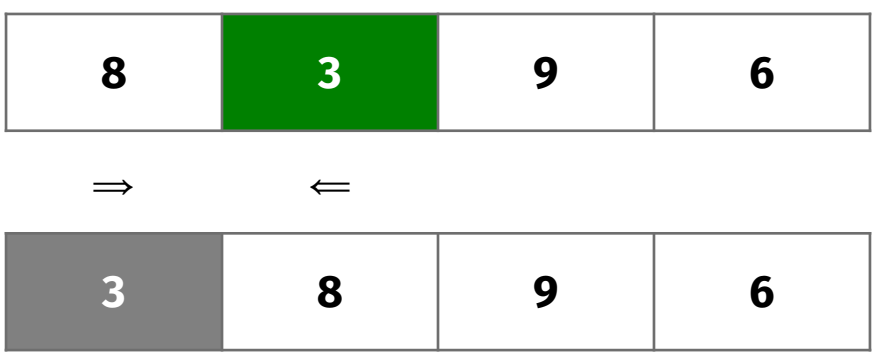
On recherche le minimum dans le tableau complet et on l'échange avec l'élément se trouvant en position 1.



Le tri par sélection : le principe

Itération $k = 1$

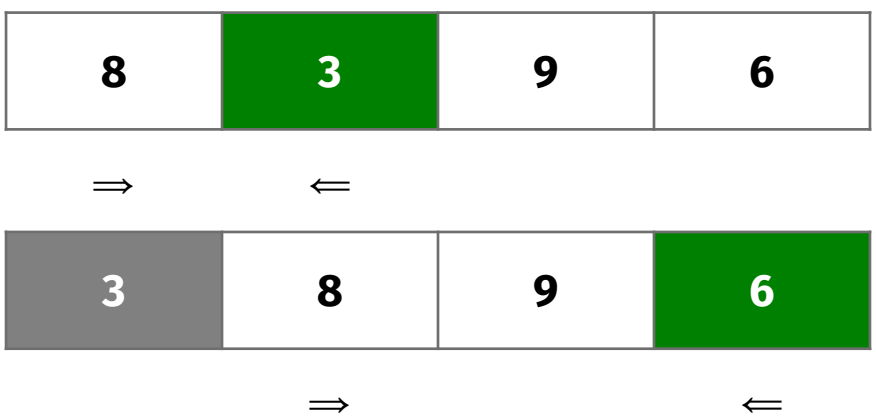
On recherche le minimum dans le tableau complet et on l'échange avec l'élément se trouvant en position 1.



Le tri par sélection : le principe

Itération $k = 2$

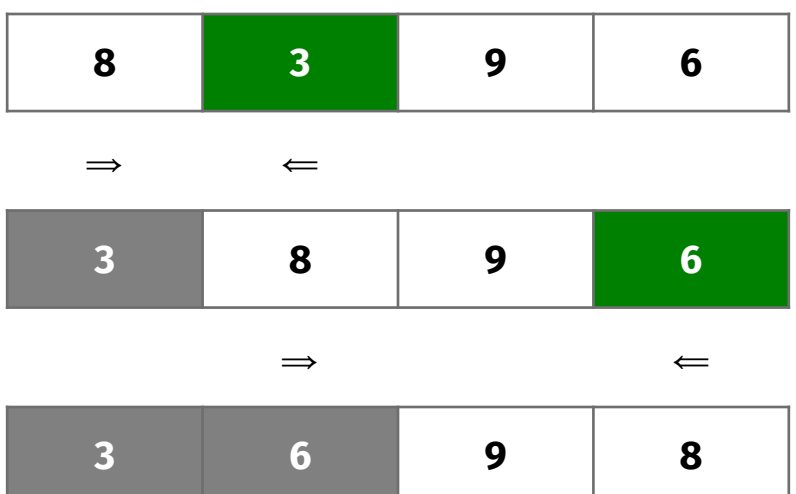
On recherche le minimum dans le tableau tel que l'indice est > 1 et on l'échange avec l'élément se trouvant en position 2.



Le tri par sélection : le principe

Itération $k = 2$

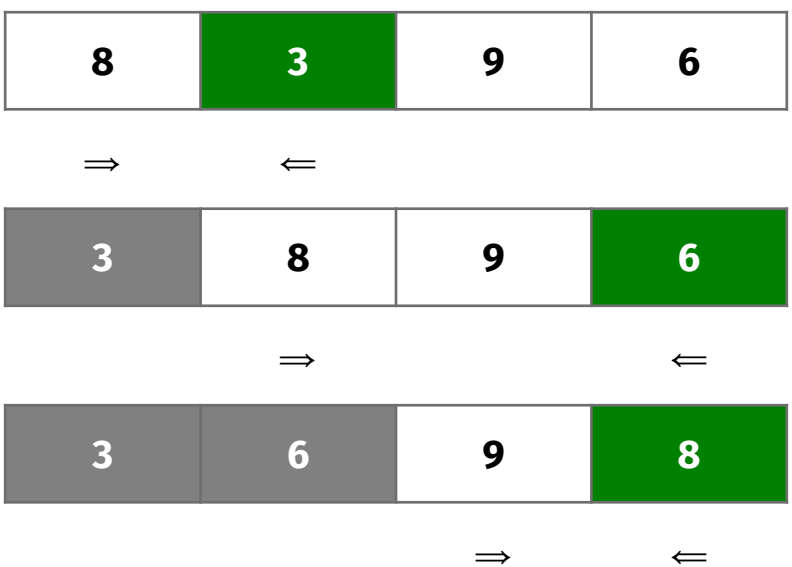
On recherche le minimum dans le tableau tel que l'indice est > 1 et on l'échange avec l'élément se trouvant en position 2.



Le tri par sélection : le principe

Itération $k = 3$

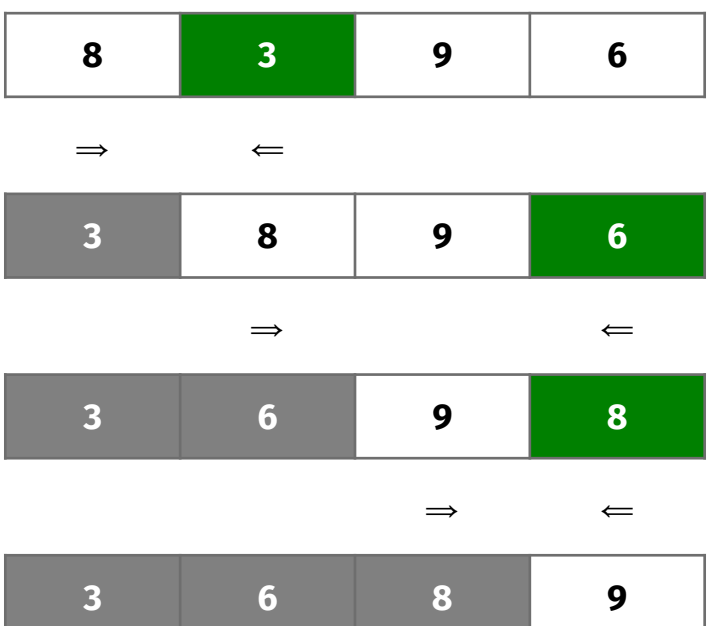
On recherche le minimum dans le tableau tel que l'indice est > 2 et on l'échange avec l'élément se trouvant en position 3.



Le tri par sélection : le principe

Itération $k = 3$

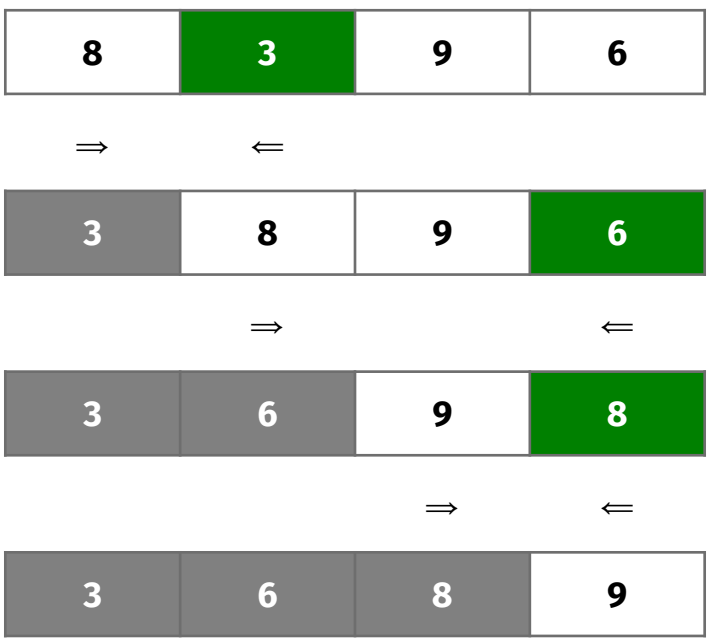
On recherche le minimum dans le tableau tel que l'indice est > 2 et on l'échange avec l'élément se trouvant en position 3.



Le tri par sélection : le principe

Finalisation

Il ne reste plus qu'un seul élément à trier, il n'y a donc rien à faire, le tableau est déjà trié.



Le tri par sélection : pseudo-code


```
Require: A un tableau à trier de taille
n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(A,n)
2: for k = 1 to n - 1 do
3: q = INDICEMIN(A, n, k)
4: if q ≠ k then
5: ECHANGE(A[q],A[k])
6: end if
7: end for
8: end procedure
```

```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: q := j
3: for i = j + 1 to n do
4: if A[i] < A[q] then
5: q := i
6: end if
7: end for
8: return q
9: end procedure
```

```
Require: a et b deux valeurs
1: procedure ÉCHANGE(a,b)
2: tmp = a
3: a = b
4: b = tmp
5: end procedure
```

Le tri par sélection : correction de INDICEMIN

```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A, n, j)
2: q := j
3: for i = j + 1 to n do
4: if A[i] < A[q] then
5: q := i
6: end if
7: end for
8: return q
9: end procedure
```

Terminaison

- La terminaison est évidente, il y a juste une boucle for dans cette procédure.
- Le variant de boucle est donc $n - i$ qui est bien une suite strictement décroissante et positive.

