Chapitre 4 – preuves et tris

Plan du chapitre

- Preuves d’algorithmes
- Quelques tris (avec les preuves)

4.1 – Preuves d’algorithmes

Qu’est-ce que c’est ?

- Preuve d’un algorithme = preuve mathématique qu’un algorithme répond au problème donné en un temps fini
  - Preuve de terminaison = L’algorithme termine toujours en temps fini
  - Preuve de correction partielle = Quand l’algorithme termine, il produit la valeur de sortie attendue, quelque soit la valeur d’entrée
- Pourquoi ?
  - Pour être sûr que le programme fonctionne (et pas juste le croire)

Preuve de terminaison

- Remarques préliminaires
  - Une boucle pour est exécutée un nombre fini de fois, et donc termine
  - Une boucle tantque peut ne pas terminer si la condition de terminaison n'est jamais vraie
  - Une fonction récursive peut ne pas terminer si les cas de base ne sont jamais atteints
- Pour chaque boucle tantque et pour chaque fonction récursive, on cherche à montrer que la suite des valeurs à chaque tour/appel termine, par exemple :
  - En définissant un variant (lié à la condition de terminaison) qui est initialisé à une valeur positive ou nulle, qui décroît strictement à chaque tour/appel, et pour lequel la boucle/fonction s'arrête quand on atteint 0

Exemple 1 de preuve de terminaison

- Algorithme Exemple1 :

```
s ← 0
pour i de 1 à n faire
        pour j de 1 à i faire
                pour k de 1 à n-j faire
                        s ← s+k
                finpour
        finpour
finpour
retourner s
```

- L'algorithme Exemple1 n'est constitué que de boucles pour, donc il termine
  - En effet, chaque boucle pour s'exécutera au plus n fois

Exemple 2 de preuve de terminaison

- Algorithme Exemple2
  entrée : T tableau trié de n entiers, x une valeur recherchée
  début ← 0
  fin ← n-1
  tantque début+1<fin faire
  milieu ← (début+fin)/2
  si T[milieu]=x alors
  retourner VRAI
  sinon si T[milieu]<x alors
  début ← milieu+1
  sinon
  fin ← milieu-1
  finsi
  fintantque
  retourner FAUX

- Variant : longueur = fin-début+1, qui représente le nombre de cases dans lequel on cherche x
- Initialement, longueur = fin-début+1 = n-1+1 = n
- A chaque tour de boucle :
  - Soit T[milieu]=x et la boucle s'arrête avec un « retourner VRAI »
  - Soit T[milieu]<x : nouvelle longueur = fin-milieu+1 < ancienne longueur
  - Soit T[milieu]>x : nouvelle longueur < ancienne longueur
- Longueur décroît strictement et la boucle s'arrête quand longueur devient inférieure ou égale à 1

Preuve de correction partielle

- Invariant = propriété qui est vraie avant une boucle, et à la fin de chaque tour de boucle (donc aussi après la fin de la boucle)
- Les invariants sont généralement utilisés pour faire des preuves de correction partielle

Exemple de correction partielle (1/3)

- Algorithme Exemple2
  entrée : T tableau trié de n entiers, x une valeur recherchée
  début ← 0
  fin ← n-1
  tantque début+1<fin faire
  milieu ← (début+fin)/2
  si T[milieu]=x alors
  retourner VRAI
  sinon si T[milieu]<x alors
  début ← milieu+1
  sinon
  fin ← milieu-1
  finsi
  fintantque
  retourner FAUX

- L'invariant est : si elle existe, la valeur x a un index dans [début;fin]
- Avant la boucle :
  - Début = 0 et fin = n-1
  - L'invariant {si elle existe, la valeur x a un index dans [0;n-1]} est vrai
- En début de boucle :
  - L'invariant est vrai
- Après le premier « si » :
  - On a trouvé la valeur « x », donc on peut retourner VRAI

Exemple de correction partielle (2/3)