Le tri par sélection : correction de INDICEMIN


```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: q := j
3: for i = j + 1 to n do
4: if A[i] < A[q] then
5: q := i
6: end if
7: end for
8: Q(q)
9: return q
10: end procedure
```

Correction partielle

- Postcondition :

$$Q(j,q) \iff \forall k \mid j \leq k \leq n, A[q] \leq A[k]$$

Le tri par sélection : correction de INDICEMIN


```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: q := j
3: i = j + 1
4: while i ≤ n do
5: if A[i] < A[q] then
6: q := i
7: end if
8: i = i + 1
9: end while
10: Q(q)
11: return q
12: end procedure
```

Correction partielle

- Postcondition :

$$Q(j,q) \iff \forall k \mid j \leq k \leq n, A[q] \leq A[k]$$

Le tri par sélection : correction de INDICEMIN


```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: q := j
3: i = j + 1
4: I(i,q)
5: while i ≤ n do
6: I(i,q) ∧ i ≤ n
7: if A[i] < A[q] then
8: q := i
9: end if
10: i = i + 1
11: I(i,q)
12: end while
13: I(i,q) ∧ i > n
14: Q(q)
15: return q
16: end procedure
```

Correction partielle

- Postcondition :

$$Q(j,q) \iff \forall k \mid j \leq k \leq n, A[q] \leq A[k]$$

- Invariant :

$$I(i,q) \iff \forall k \mid j \leq k \leq i-1, A[q] \leq A[k]$$

Le tri par sélection : correction de INDICEMIN


```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: q := j
3: i = j + 1
4: I(i, q)
5: while i ≤ n do
6: I(i, q) ∧ i ≤ n
7: if A[i] < A[q] then
8: q := i
9: end if
10: I(i + 1, q)
11: i = i + 1
12: I(i, q)
13: end while
14: I(i, q) ∧ i > n
15: Q(q)
16: return q
17: end procedure
```

Correction partielle

Postcondition : $Q(j,q) \iff \forall k \mid j \leq k \leq n, A[q] \leq A[k]$ - Invariant : $I(i,q) \iff \forall k \mid j \leq k \leq i-1, A[q] \leq A[k]$ - $I(i+1,q) \iff \forall k \mid j \leq k \leq i, A[q] \leq A[k]$

Le tri par sélection : correction de INDICEMIN


```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: q := j
3: i = j + 1
4: I(i, q)
5: while i ≤ n do
6: I(i, q) ∧ i ≤ n
7: if A[i] < A[q] then
8: I(i + 1, i) ∧ A[i] < A[q]
9: q := i
10: end if
11: I(i + 1, q)
12: i = i + 1
13: I(i, q)
14: end while
15: I(i, q) ∧ i > n
16: Q(q)
17: return q
18: end procedure
```

Correction partielle

- Postcondition : $Q(j,q) \iff \forall k \mid j \leq k \leq n, A[q] \leq A[k]$
- Invariant : $I(i,q) \iff \forall k \mid j \leq k \leq i-1, A[q] \leq A[k]$
- $I(i+1,q) \iff \forall k \mid j \leq k \leq i, A[q] \leq A[k]$
- $I(i+1,i) \iff \forall k \mid j \leq k \leq i, A[i] \leq A[k]$

Le tri par sélection : correction de INDICEMIN


```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: q := j
3: i = j + 1
4: I(i, q)
5: while i ≤ n do
6: I(i, q) ∧ i ≤ n
7: if A[i] < A[q] then
8: I(i + 1, i) ∧ A[i] < A[q]
9: q := i
10: end if
11: I(i + 1, q)
12: i = i + 1
13: I(i, q)
14: end while
15: I(i, q) ∧ i > n
16: Q(q)
17: return q
18: end procedure
```

Correction partielle

- Postcondition : $Q(j,q) \iff \forall k \mid j \leq k \leq n, A[q] \leq A[k]$
- Invariant : $I(i,q) \iff \forall k \mid j \leq k \leq i-1, A[q] \leq A[k]$
- $I(i+1,q) \iff \forall k \mid j \leq k \leq i, A[q] \leq A[k]$
- $I(i+1,i) \iff \forall k \mid j \leq k \leq i, A[i] \leq A[k]$
- $I(i,q) \wedge i \leq n \implies I(i+1,i) \wedge A[i] < A[q] \vee I(i+1,q) \wedge A[i] \geq A[q]$

Le tri par sélection : correction de INDICEMIN


```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: I(j + 1, j)
3: q := j
4: I(j + 1, q)
5: i = j + 1
6: I(i, q)
7: while i ≤ n do
8: I(i, q) ∧ i ≤ n
9: if A[i] < A[q] then
10: I(i + 1, i) ∧ A[i] < A[q]
11: q := i
12: end if
13: I(i + 1, q)
14: i = i + 1
15: I(i, q)
16: end while
17: I(i, q) ∧ i > n
18: Q(q)
19: return q
20: end procedure
```

Correction partielle

- Postcondition : $Q(j,q) \iff \forall k \mid j \leq k \leq n, A[q] \leq A[k]$
- Invariant : $I(i,q) \iff \forall k \mid j \leq k \leq i-1, A[q] \leq A[k]$
- $I(i+1,q) \iff \forall k \mid j \leq k \leq i, A[q] \leq A[k]$
- $I(i+1,i) \iff \forall k \mid j \leq k \leq i, A[i] \leq A[k]$
- $I(i,q) \wedge i \leq n \implies I(i+1,i) \wedge A[i] < A[q] \vee I(i+1,q) \wedge A[i] \geq A[q]$
- Précondition : $P \iff I(j+1,j) \iff \forall k \mid j \leq k \leq j, A[j] \leq A[k] \iff \text{true}$

Le tri par sélection : correction de TRISELECTION


```
Require: A un tableau de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(T,n)
2: for k = 1 to n - 1 do
3: q = INDICEMIN(A,n,k)
4: if q ≠ k then
5: ECHANGE(A[q],A[k])
6: end if
7: end for
8: end procedure
```

Terminaison

- La terminaison est évidente, il y a juste une boucle for dans cette procédure.
- Le variant de boucle est donc $n - k - 1$ qui est bien une suite strictement décroissante et positive.

Le tri par sélection : correction de TRISELECTION


```
Require: A un tableau de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(T,n)
2: for k = 1 to n - 1 do
3: q = INDICEMIN(A, n, k)
4: if q ≠ k then
5: ECHANGE(A[q],A[k])
6: end if
7: end for
8: Q
9: end procedure
```

Correction partielle

$$Q \iff \forall r \mid 1 < r \leq n, A[r-1] \leq A[r]$$

Le tri par sélection : correction de TRISELECTION


```
Require: A un tableau de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(A,n)
2: k = 1
3: while k ≤ n - 1 do
4: q = INDICEMIN(A,n,k)
5: if q ≠ k then
6: ECHANGE(A[q],A[k])
7: end if
8: k = k + 1
9: end while
10: Q
11: end procedure
```

Correction partielle

$$Q \iff \forall r \mid 1 < r \leq n, A[r-1] \leq A[r]$$

Le tri par sélection : correction de TRISELECTION


```
Require: A un tableau de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(A,n)
2: k = 1
3: I(k)
4: while k ≤ n - 1 do
5: I(k) ∧ k ≤ n - 1
6: q = INDICEMIN(A,n,k)
7: if q ≠ k then
8: ECHANGE(A[q],A[k])
9: end if
10: k = k + 1
11: I(k)
12: end while
13: I(k) ∧ k = n
14: Q
15: end procedure
```

Correction partielle

- $Q \iff \forall r \mid 1 < r \leq n, A[r-1] \leq A[r]$
- Invariant : $I(k) = I_1(k) \wedge I_2(k)$
- $I_1(k) \iff \forall r \mid 1 < r \leq k, A[r-1] \leq A[r]$
- $I_2(k) \iff \forall r, p \mid 1 < r \leq k < p \leq n, A[p] \geq A[r]$

Le tri par sélection : correction de TRISELECTION


```
Require: A un tableau de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(A,n)
2: k = 1
3: I(k)
4: while k ≤ n - 1 do
5: I(k) ∧ k ≤ n - 1
6: q = INDICEMIN(A,n,k)
7: if q ≠ k then
8: ECHANGE(A[q],A[k])
9: end if
10: I(k + 1)
11: k = k + 1
12: I(k)
13: end while
14: I(k) ∧ k = n
15: Q
16: end procedure
```

Correction partielle

- $Q \iff \forall r \mid 1 < r \leq n, A[r-1] \leq A[r]$
- Invariant : $I(k) = I_1(k) \wedge I_2(k)$
- $I_1(k) \iff \forall r \mid 1 < r \leq k, A[r-1] \leq A[r]$
- $I_2(k) \iff \forall r, p \mid 1 < r \leq k < p \leq n, A[p] \geq A[r]$

Le tri par sélection : correction de TRISELECTION


```
Require: A un tableau de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(A,n)
2: k = 1
3: I(k)
4: while k ≤ n - 1 do
5: I(k) ∧ k ≤ n - 1
6: ⇒ I₁(k + 1) ∧ I₂*(k + 1)
7: q = INDICEMIN(A,n,k)
8: if q ≠ k then
9: ECHANGE(A[q],A[k])
10: end if
11: I(k + 1)
12: k = k + 1
13: I(k)
14: end while
15: I(k) ∧ k = n
16: Q
17: end procedure
```

Correction partielle

- $Q \iff \forall r \mid 1 < r \leq n, A[r-1] \leq A[r]$
- Invariant : $I(k) = I_1(k) \wedge I_2(k)$
  - $I_1(k) \iff \forall r \mid 1 < r \leq k, A[r-1] \leq A[r]$
  - $I_2(k) \iff \forall r, p \mid 1 < r \leq k < p \leq n, A[p] \geq A[r]$
- $I_2^*(k)$ correspond à $I_2(k)$, mais avec le fait que $A[k]$ est permuté avec $A[q] (= \min_{i=k}^{n} A[i])$

Le tri par sélection : correction de TRISELECTION


```
Require: A un tableau de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(A,n)
2: k = 1
3: I(k)
4: while k ≤ n - 1 do
5: I(k) ∧ k ≤ n - 1
6: ⇒ I₁(k + 1) ∧ I₂*(k + 1)
7: q = INDICEMIN(A,n,k)
8: if q ≠ k then
9: ECHANGE(A[q],A[k])
10: end if
11: I(k + 1)
12: k = k + 1
13: I(k)
14: end while
15: I(k) ∧ k = n
16: Q
17: end procedure
```

Correction partielle

- $Q \iff \forall r \mid 1 < r \leq n, A[r-1] \leq A[r]$
- Invariant : $I(k) = I_1(k) \wedge I_2(k)$
  - $I_1(k) \iff \forall r \mid 1 < r \leq k, A[r-1] \leq A[r]$
  - $I_2(k) \iff \forall r, p \mid 1 < r \leq k < p \leq n, A[p] \geq A[r]$
- $I_2^*(k)$ correspond à $I_2(k)$, mais avec le fait que $A[k]$ est permuté avec $A[q] (= \min_{i=k}^{n} A[i])$

Le tri par sélection : correction de TRISELECTION


```
Require: A un tableau de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(A,n)
2: P
3: k = 1
4: I(k)
5: while k ≤ n - 1 do
6: I(k) ∧ k ≤ n - 1
7: ⇒ I₁(k + 1) ∧ I₂*(k + 1)
8: q = INDICEMIN(A,n,k)
9: if q ≠ k then
10: ECHANGE(A[q],A[k])
11: end if
12: I(k + 1)
13: k = k + 1
14: I(k)
15: end while
16: I(k) ∧ k = n
17: Q
18: end procedure
```

Correction partielle

- $Q \iff \forall r \mid 1 < r \leq n, A[r-1] \leq A[r]$
- Invariant : $I(k) = I_1(k) \wedge I_2(k)$
  - $I_1(k) \iff \forall r \mid 1 < r \leq k, A[r-1] \leq A[r]$
  - $I_2(k) \iff \forall r, p \mid 1 < r \leq k < p \leq n, A[p] \geq A[r]$
- $I_2^*(k)$ correspond à $I_2(k)$, mais avec le fait que $A[k]$ est permuté avec $A[q] (= \min_{i=k}^{n} A[i])$
- Précondition : $P \iff I(1) \iff \text{true}$

Le tri par sélection : complexité de INDICEMIN et ÉCHANGE


```
Require: Un tableau A de taille n et j ≤ n
Ensure: Renvoie q = argmin A[k]
                    j≤k≤n
1: procedure INDICEMIN(A,n,j)
2: q := j
3: for i = j + 1 to n do
4: if A[i] < A[q] then
5: q := i
6: end if
7: end for
8: return q
9: end procedure
```

```
Require: a et b deux valeurs
1: procedure ÉCHANGE(a,b)
2: tmp = a
3: a = b
4: b = tmp
5: end procedure
```

INDICEMIN

- n - j itérations dans la boucle for
- pire des cas : $T^{\text{im}}(n;j) = c_1^{\text{im}} + (n-j)(c_1^{\text{im}} + c_2^{\text{im}} + c_3^{\text{im}})$
- meilleur des cas : $T^{\text{im}}(n;j) = c_1^{\text{im}} + (n-j)(c_2^{\text{im}} + c_3^{\text{im}})$
- Dans tous les cas : $T^{\text{im}}(n;j) \in \Theta(n)$

ÉCHANGE

- 3 affectations : $T(n) = 3c_1^e$
- Dans tous les cas : $T^{e(n)} \in \Theta(1)$

Le tri par sélection : complexité de TRISELECTION


```
Require: A un tableau à trier de taille n
Ensure: Le tableau A est trié
1: procedure TRISELECTION(A,n)
2: for k = 1 to n - 1 do ▷ c_1
3: q = INDICEMIN(A, n, k) ▷ c_2 + T^im
4: if q ≠ k then ▷ c_3
5: ECHANGE(A[q],A[k]) ▷ T^e
6: end if
7: end for
8: end procedure
```

- $n-1$ iterations dans la boucle for
- pire des cas :
- $n-1$ appel à ÉCHANGE
- Par exemple lorsqu'on trie la série [2, 3, ..., n, 1]
- meilleur des cas :

Aucun appel à ÉCHANGE Tableau déjà trié

- Complexité dans le pire des cas :

$$T(n) = (n-1)\big(c_1 + c_2 + c_3 + T^{e(n)}\big) + \sum_{k=1}^{n-1} T^{\text{im}}(n,k)$$

$$= (n-1)(c_1 + c_2 + c_3 + 3c_1^e) + (c_1^{\text{im}} + c_2^{\text{im}} + c_3^{\text{im}})\frac{n(n-1)}{2} + (n-1)c_1^{\text{im}}$$

$$\Longrightarrow T(n) \in \Theta(n^2)$$

- Idem pour le meilleur des cas et donc aussi en moyenne

Le tri par sélection : synthèse

	Complexité
meilleur des cas	$\Theta(n^2)$
cas moyen	$\Theta(n^2)$
pire des cas	$\Theta(n^2)$

- Complexité quadratique (donc peu efficace)
- Coût quadratique en terme de comparaisons.
- Avantage : le nombre réduit d'échanges effectués (au pire $n-1$).
- Algorithmme en place.

Conclusion

Il présente un intérêt sur des données coûteuses à déplacer mais qui sont aisément comparables.

Le tri par sélection : code C++


```
1 inline int indiceMin(int * A, int n, int j)
2
3 int q = j;
4 for (int i=j+1;i<n;++i)
5
6 if ( A[i] < A[q] )
7 q = i;
8
9 return q;
10
11
12 void triParSelection(int * A, int n)
13
14 int q, tmp;
15 for (int k=0;k<(n-1);++k)
16
17 q = indiceMin(A,n,k);
18 if ( q!=k)
19
20 tmp = A[q];
21 A[q] = A[k];
22 A[k] = tmp;
23
24
25
```

Le tri par insertion : le principe

Le principe du tri par insertion est utilisé naturellement par la plupart des personnes qui veulent trier des cartes à jouer. On commence par trier les deux premières cartes l'une par rapport à l'autre, puis on trie la troisième par rapport aux deux premières, et ainsi de suite jusqu'à ce que toutes les cartes soient triées.

- On prend le premier de la séquence non ordonnée, ce qui constitue une nouvelle séquence ordonnée composée d'un seul élément.

A[1] Tableau non ordonné

- On parcourt ensuite le reste de la séquence non ordonnée de gauche à droite et pour chaque élément :
- On parcourt la série ordonnée de droite à gauche
- On l'insère dès que l'on trouve sa position

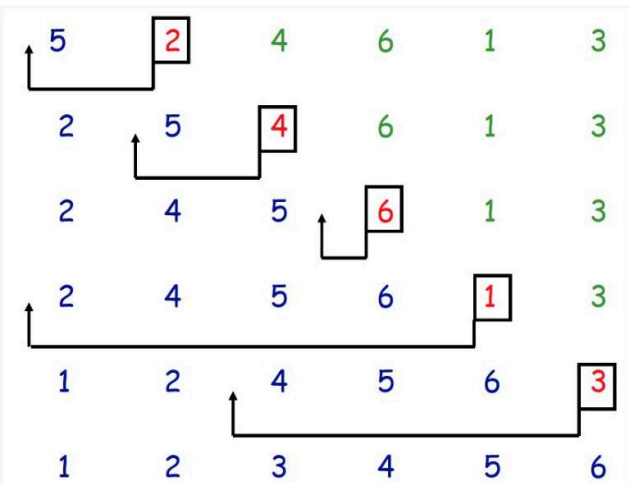
ordonné

non ordonné

- On s'arrête dès que le dernier élément a été inséré



Le tri par insertion : le principe



Tri par insertion : pseudo-code et correction


```
Require: A tableau de taille n
Ensure: Le tableau A est trié
1: for j = 2 to n do
2: key = A[j]
3: i = j - 1
4: while i > 0 and A[i] > key do
5: A[i + 1] = A[i]
6: i = i - 1
7: end while
8: A[i + 1] = key
9: end for
```

Terminaison

- Boucle for : le variant est $n - j$
- Boucle while : le variant est i

Correction partielle

La preuve nécessite de définir en particulier les 2 invariants de boucle :

- Boucle for :

$$\forall k \mid 1 < k < j, A[k-1] \leq A[k]$$

- Boucle while :

$$\forall k \mid i < k < j, \text{key} < A[k]$$

Le tri par insertion : complexité

9: end for

Tableau trié

- j=2 13 14 15 16 17 (0 iter)
- j=3 13 14 15 16 17 (0 iter)
- j=4 13 14 15 16 17 (0 iter)
- j=5 13 14 15 16 17 (0 iter)

Meilleur des cas

- Le tableau est déjà trié
- La condition A[i] > key sera toujours fausse
- $T(n) = (n-1)(c_1 + c_2 + c_3 + c_4 + c_7) + c_1$
- $T(n) \in \Theta(n)$

```
Require: A tableau de taille n
1: for j = 2 to n do
2: key = A[j]
3: i = j - 1
4: while i > 0 and A[i] > key do
5: A[i + 1] = A[i]
6: i = i - 1
7: end while
8: A[i + 1] = key
```

Le tri par insertion : complexité


```
Require: A tableau de taille n
1: for j = 2 to n do
2: key = A[j] ▷ c_2
3: i = j - 1 ▷ c_3
4: while i > 0 and A[i] > key do ▷ c_4
5: A[i + 1] = A[i] ▷ c_5
6: i = i - 1 ▷ c_6
7: end while
8: A[i + 1] = key ▷ c_7
9: end for
```

Tableau trié en sens inverse

- j=2 17 16 15 14 13 (1 iter)
- j=3 16 17 15 14 13 (2 iter)
- j=4 15 16 17 14 13 (3 iter)
- j=5 14 15 16 17 13 (4 iter)

Pire des cas

- La boucle while s'arrête lorsque $i == 0$ ($A[i] > \text{key}$ sera toujours vrai)
- $T(n) = (n-1) * (c_1 + c_2 + c_3 + c_7) + (c_4 + c_5 + c_6) * \sum_{k=1}^{n-1} k + c_1$

$= (n-1) * (c_1 + c_2 + c_3 + c_7) + (c_4 + c_5 + c_6) * \left(n * \frac{n-1}{2}\right) + c_1$

- $T(n) \in \Theta(n^2)$

Le tri par insertion : complexité

9: end for

Exemple de tableau du cas moyen

- j=2 14 13 15 17 16 (1 iter)
- j=3 13 14 15 17 16 (0 iter)
- j=4 13 14 15 17 16 (0 iter)
- j=5 13 14 15 17 16 (1 iter)

Cas moyen (avec équiprobabilité)

- La moitié du tableau $A[1..j-1]$ est $< A[j]$ et l'autre est $> A[j]$.
- En moyenne on teste la moitié du sous-tableau $A[1..j-1]$
- $T(n) = (n-1) * (c_1 + c_2 + c_3 + c_7) + (c_4 + c_5 + c_6) * \sum_{k=1}^{n-1}(\frac{k}{2}) + c_1$. $T(n) \in \Theta(n^2)$

```
Require: A tableau de taille n
1: for j = 2 to n do
2: key = A[j]
3: i = j - 1
4: while i > 0 and A[i] > key do
5: A[i + 1] = A[i]
6: i = i - 1
7: end while
8: A[i + 1] = key
```

Le tri par insertion : synthèse

	Complexité
meilleur des cas	$\Theta(n)$
cas moyen	$\Theta(n^2)$
pire des cas	$\Theta(n^2)$

- Complexité quadratique en moyenne (donc peu efficace sur des grands tableaux)
- Cependant, on pourrait démontrer que dans le cas d'un petit nombre d'éléments à trier, c'est le plus rapide en moyenne.
- Il est également très efficace lorsque le tableau est déjà presque trié car la complexité est en $\Theta(n)$.
- Algorithmme en place.

Conclusion

Il présente un intérêt sur des tableaux de petite taille.

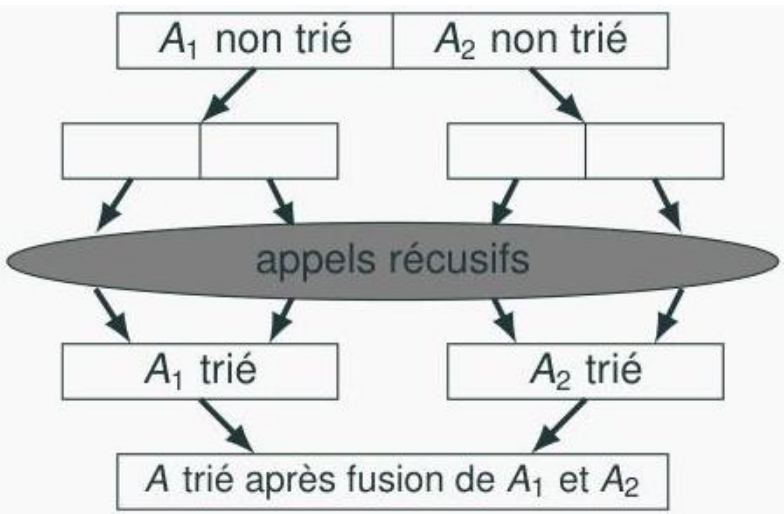
Le tri par insertion : code C++


```
1 void triParInsertion(int * A, int n)
2
3 int key, i;
4 for ( int j = 1; j < n ; j++)
5
6 key = A[j];
7 i = j-1;
8 while ( i >=0 && A[i] > key )
9
10 A[i+1] = A[i];
11 i = i-1;
12
13 A[i+1] = key;
14
15
```

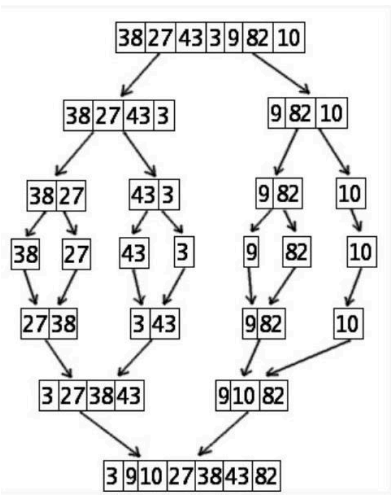
Le tri fusion : le principe

L'algorithme du tri fusion est un exemple typique des algorithmes de type « diviser pour régner ». Pour trier un tableau de taille n, son principe consiste à partager le tableau en deux parties de tailles égales (ou presque si n est impair). Ces 2 sous-ensembles sont triés par appel récursif, puis on fusionne les deux parties triées.