- Algorithme Exemple2
  entrée : T tableau trié de n entiers, x une valeur recherchée
  début ← 0
  fin ← n-1
  tantque début+1<fin faire
  milieu ← (début+fin)/2
  si T[milieu]=x alors
  retourner VRAI
  sinon si T[milieu]<x alors
  début ← milieu+1
  sinon
  fin ← milieu-1
  finsi
  fintantque
  retourner FAUX

- Après le deuxième « si » :
  - On sait que T[milieu]<x
  - Donc x ne peut pas se trouver dans [début;milieu], car T est trié
  - De plus, on avait l'invariant qui indiquait : {si elle existe, la valeur x a un index dans [début;fin]} => on en déduit que {si elle existe, la valeur x a un index dans [milieu+1;fin]}
  - Après la mise à jour de début, l'invariant devient {si elle existe, la valeur x a un index dans [début;fin]}
- Après le troisième « si » :
  - L'invariant reste vrai après la mise à jour de « fin »

Exemple de correction partielle (3/3)

- Algorithme Exemple2
  entrée : T tableau trié de n entiers, x une valeur recherchée
  début ← 0
  fin ← n-1
  tantque début+1<fin faire
  milieu ← (début+fin)/2
  si T[milieu]=x alors
  retourner VRAI
  sinon si T[milieu]<x alors
  début ← milieu+1
  sinon
  fin ← milieu-1
  finsi
  fintantque
  retourner FAUX

- En fin de boucle :
  - L'invariant {si elle existe, la valeur x a un index dans [début;fin]} reste vrai
  - De plus, on sait que début+1<fin est faux, donc fin>=début+1, donc fin>début
  - L'invariant devient {si elle existe, la valeur x a un index dans []}
  - Cela signifie que x ne peut pas être dans le tableau, on peut donc retourner FAUX

4.2 – Quelques tris

Tri par sélection du maximum

- Principe :
  - On cherche la plus grande valeur, puis on la place à la fin du tableau
  - On recommence sur un tableau plus petit (d'une case)
- Dans la fonction ci-contre :
  - « i » représente le numéro de l'étape (de 1 à n-1, car la dernière étape sur un tableau d'une case est inutile)
  - « j » représente la recherche du maximum dans un tableau de plus en plus petit, sans la case d'indice 0

- Fonction en C :

```
void maxSelectSort(int * T, int n) {
        int i, j, maxIndex;
        for (i=1; i<=n-1; i++) {
                maxIndex = 0;
                for (j=1; j<n-i+1; j++) {
                        if (T[j]>T[maxIndex]) {
                                maxIndex = j;
                        }
                }
                swap(T, maxIndex, n-i);
        }
}
```

Preuve de terminaison du tri par sélection du maximum

- Il n'y a que des boucles pour imbriquées => l'algorithme termine (en moins de n^2 tours)

Preuve de correction partielle du tri par sélection du maximum (1/3)

- Fonction en C :

```
void maxSelectSort(int * T, int n) {
        int i, j, maxIndex;
        for (i=1; i<=n-1; i++) {
                maxIndex = 0;
                for (j=1; j<n-i+1; j++) {
                  if (T[j]>T[maxIndex]) {
                          maxIndex = j;
                  }
                }
                swap(T, maxIndex, n-i);
        }
}
```

- Invariant :
  les valeurs dans]n-i;n-1] correspondent aux plus grandes valeurs de T, triées
- Avant la boucle :
  - i vaut 1, et on a bien les plus petites valeurs de T dans]n-1;n-1] = ensemble vide
- En début de boucle :
  - L'invariant est vrai

Preuve de correction partielle du tri par sélection du maximum (2/3)

- Fonction en C :

```
void maxSelectSort(int * T, int n) {
        int i, j, maxIndex;
        for (i=1; i<=n-1; i++) {
                maxIndex = 0;
                for (j=1; j<n-i+1; j++) {
                  if (T[j]>T[maxIndex]) {
                          maxIndex = j;
                  }
                }
                swap(T, maxIndex, n-i);
        }
}
```