- Les propriétés de cet algorithme reposent en particulier sur une fonction auxiliaire, la fonction de fusion.
- La fonction Fusion fusionnera deux tableaux déjà triés, et c'est grâce à cette propriété que cette fonction pourra être efficace.



Le tri fusion : le principe



Le tri fusion : pseudo-code et correction de TRIFUSION

La division en deux sous-tableaux de taille égale ou presque est faite en définissant

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1$$

La première partition sera composée des n_1 premiers éléments, l'autre contiendra les n_2 restants. Ainsi, si n est pair le partage sera égal, sinon la première partition aura un élément de plus.

Terminaison

- Soit $(u_k)_{k \geq 0}$ la suite définie par $u_0 = n$ et u_k le nombre maximal d'éléments à trier entre les 2 appels récursifs à TRIFUSION à la récursion k.
- Tant que $u_k > 1$, $u_{k+1} = \left\lfloor \left(\frac{u_k+1}{2} \right) \right\rfloor \leq \left(\frac{u_k+1}{2} \right) < u_k$
- Par récurrence, on en déduit que $(u_k)_{k \geq 0}$ est strictement décroissante et positive.

```
Require: A tableau de taille n et deux indices p et r de A
1: procedure TRIFUSION(A,p,r)
2: if p < r then
3: q = ⌊ (p + r) / 2⌋
4: TRIFUSION(A,p,q)
5: TRIFUSION(A,q+1,r)
6: FUSION(A,p,q,r)
7: end if
8: end procedure
```

Le tri fusion : pseudo-code et correction de TRIFUSION

La division en deux sous-tableaux de taille égale ou presque est faite en définissant

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1$$

La première partition sera composée des n_1 premiers éléments, l'autre contiendra les n_2 restants. Ainsi, si n est pair le partage sera égal, sinon la première partition aura un élément de plus.

Correction partielle : preuve par induction

- Si $n = 0$ ou $n = 1$, le tableau est déjà trié.
- Si $n \geq 2$, l'exécution des deux appels récursifs de TRIFUSION permet de définir les 2 sous-tableaux (qui sont triés par hypothèse inductive)
- $A_1 = A[p : q]$
- $A_2 = A[q + 1 : r]$
- Nous pouvons donc utiliser FUSION et obtenir un tableau trié.

```
Require: A tableau de taille n et deux indices p et r de A
1: procedure TRIFUSION(A,p,r)
2: if p < r then
3: q = ⌊ (p + r) / 2⌋
4: TRIFUSION(A,p,q)
5: TRIFUSION(A,q+1,r)
6: FUSION(A,p,q,r)
7: end if
8: end procedure
```

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Initialisation

- $n_1 = 4$ et $n_2 = 3$
- $i_1 = 1$ et $i_2 = 1$

14	16	31	54	8	27	30

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 1

- $i_1 = 1$ et $i_2 = 1$

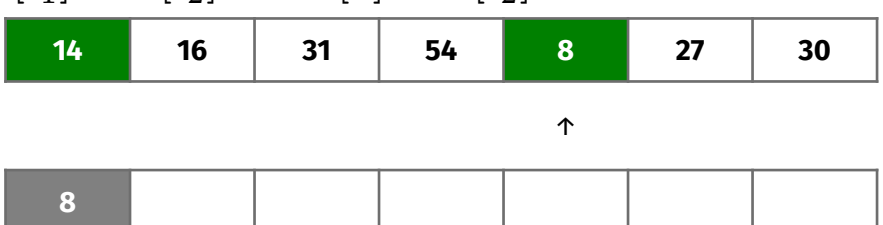
14	16	31	54	8	27	30

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 1

- $i_1 = 1$ et $i_2 = 1$
- $A[i_1] \geq A[i_2] \Longrightarrow B[k] = A[i_2]$



Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 2

- $i_1 = 1$ et $i_2 = 2$

14	16	31	54	8	27	30

8						

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 2

- $i_1 = 1$ et $i_2 = 2$
- $A[i_1] < A[i_2] \Longrightarrow B[k] = A[i_1]$

14	16	31	54	8	27	30

↑

8	14					

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 3

- $i_1 = 2$ et $i_2 = 2$

14	16	31	54	8	27	30

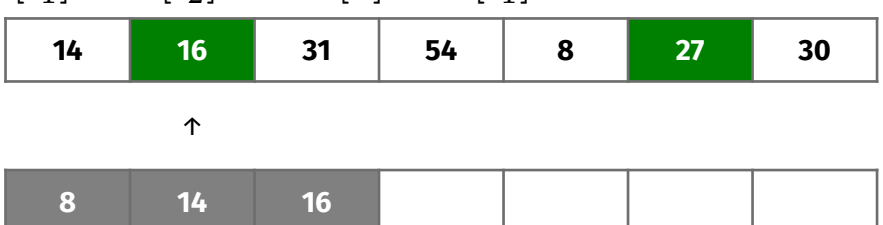
8	14					

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 3

- $i_1 = 2$ et $i_2 = 2$
- $A[i_1] < A[i_2] \Longrightarrow B[k] = A[i_1]$



Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 4

- $i_1 = 3$ et $i_2 = 2$

14	16	31	54	8	27	30

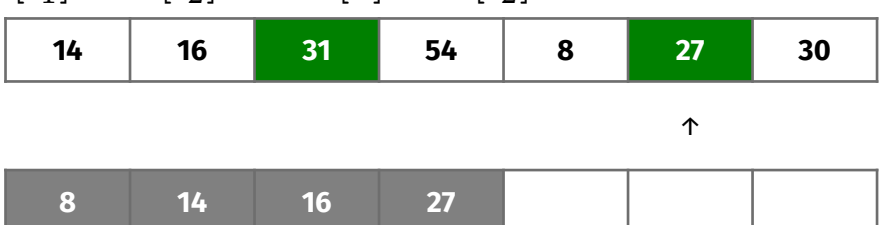
8	14	16				

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 4

- $i_1 = 3$ et $i_2 = 2$
- $A[i_1] \geq A[i_2] \Longrightarrow B[k] = A[i_2]$



Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 5

- $i_1 = 3$ et $i_2 = 3$

14	16	31	54	8	27	30

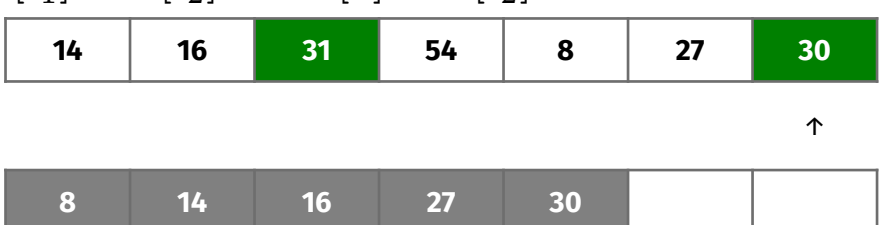
8	14	16	27			

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 5

- $i_1 = 3$ et $i_2 = 3$
- $A[i_1] \geq A[i_2] \Longrightarrow B[k] = A[i_2]$



Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 6

- $i_1 = 3 (\text{et} \quad i_2 = 4 > n_2)$

14	16	31	54	8	27	30

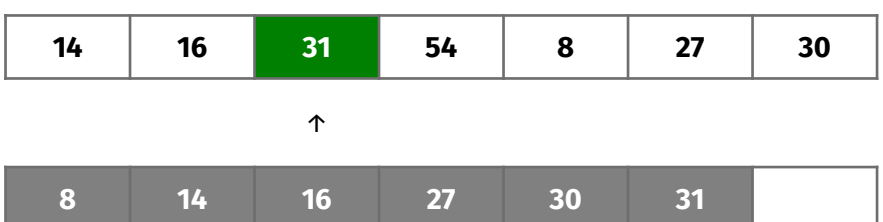
8	14	16	27	30		

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 6

- $i_1 = 3$ (et $i_2 = 4 > n_2$)
- B[k] = A[i_1]



Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 7

- $i_1 = 4 (\text{et} \quad i_2 = 4 > n_2)$

14	16	31	54	8	27	30

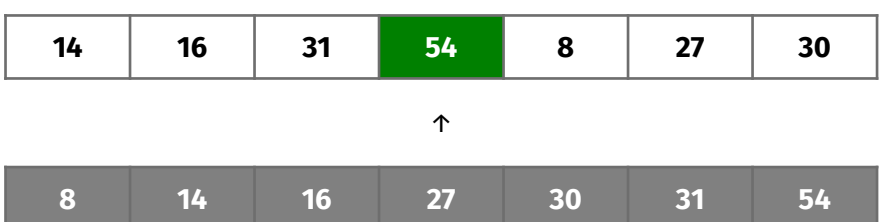
8	14	16	27	30	31	

Le tri fusion : principe et pseudo-code de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i₁ = p, i₂ = q + 1, k = 1
3: n₁ = q - p + 1, n₂ = r - q
4: B = createTab(n₁ + n₂)
5: while i₁ ≤ q and i₂ ≤ r do
6: if A[i₁] < A[i₂] then
7: B[k] = A[i₁]
8: i₁ = i₁ + 1; k = k + 1
9: else
10: B[k] = A[i₂]
11: i₂ = i₂ + 1; k = k + 1
12: end if
13: end while
14: while i₁ ≤ q do
15: B[k] = A[i₁]
16: i₁ = i₁ + 1, k = k + 1
17: end while
18: while i₂ ≤ r do
19: B[k] = A[i₂]
20: i₂ = i₂ + 1, k = k + 1
21: end while
22: for k = 1 to n₁ + n₂ do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Itération k = 7

- $i_1 = 4$ (et $i_2 = 4 > n_2$)
- B[k] = A[i_1]



Le tri fusion : complexité de FUSION


```
1: procedure FUSION(A,p,q,r)
2: i1 = p, i2 = q + 1, k = 1
3: n1 = q - p + 1, n2 = r - q
4: B = createTab(n1 + n2)
5: while i1 ≤ q and i2 ≤ r do
6: if A[i1] < A[i2] then
7: B[k] = A[i1]
8: i1 = i1 + 1; k = k + 1
9: else
10: B[k] = A[i2]
11: i2 = i2 + 1; k = k + 1
12: end if
13: end while
14: while i1 ≤ q do
15: B[k] = A[i1]
16: i1 = i1 + 1, k = k + 1
17: end while
18: while i2 ≤ r do
19: B[k] = A[i2]
20: i2 = i2 + 1, k = k + 1
21: end while
22: for k = 1 to n1 + n2 do
23: A[p + k] = B[k]
24: end for
25: end procedure
```

Terminaison

Variant de boucle :

- $(n_1 - i_1) * (n_2 - i_2)$ $n_1 - i_1$ et $n_2 - i_2$

Correction partielle

Invariant de boucle :

- $\forall r \mid 1 < r < k, B[r-1] \leq B[r]$
- si $k > 1$ et $i_1 \leq n$: $B[k-1] \leq A[i_1]$
- si $k > 1$ et $i_2 \leq n$: $B[k-1] \leq A[i_2]$

Complexité

- Dans le pire ou meilleur des cas : $T^{f(n)} = a(n_1 + n_2) + bn_1 + cn_2 + d.$
- La complexité est donc $O(n_1 + n_2) = O(n)$.

Le tri fusion : complexité de TRIFUSION

Pour tous les cas

$$T(n) = 2T\left(\frac{n}{2}\right) + T^{f(n)} + c_1 + c_2$$

$$T(0) = T(1) = c_1$$

- La complexité de l'algorithme Fusion est $T^{f(n)} \in O(n)$.
- On peut donc appliquer le Master théorème avec $a = 2$, $b = 2$, $d = 1$.
- On en déduit que la complexité du tri par fusion: $T(n) \in \Theta(n \log(n))$

```
Require: A tableau de taille n et deux indices p et r de A
1: procedure TRIFUSION(A,p,r)
2: if p < r then ▷ c_1
3: q = ⌊ (p + r) / 2⌋ ▷ c_2
4: TRIFUSION(A,p,q)
5: TRIFUSION(A,q+1,r)
6: FUSION(A,p,q,r)
7: end if
8: end procedure
```

Le tri fusion : synthèse

	Complexité
meilleur des cas	$\Theta(n \log(n))$
cas moyen	$\Theta(n \log(n))$
pire des cas	$\Theta(n \log(n))$

- Avantage : sa complexité est la même dans tous les cas et est assez rapide
- Inconvénient : ce n'est pas un tri en place ($\Theta(n)$ en espace) C'est un tri stable

Le tri fusion : code C++


```
1 void fusion(int * A, int p, int q, int r)
2
3 int i1=p, i2=q+1, k=0;
4 int n1=q-p+1, n2=r-q;
5 int * B = new int[n1+n2];
6 while (i1<=q && i2<=r)
7
8 if (A[i1] < A[i2])
9
10 B[k] = A[i1];
11 ++i1;++k;
12
13 else
14
15 B[k] = A[i2];
16 ++i2;++k;
17
18
19 while (i1<=q)
20
21 B[k] = A[i1];
22 ++i1;++k;
23
24 while (i2<=r)
25
26 B[k] = A[i2];
27 ++i2;++k;
28
29 for (k=0;k<(n1+n2);++k)
30
31 A[p+k] = B[k];
32
33 delete [] B;
34
```

```
1 void triFusionRec(int * A, int p, int r)
2
3 if ( p < r )
4
5 int q = (p+r)/2;
6 triFusionRec( A,p,q);
7 triFusionRec( A,q+1,r);
8 fusion(A,p,q,r);
9
10
11 void triFusion(int * A, int n)
12
13 triFusionRec(A,0,n-1);
14
```

Le tri rapide : le principe

L'algorithme de tri rapide est aussi un algorithme de type « diviser pour régner ». Son principe consiste à séparer l'ensemble des éléments en deux parties. La différence par rapport au tri fusion est que la séparation ne s'effectue pas au milieu mais par rapport à une valeur appelée pivot.

- Choisir arbitrairement dans le tableau un élément pivot q :

	q	

- Enlever le pivot de la liste, et segmenter le tableau en deux sous-tableaux l'un contenant les éléments plus petits ou égaux à q, l'autre les éléments restants strictement supérieurs au pivot q :

≤ q	q	> q

- Recommencer récursivement avec les tableaux situés à gauche et à droite du pivot, puis rassembler le tout :

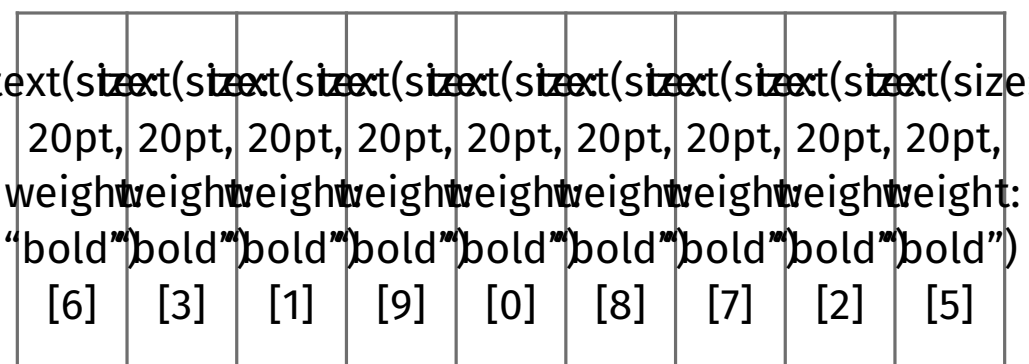
tri rapide	q	tri rapide

Dans ce cours, le pivot q sera choisi comme le dernier élément de la liste.

Le tri rapide : le principe

text(size: 20pt, weight: “bold”)[6]	text(size: 20pt, weight: “bold”)[3]	text(size: 20pt, weight: “bold”)[1]	text(size: 20pt, weight: “bold”)[9]	text(size: 20pt, weight: “bold”)[0]	text(size: 20pt, weight: “bold”)[8]	text(size: 20pt, weight: “bold”)[7]	text(size: 20pt, weight: “bold”)[2]	text(size: 20pt, weight: “bold”)[5]

Le tri rapide : le principe



Le tri rapide : le principe

6	3	1	9	0	8	7	2	5

3	1	0	2	5	6	9	8	7

Le tri rapide : le principe

6	3	1	9	0	8	7	2	5

3	1	0	2	5	6	9	8	7

Le tri rapide : le principe

6	3	1	9	0	8	7	2	5

3	1	0	2	5	6	9	8	7

1	0	2	3	5	6	7	9	8

Le tri rapide : le principe

6	3	1	9	0	8	7	2	5

3	1	0	2	5	6	9	8	7

1	0	2	3	5	6	7	9	8

Le tri rapide : le principe

6	3	1	9	0	8	7	2	5

3	1	0	2	5	6	9	8	7

1	0	2	3	5	6	7	9	8

Le tri rapide : le principe

6	3	1	9	0	8	7	2	5

3	1	0	2	5	6	9	8	7

1	0	2	3	5	6	7	9	8

0	1	2	3	5	6	7	8	9

Le tri rapide : le principe

6	3	1	9	0	8	7	2	5

3	1	0	2	5	6	9	8	7

1	0	2	3	5	6	7	9	8

0	1	2	3	5	6	7	8	9

La procédure PARTITION choisit le pivot comme étant la valeur du dernier élément du tableau et réordonne ce tableau de telle manière que tous les éléments inférieurs ou égaux au pivot sont positionnés avant et les autres strictement supérieurs après ce pivot.

≤ q	q	> q

Terminaison

- Soit $(u_k)_{k>0}$ la suite du nombre maximal d'éléments à trier entre les 2 appels récursifs à TRIRAPIDE, i.e. $u_k = \max(q-p, r-q)$.
- $\forall k$, on $u_{k+1} < u_k$ car même si tous les éléments restants sont du même côté du pivot, on a retiré un élément : le pivot
- La suite $(u_k)_{k>0}$ est une suite strictement décroissante et positive

```
Require: Un tableau A de taille n et deux indices p et r de A
1: procedure TRIRAPIDE(A,p,r)
2: if p < r then
3: q = PARTITION(A,p,r)
4: TRIRAPIDE(A,p,q-1)
5: TRIRAPIDE(A,q+1,r)
6: end if
7: end procedure
```

Le tri rapide : pseudo-code et correction de TRIRAPIDE

La procédure PARTITION choisit le pivot comme étant la valeur du dernier élément du tableau et réordonne ce tableau de telle manière que tous les éléments inférieurs ou égaux au pivot sont positionnés avant et les autres strictement supérieurs après ce pivot.

≤ q	q	> q

Correction partielle : preuve par induction

- Si $n = 0$ ou $n = 1$, le tableau est déjà trié.
- Si $n \geq 2$, l'exécution de la procédure PARTITION permet de définir les 2 sous-tableaux :
- $A_1 = A[p : q-1]$ dont tous les éléments sont $\leq q$
- $A_2 = A[q+1 : r]$ dont tous les éléments sont $> q$
- Par hypothèse inductive, le tableau représenté par la concaténation TRIRAPIDE(A_1) + [q] + TRIRAPIDE(A_2) est un tableau trié.

```
Require: Un tableau A de taille n et deux indices p et r de A
1: procedure TRIRAPIDE(A,p,r)
2: if p < r then
3: q = PARTITION(A,p,r)
4: TRIRAPIDE(A,p,q-1)
5: TRIRAPIDE(A,q+1,r)
6: end if
7: end procedure
```

Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Initialisation

On sélectionne le pivot comme étant le dernier élément.

6	5	2	1	3	4

Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Itération j = 1

On compare 6 et 4, 6 est plus grand, rien à faire, c'est le premier plus grand, en position 0.

6	5	2	1	3	4

Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Itération j = 2

On compare 5 et 4, 5 est plus grand, rien à faire, c'est le 2ème plus grand, en position 1.

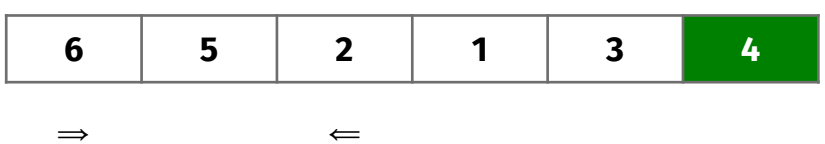
6	5	2	1	3	4

Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Itération j = 3

On compare 2 et 4, 2 est plus petit, on le permute avec le premier plus grand (ici 6).



Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Itération j = 3

On compare 2 et 4, 2 est plus petit, on le permute avec le premier plus grand (ici 6).

6	5	2	1	3	4

⇒ ⇐

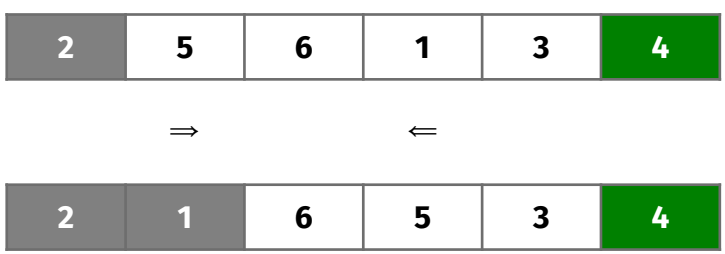
2	5	6	1	3	4

Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Itération j = 4

On compare 1 et 4, 1 est plus petit, on permute avec le 1er plus grand (qui est 5 ici).

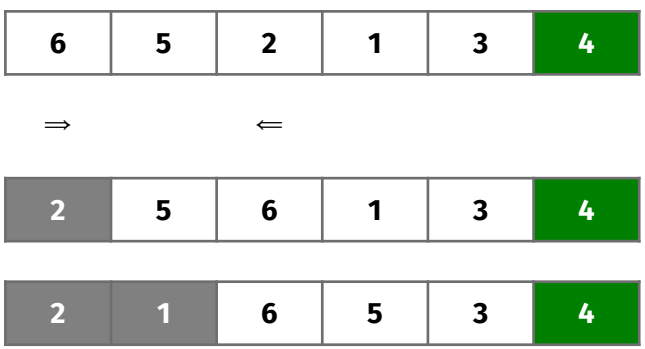


Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Itération j = 4

On compare 1 et 4, 1 est plus petit, on permute avec le 1er plus grand (qui est 5 ici).