- Dans la boucle :
  - maxIndex est l'indice de la plus grande valeur dans [0;j-1]
  - Avant la deuxième boucle for : l'invariant est vrai
  - Dans la deuxième boucle for : si T[j]>T[maxIndex], alors l'indice de la plus grande valeur dans [0;j] est j
  - Sinon (T[j]<=T[maxIndex]), alors l'indice de la plus grande valeur dans [0;j] est maxIndex
  - En fin de boucle : maxIndex est l'indice de la plus grande Valeur dans [0;j]

Preuve de correction partielle du tri par sélection du maximum (3/3)

- Fonction en C :

```
void maxSelectSort(int * T, int n) {
        int i, j, maxIndex;
        for (i=1; i<=n-1; i++) {
                maxIndex = 0;
                for (j=1; j<n-i+1; j++) {
                  if (T[j]>T[maxIndex]) {
                          maxIndex = j;
                  }
                }
                swap(T, maxIndex, n-i);
        }
}
```

- Après l'appel à « swap » :
  - Les valeurs dans [n-i;n-1] correspondent aux plus grandes valeurs de T, triées
- A la fin de la boucle :
  - i vaut n
  - Les valeurs dans]0;n-1] correspondent aux plus grandes valeurs de T, triées
  - Les valeurs dans [0;n-1] correspondent aux plus grandes valeurs de T, triées