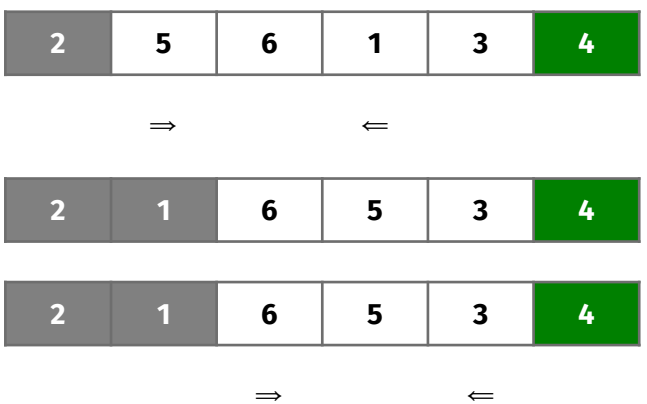


Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Itération j = 5

On compare 3 et 4, 3 est plus petit, on permute avec le 1er plus grand (qui est 6 ici).

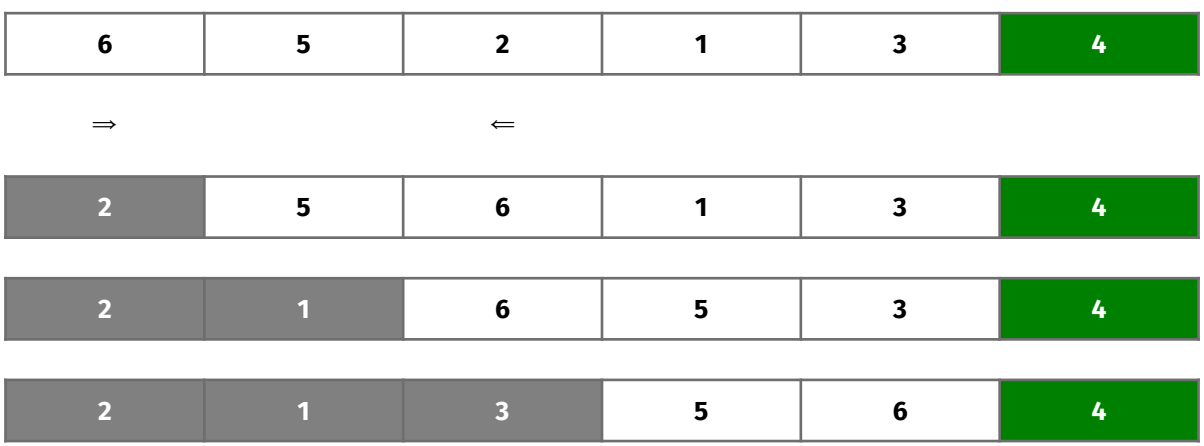


Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Itération j = 5

On compare 3 et 4, 3 est plus petit, on permute avec le 1er plus grand (qui est 6 ici).

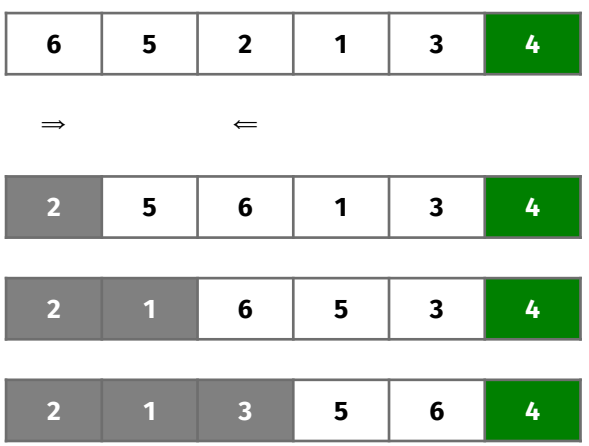


Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Finalisation

On termine en posant le pivot (ici 4) à sa position, c'est-à-dire à la place du premier plus grand (ici 5).

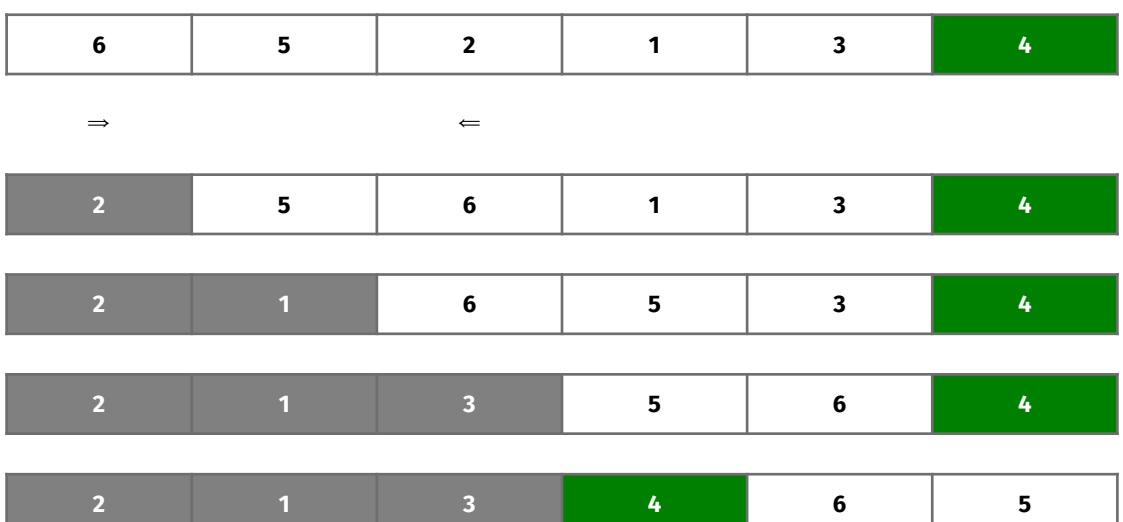


Le tri rapide : pseudo-code de PARTITION


```
1: procedure PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i = i + 1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Finalisation

On termine en posant le pivot (ici 4) à sa position, c'est-à-dire à la place du premier plus grand (ici 5).



Le tri rapide : correction de PARTITION


```
1: procedure
PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i=i+1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: return i
12: end procedure
```

Terminaison

- La terminaison est évidente, il y a juste une boucle for dans cette procédure.
- Le variant de boucle est donc $r - 1 - j$ qui est bien une suite strictement décroissante et positive.

Le tri rapide : correction de PARTITION


```
1: procedure
PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: for j = p to r - 1 do
5: if A[j] ≤ pivot then
6: ECHANGE(A[i],A[j])
7: i=i+1
8: end if
9: end for
10: ECHANGE(A[i],A[r])
11: Q
12: return i
13: end procedure
```

Terminaison

- La terminaison est évidente, il y a juste une boucle for dans cette procédure.
- Le variant de boucle est donc $r - 1 - j$ qui est bien une suite strictement décroissante et positive.

Correction partielle

- Postcondition $Q = Q_1 \wedge Q_2$:
- $Q_1 \iff \forall k \mid p \leq k < i, A[k] \leq A[i]$
- $Q_2 \iff \forall k \mid i < k \leq r, A[k] > A[i]$

Le tri rapide : correction de PARTITION


```
1: procedure
PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: j = p
5: while j ≤ r - 1 do
6: if T[j] ≤ pivot then
7: ECHANGE(A[i],A[j])
8: i = i + 1
9: end if
10: j = j + 1
11: end while
12: ECHANGE(A[i],A[r])
13: Q
14: return i
15: end procedure
```

Terminaison

- La terminaison est évidente, il y a juste une boucle for dans cette procédure.
- Le variant de boucle est donc $r - 1 - j$ qui est bien une suite strictement décroissante et positive.

Correction partielle

- Postcondition $Q = Q_1 \wedge Q_2$:
- $Q_1 \iff \forall k \mid p \leq k < i, A[k] \leq A[i]$
- $Q_2 \iff \forall k \mid i < k \leq r, A[k] > A[i]$

Le tri rapide : correction de PARTITION


```
1: procedure
PARTITION(A,p,r)
2: pivot = A[r]
3: i = p
4: j = p
5: I
6: while j ≤ r - 1 do
7: I ∧ j ≤ r - 1
8: if T[j] ≤ pivot then
9: ECHANGE(A[i],A[j])
10: i = i + 1
11: end if
12: j = j + 1
13: I
14: end while
15: I ∧ j = r
16: ECHANGE(A[i],A[r])
17: Q
18: return i
19: end procedure
```

Terminaison

- La terminaison est évidente, il y a juste une boucle for dans cette procédure.
- Le variant de boucle est donc $r - 1 - j$ qui est bien une suite strictement décroissante et positive.

Correction partielle

- Postcondition $Q = Q_1 \wedge Q_2$: $\bullet\ Q_1 \iff \forall k \mid p \leq k < i, A[k] \leq A[i]$ $\bullet\ Q_2 \iff \forall k \mid i < k \leq r, A[k] > A[i]$
- Invariant $I = I_1 \wedge I_2$: $\bullet\ I_1 \iff \forall k \mid p \leq k < i, A[k] \leq A[r]$ $\bullet\ I_2 \iff \forall k \mid i \leq k < j, A[k] > A[r]$

Le tri rapide : correction de PARTITION


```
1: procedure
PARTITION(A,p,r)
2: pivot = A[r]
3: true
4: i = p
5: j = p
6: I
7: while j ≤ r - 1 do
8: I ∧ j ≤ r - 1
9: if T[j] ≤ pivot then
10: ECHANGE(A[i],A[j])
11: i = i + 1
12: end if
13: j = j + 1
14: I
15: end while
16: I ∧ j = r
17: ECHANGE(A[i],A[r])
18: Q
19: return i
20: end procedure
```

Terminaison

- La terminaison est évidente, il y a juste une boucle for dans cette procédure.
- Le variant de boucle est donc $r - 1 - j$ qui est bien une suite strictement décroissante et positive.

Correction partielle

- Postcondition $Q = Q_1 \wedge Q_2$:
- $Q_1 \iff \forall k \mid p \leq k < i, A[k] \leq A[i]$
- $Q_2 \iff \forall k \mid i < k \leq r, A[k] > A[i]$
- Invariant $I = I_1 \wedge I_2$:
- $I_1 \iff \forall k \mid p \leq k < i, A[k] \leq A[r]$
- $I_2 \iff \forall k \mid i \leq k < j, A[k] > A[r]$
- Précondition P = true

Le tri rapide : complexité de PARTITION