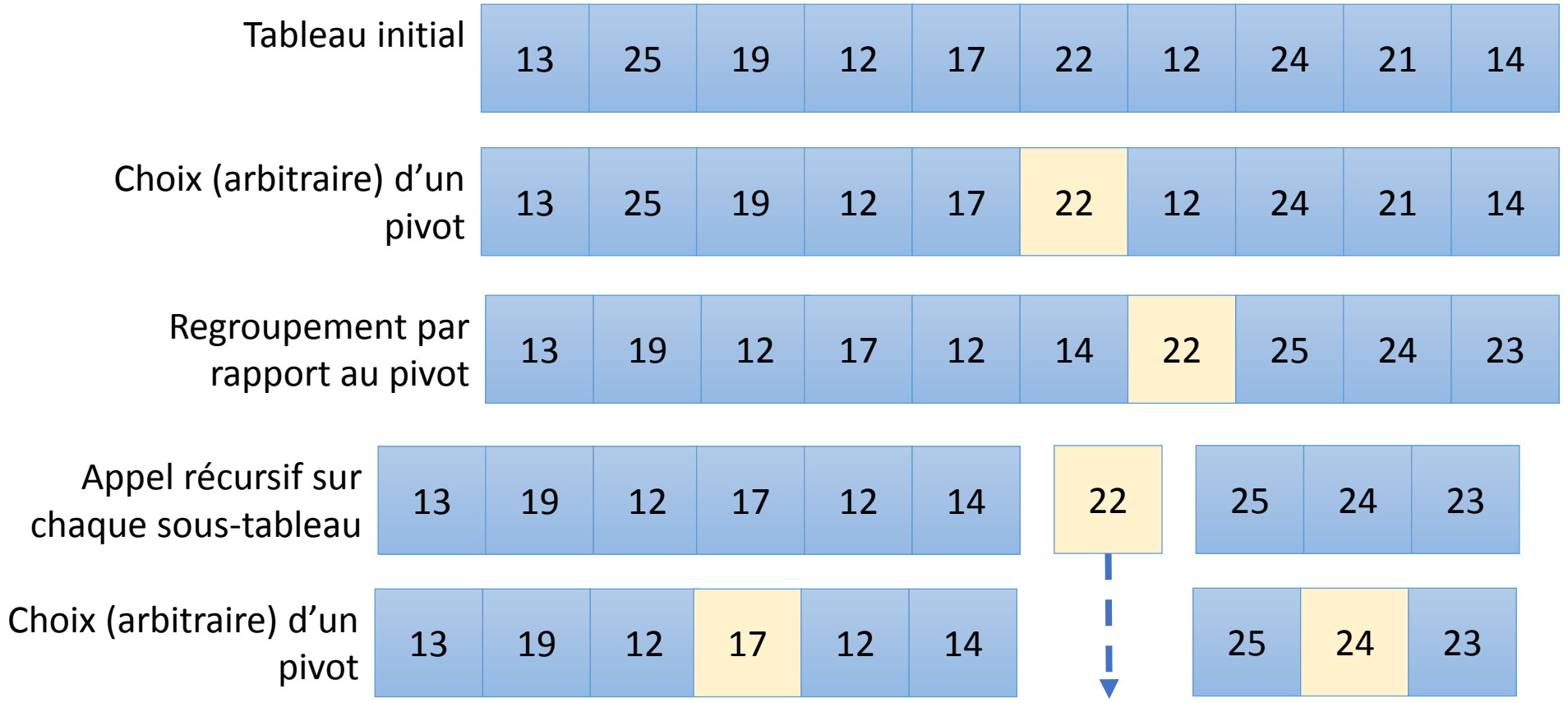
Tri par sélection du maximum

- Complexité : $O(n^2)$ dans le pire des cas
  - Il s'agit d'un algorithme très simple à implémenter, mais lent en pratique