```
2: pivot = A[r] ▷ c1^p
3: i = p ▷ c2^p
4: for j = p to r - 1 do ▷ c3^p
5: if T[j] ≤ pivot then ▷ c4^p
6: ECHANGE(A[i],A[j]) ▷ c5^p
7: i = i + 1 ▷ c6^p
8: end if
9: ECHANGE(A[i],A[r]) ▷ c5^p
10: end for
11: return i
12: end procedure
```

1: procedure PARTITION(A,p,r)

- n = r - p + 1
- r - p = n - 1 iterations
- $\text{ECHANGE} \in \Theta(1)$
- Pire des cas :

$$T^{p(n)} = c_1^p + c_2^p + (n-1)(c_2^p + c_3^p + 2c_5^p + c_6^p)$$

- Meilleur des cas :

$$T^{p(n)} = c_1^p + c_2^p + (n-1)(c_2^p + c_3^p + c_5^p)$$

- Dans tous les cas : $T^{p(n)} \in \Theta(n)$

Le tri rapide : complexité de TRIRAPIDE

- La complexité du tri rapide dépend de la distribution autour du pivot.
- Si $n \geq 2$, on obtient la relation de récurrence:

$$T(n) = c_1 + T^{p(n)} + T(q-p) + T(r-q)$$

- Cas de base :

$$T(1) = c_1$$

Pire des cas

- Le pivot est le min $(q = p)$ ou la $\max(q = r)$ du tableau
- Le partitionnement transforme un problème de taille n en un problème de taille $n-1$

$$T(n) = c_1 + T^{p(n)} + T(n-1) \iff T(n) - T(n-1) = c_1 + T^{p(n)}$$

$$\sum_{i=1}^{n} T(i) - T(i-1) = \sum_{i=1}^{n} c_1 + T^{p(i)} \implies T(n) - T(0) = \sum_{i=1}^{n} c_1 + T^{p(i)}$$

$$\sum_{i=1}^{n} c_1 + T^{p(i)} = \sum_{i=1}^{n} \alpha(i-1) + \beta = \alpha\left(\frac{n(n-1)}{2}\right) + \beta n \implies T(n) \in \Theta(n^2)$$

```
Require: Un tableau A de taille n et deux indices p et r
1: procedure TRIRAPIDE(A,p,r)
2: if p < r then ▷ c_1
3: q = PARTITION(A,p,r)
4: TRIRAPIDE(A,p,q-1)
5: TRIRAPIDE(A,q+1,r)
6: end if
7: end procedure
```

Le tri rapide : complexité de TRIRAPIDE

- La complexité du tri rapide dépend de la distribution autour du pivot.
- Si $n \geq 2$, on obtient la relation de récurrence: $T(n) = c_1 + T^{p(n)} + T(q-p) + T(r-q)$
- Cas de base : $T(1) = c_1$

Meilleur des cas

- Les deux sous-tableaux ont le même nombre d'éléments ($q = \lfloor \frac{n}{2} \rfloor$) si n est impair, sinon l'un d'eux a 1 élément de moins

$$T(n) = c_1 + T^{p(n)} + 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

- On peut appliquer le Master théorème avec $a = 2, b = 2, d = 1$ car $T^{p(n)} \in \Theta(n) \implies T(n) \in \Theta(n \log(n))$

```
Require: Un tableau A de taille n et deux indices p et r
1: procedure TRIRAPIDE(A,p,r)
2: if p < r then ▷ c1
3: q = PARTITION(A,p,r)
4: TRIRAPIDE(A,p,q-1)
5: TRIRAPIDE(A,q+1,r)
6: end if
7: end procedure
```

Le tri rapide : complexité de TRIRAPIDE


```
Require: Un tableau A de taille n et deux indices p et r
1: procedure TRIRAPIDE(A,p,r)
2: if p < r then ▷ c_1
3: q = P_ARTITION(A,p,r)
4: TRIRAPIDE(A,p,q-1)
5: TRIRAPIDE(A,q+1,r)
6: end if
7: end procedure
```

- La complexité du tri rapide dépend de la distribution autour du pivot.
- Si $n \geq 2$, on obtient la relation de récurrence:

$$T(n) = c_1 + T^{p(n)} + T(q-p) + T(r-q)$$

- Cas de base :

$$T(1) = c_1$$

En moyenne

- Le pivot dans la liste triée a autant de chances d'être sur chacune des n places
- Q la variable aléatoire de loi uniforme qui correspond à la position du pivot

$$T(n) = c_1 + T^{p(n)} + \mathrm{E}(T(Q-1) + T(n-Q-1))$$

$$= c_1 + T^{p(n)} + \sum_{k=1}^{n} \mathbb{P}(Q=k)(T(k-1) + T(n-k-1))$$

$$= c_1 + T^{p(n)} + \left(\frac{1}{n}\right)\sum_{k=1}^{n}(T(k-1) + T(n-k-1)) = c_1 + T^{p(n)} + \left(\frac{2}{n}\right)\sum_{k=1}^{n} T(k-1)$$

- ... (manipulant les équations et en utilisant la série harmonique)

$$\Longrightarrow T(n) \in \Theta(n\log(n))$$

Le tri rapide : optimisation

- Le choix du pivot :
- En théorie, on pourrait faire en sorte que la complexité de l'algorithme soit $\Theta(n\log(n))$ dans le pire des cas en prenant comme pivot la valeur médiane du sous-tableau. Mais l'algorithme de calcul de la Médiane sur un tableau non trié n'est pas suffisamment efficace dans la pratique, et cette variante est peu utilisée.
- En pratique on utilise un pivot choisi aléatoirement. Cela permet de réduire les chances d'être dans le pire des cas $(\Theta(n^2))$.
- Il existe un algorithme de partitionnement plus complexe, qui permet de réduire le nombre d'échanges effectués lors du partitionnement. De plus, les éléments égaux au pivot sont mieux répartis des deux côtés de celui-ci qu'avec l'algorithme précédent.
- Lorsque les tableaux deviennent très petits (< 50) suite aux appels récursifs, une optimisation classique consiste à utiliser le tri par insertion qui est généralement le plus efficace.

Le tri rapide : synthèse

	Complexité
meilleur des cas	$\Theta(n \log(n))$
cas moyen	$\Theta(n \log(n))$
pire des cas	$\Theta(n^2)$

- Force du tri rapide : Complexité en moyenne en $\Theta(n \log(n))$.
- Le tri rapide est aujourd'hui probablement le tri le plus employé au monde !
- La complexité dans le pire des cas est $\Theta(n^2)$ ... mais on peut réduire les chances de le rencontrer (choix d'un pivot aléatoire).
- Algorithme quasi en place $(\Theta(\log(n)))$ mais pas stable.
- En pratique, souvent un peu plus rapide que le tri fusion

Le tri rapide : code C++ (version 1)


```
inline int partition(int * A, int p, int r)

int pivot = A[r];
int i = p, tmp=0;
for (int j=p; j<=r-1;++j)

int& wj = A[j];
if (wj <= pivot)

int &wi = A[i];
tmp = wi; wi=wj; wj=tmp;
++i;


A[r] = A[i]; A[i] = pivot;
return i;
```

```
void triRapideRec(int * A, int p, int r)

if ( p < r )

int q = partition(A, p, r);
triRapideRec(A, p, q-1);
triRapideRec(A, q+1, r);
```

```
void triRapide(int * A, int n)

triRapideRec(A, 0, n-1);
```

Le tri rapide : code C++ (version 2)

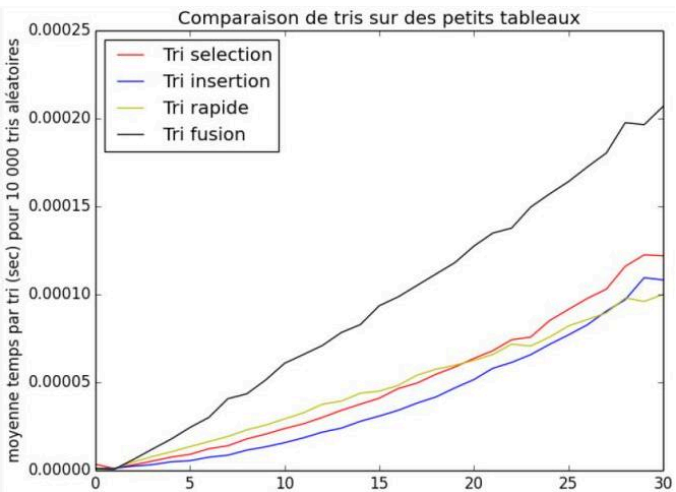

```
1 inline int partition( int * A, int p , int r )
2
3 int pivot = A[r], i = p-1, j=r, tmp=0;
4 while ( i<j )
5
6 do ++i; while( A[i] < pivot);
7 do --j; while( A[j] > pivot);
8 tmp = A[i]; A[i] =A[j]; A[j]=tmp;
9
10 A[j]=A[i]; A[i]=A[r]; A[r]=tmp;
11 return i;
12
```

```
1 void triRapideRec(int * A, int p,int r)
2
3 if ( p < r )
4
5 // tri par insertion pour les petites tailles
6 if ( (r-p+1) < 30 )
7
8 triParInsertion(&(A[p]),r-p+1);
9 return;
10
11 // pivot aleatoire
12 int pid = p+ (rand()%(r-p+1));
13 int tmp = A[r];
14 A[r] = A[pid];A[pid]=tmp;
15
16 int q = partition(A,p,r);
17 triRapideRec(A,p,q-1);
18 triRapideRec(A,q+1,r);
19
20
21
22 void triRapide(int * A, int n)
23
24 triRapideRec(A,0,n-1);
25
```

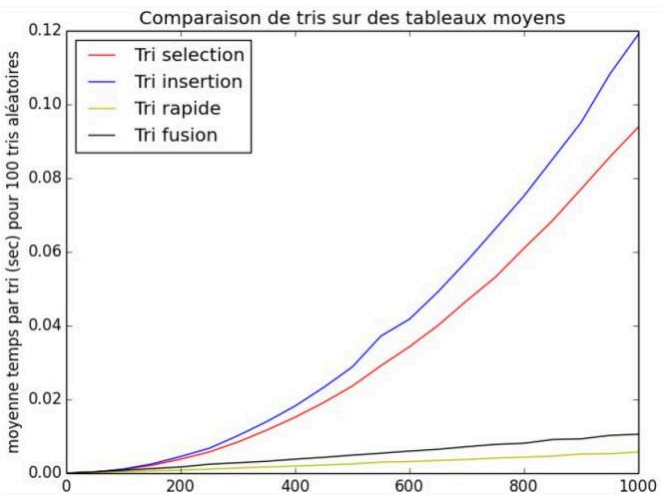
Les algorithmes de tri : synthèse

Nom	meilleur cas	pire cas	en moyenne	en place	stable
Tri par sélection	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	oui	non
Tri à bulles	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	oui	oui
Tri par insertion	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	oui	oui
Tri fusion	$\Theta(n\log(n))$	$\Theta(n\log(n))$	$\Theta(n\log(n))$	non ($\Theta(n)$)	oui
Tri par tas	$\Theta(n\log(n))$	$\Theta(n\log(n))$	$\Theta(n\log(n))$	oui	non
Tri rapide	$\Theta(n\log(n))$	$\Theta(n^2)$	$\Theta(n\log(n))$	non ($\Theta(\log(n))$)	non

Les algorithmes de tri : synthèse



Les algorithmes de tri : synthèse



Les algorithmes de tri : synthèse

Nom	En moyenne	Remarque
Tri par sélection	$\Theta(n^2)$	Inefficaces en pratique
Tri à bulles	$\Theta(n^2)$	
Tri par insertion	$\Theta(n^2)$	Très efficace pour des petites tailles (meilleur que le tri fusion)
Tri fusion	$\Theta(n\log(n))$	Coût asymptotique optimal, efficaces pour des grandes tailles
Tri par tas	$\Theta(n\log(n))$	
Tri rapide	$\Theta(n\log(n))$	Très efficace en pratique (meilleur que le tri par tas). En version randomisée: considéré le meilleur algorithme de tri pour des grandes tailles

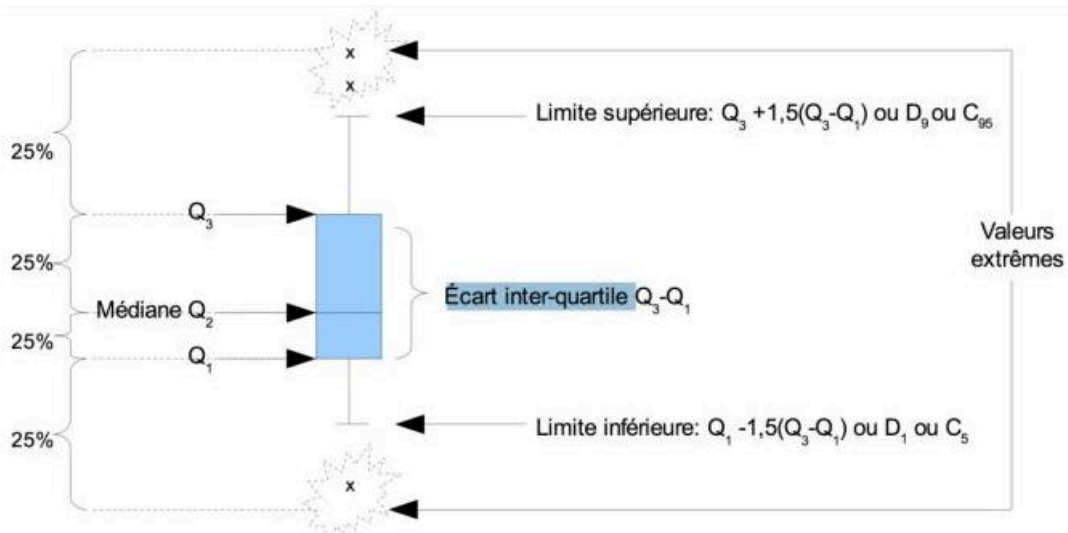
- 1.1 Introduction
- 1.2 Conception, analyse et implémentation en C++
- **1.3 Applications**

Applications : la boîte à moustaches

La boîte à moustaches (traduction de Box & Whiskers Plot) est une invention de John Tukey (1977) pour représenter schématiquement la distribution d'une série statistique.

Valeurs représentées

La boîte à moustaches utilise 5 valeurs qui résument des données : les 3 quartiles Q1, Q2 (médiane), Q3 et les limites inférieures et supérieures L_i et L_s.



Applications : la boîte à moustaches

Médiane

Soit A un tableau de taille n représentant une série statistique. La médiane de cette série est la valeur x qui permet de couper l'ensemble des valeurs en deux parties égales. (Attention, ce n'est pas la moyenne). En d'autres termes, il est possible de trouver M éléments qui sont inférieurs ou égaux à x et les M autres sont supérieurs ou égaux à x. S'il y a un nombre impair de valeurs, la valeur centrale sera mise des deux côtés.

Principe du calcul de la médiane :

1. On commence par trier la série 2. En fonction de la taille de la série :

- Lorsque n est impair, il existe $p \in \mathbb{N}$ tel que $n = 2p + 1$, la médiane de A est l'élément $A[p]$
- Lorsque n est pair, il existe $p \in \mathbb{N}$ tel que $n = 2p$, la médiane de A est alors la moyenne des éléments $A[p-1]$ et $A[p]$ soit: $\left(\frac{A[p-1]+A[p]}{2}\right)$

```
Require: A un tableau de taille n
Ensure: Renvoie les valeurs de la boîte à moustaches
1: procedure BOITEMOUSTACHES(A,n)
2: ALGO TRI(A, n)
3: i_2, j_2 = INDICEMEDIANE(1, n)
4: i_1, j_1 = INDICEMEDIANE(1, i_2)
5: i_3, j_3 = INDICEMEDIANE(j_2, n)
6: Q_2 = (A[i_2] + A[j_2]) / 2
7: Q_1 = (A[i_1] + A[j_1]) / 2
8: Q_3 = (A[i_3] + A[j_3]) / 2
9: L_i = max (A[1],Q_1 - 1.5(Q_3 - Q_1))
10: L_s = min (A[n],Q_3 + 1.5(Q_3 - Q_1))
11: return Q_1, Q_2, Q_3, L_i, L_s
12: end procedure
```

```
Require: Indice de départ p et indice de fin r
Ensure: Renvoie 2 indices de la position de la médiane
1: procedure INDICEMEDIANE (p,r)
2: n = r - p + 1
3: if n modulo 2 = 0 then
4: return p + n / 2, p + n / 2 + 1
5: else
6: return p + (n + 1) / 2, p + (n + 1) / 2
7: end if
8: end procedure
```

Complexité

Mis à part l'appel à ALGO TRI, toutes les opérations effectuées sont en $\Theta(1)$. La complexité de cet algorithme dépend du choix de l'algorithme de tri, donc au mieux on est en $\Theta(n \log(n))$

Applications : recherche par dichotomie

Un autre type d'algorithme que l'on rencontre fréquemment en informatique est la recherche dans une collection (structure de données type tableau ou autres).

Plusieurs problématiques peuvent être associées à ces algorithmes de recherche comme :

- Présence ou non d'un élément
- Nombre d'occurrences d'un élément
- Accéder à des données selon un critère
- ...

Quelques exemples d'utilisations :

- Est-ce que ce mot existe? (est-il dans le dictionnaire?)
- Recherche de la définition d'un mot dans un dictionnaire
- Numéros de téléphone commençant par 0368 dans un annuaire
- Recherche des extremums d'une série statistique
- ...

Applications : recherche par dichotomie

Nous allons nous intéresser uniquement à la problématique du test d'appartenance à la collection. Le principe sera similaire pour les autres problématiques de recherche.

Recherche séquentielle

Une manière simple et naïve est de parcourir tous les éléments jusqu'à ce que l'on trouve l'élément recherché. La complexité est :

- Meilleur des cas : $\Theta(1)$
- Pire des cas : $\Theta(n)$
- En moyenne : $\Theta(n)$

Cependant, si l'on a besoin d'effectuer un grand nombre de recherches dans cette collection, cet algorithme de recherche séquentielle va devenir coûteux.

Pour optimiser ces nombreuses recherches, une idée bien plus efficace consiste à trier d'abord la collection et à utiliser cette propriété dans l'algorithme de recherche. Un exemple typique est la recherche par dichotomie.

```
Require: A un tableau de taille n et e l'élément recherché dans A
1: procedure RECHERCHESEQ(T,n,e)
2: for k = 1 to n do
3: if A[k] = e then
4: return true
5: end if
6: end for
7: return false
8: end procedure
```

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RecDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊ (p + r) / 2⌋
10: if A[q] < e then
11: return RecDicho(A,e,q+1,r)
12: else
13: return RecDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RecDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RecDicho(A,e,q + 1,r)
12: else
13: return RecDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Exemple : recherche de la valeur 6 par dichotomie

1	3	4	6	7	8	10	13	14

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RecDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RecDicho(A,e,q + 1,r)
12: else
13: return RecDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Exemple : recherche de la valeur 6 par dichotomie

1	3	4	6	7	8	10	13	14

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RecDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RecDicho(A,e,q + 1,r)
12: else
13: return RecDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Exemple : recherche de la valeur 6 par dichotomie

1	3	4	6	7	8	10	13	14

Applications : recherche par dichotomie

```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RECDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RECDicho(A,e,q + 1,r)
12: else
13: return RECDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Exemple : recherche de la valeur 6 par dichotomie

1	3	4	6	7	8	10	13	14

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RECDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RECDicho(A,e,q + 1,r)
12: else
13: return RECDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Exemple : recherche de la valeur 6 par dichotomie

1	3	4	6	7	8	10	13	14

1	3	4	6	7	8	10	13	14

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RECDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RECDicho(A,e,q + 1,r)
12: else
13: return RECDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Exemple : recherche de la valeur 6 par dichotomie

1	3	4	6	7	8	10	13	14

1	3	4	6	7	8	10	13	14

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RECDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RECDicho(A,e,q + 1,r)
12: else
13: return RECDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Exemple : recherche de la valeur 6 par dichotomie

1	3	4	6	7	8	10	13	14

1	3	4	6	7	8	10	13	14

1	3	4	6	7	8	10	13	14

Applications : recherche par dichotomie

```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RECDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RECDicho(A,e,q + 1,r)
12: else
13: return RECDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Terminaison

- Soit $(u_k)_{k \geq 0}$ la suite définie par $u_0 = n$ et u_k le nombre maximal d'éléments à trier entre les 2 appels récursifs à RECDicho à la récursion k.
- Tant que $u_k > 1$,

$$u_{k+1} = \left\lfloor \left(\frac{u_k + 1}{2} \right) \right\rfloor \leq \left(\frac{u_k + 1}{2} \right) < u_k$$

- Par récurrence, on en déduit que $(u_k)_{k \geq 0}$ est strictement décroissante et positive.

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RECDicho(A,e,p,r)
2: if p = r then
3: if A[p] = e then
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋
10: if A[q] < e then
11: return RECDicho(A,e,q + 1,r)
12: else
13: return RECDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Correction partielle : preuve par induction

- Si $n = 1$, on renvoie la bonne réponse.
- Si $n \geq 2$, on partitionne par rapport à q
- Si $A[q] < e$, on appelle RechDicho dans $A[q+1 : r]$
- Sinon on appelle RechDicho dans $A[p : q]$
- Par hypothèse inductive, on obtient la bonne réponse

Applications : recherche par dichotomie


```
Require: A un tableau trié de taille n, e l'élément recherché dans A et deux
indices p et r
1: procedure RECDicho(A,e,p,r)
2: if p = r then ▷ c1
3: if A[p] = e then ▷ c2
4: return true
5: else
6: return false
7: end if
8: end if
9: q = ⌊(p + r)/2⌋ ▷ c3
10: if A[q] < e then ▷ c4
11: return RECDicho(A,e,q + 1,r)
12: else
13: return RECDicho(A,e,p,q)
14: end if
15: end procedure
```

Principe

En fonction de la valeur de A[q], la recherche s'effectue dans la moitié du tableau où doit se trouver la valeur :

$$n_1 = \left\lfloor \left(\frac{n+1}{2} \right) \right\rfloor \quad \text{et} \quad n_2 = n - n_1 \quad \Longrightarrow \quad q = \left\lfloor \left(\frac{p+r}{2} \right) \right\rfloor$$

Complexité

C'est un algorithme de type « diviser pour régner ». Pour tous les cas, on a :

$$T(n) = T\left(\frac{n}{2}\right) + c_1 + c_3 + c_4$$

$$T(1) = c_1 + c_2$$

- On peut appliquer le Master théorème avec $a = 1, b = 2, d = 0$.
- On en déduit que la complexité de la recherche par dichotomie : $T(n) \in \Theta(\log(n))$

Applications : recherche par dichotomie

- La recherche par dichotomie est un algorithme rapide $\Theta(\log(n))$
- Mais uniquement valable pour des collections triées
- Dans le cas d'un grand nombre de recherches dans une collection non triée, la stratégie à appliquer pour réduire le temps d'exécution :

1. On trie la collection $(\Theta(n \log(n)))$ 2. Recherches par dichotomie $(\Theta(\log(n)))$

- C'est un algorithme récursif terminal : il peut être converti en une boucle itérative (le compilateur C++ pourra le faire pour vous)