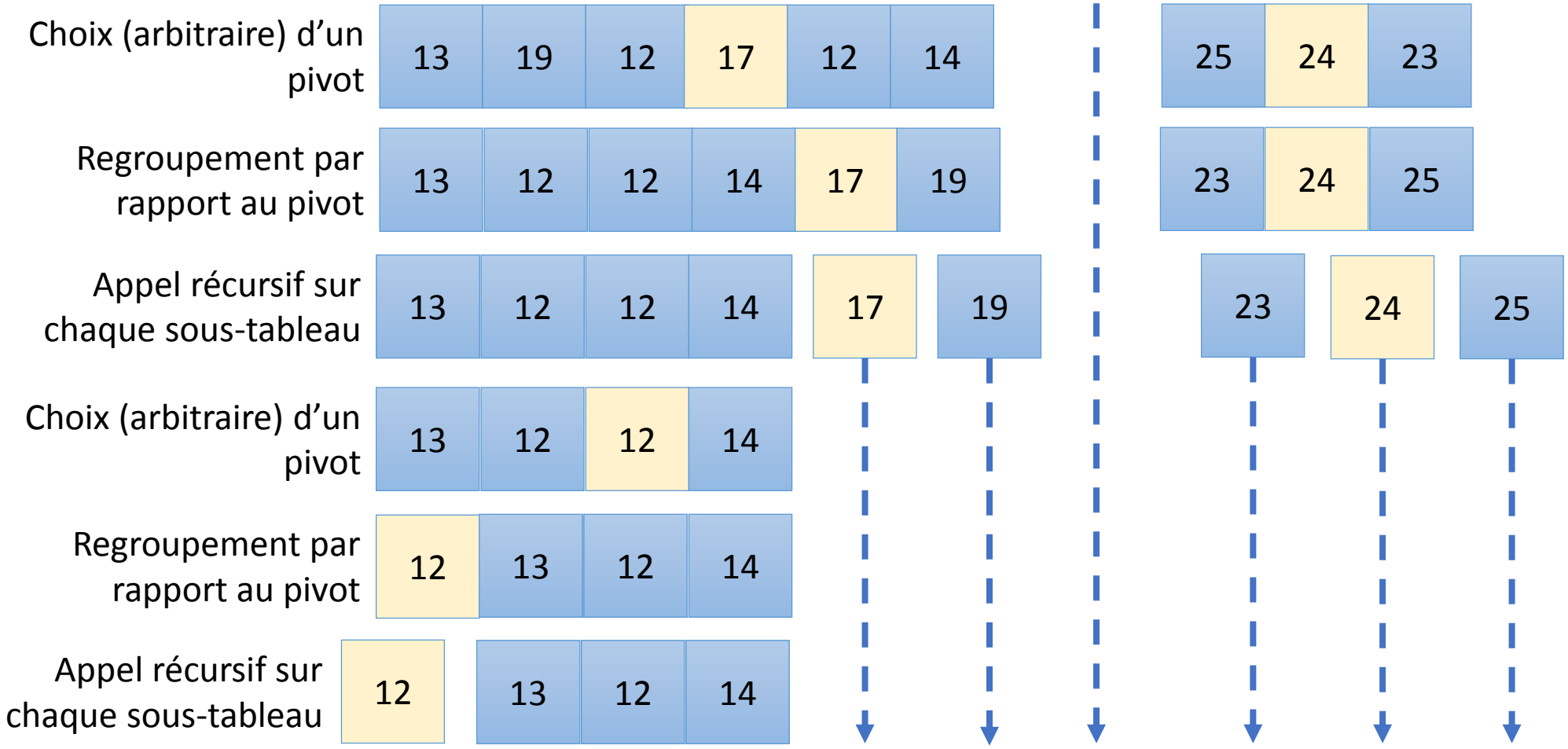
Tri rapide

- Le tri rapide a été inventé par C. A. R. Hoare en 1961
- Principe :
  - On choisit une valeur quelconque du tableau que l'on nomme pivot
  - On met toutes les valeurs inférieures au pivot dans un sous-tableau gauche, et toutes les valeurs supérieures ou égales au pivot dans un sous-tableau droit
  - On recommence récursivement sur les sous-tableaux gauches et droits, en s'arrêtant quand le tableau est petit (par exemple, de taille inférieure ou égale à 1)

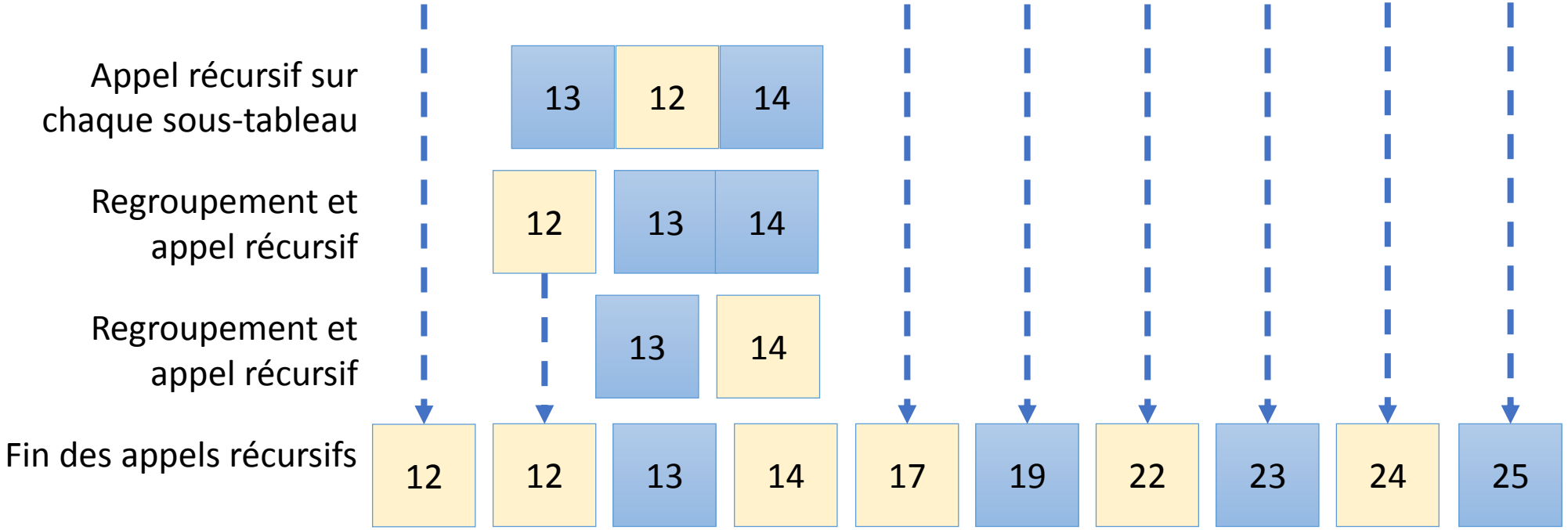
Exemple de tri rapide (1/3)



Exemple de tri rapide (2/3)



Exemple de tri rapide (3/3)



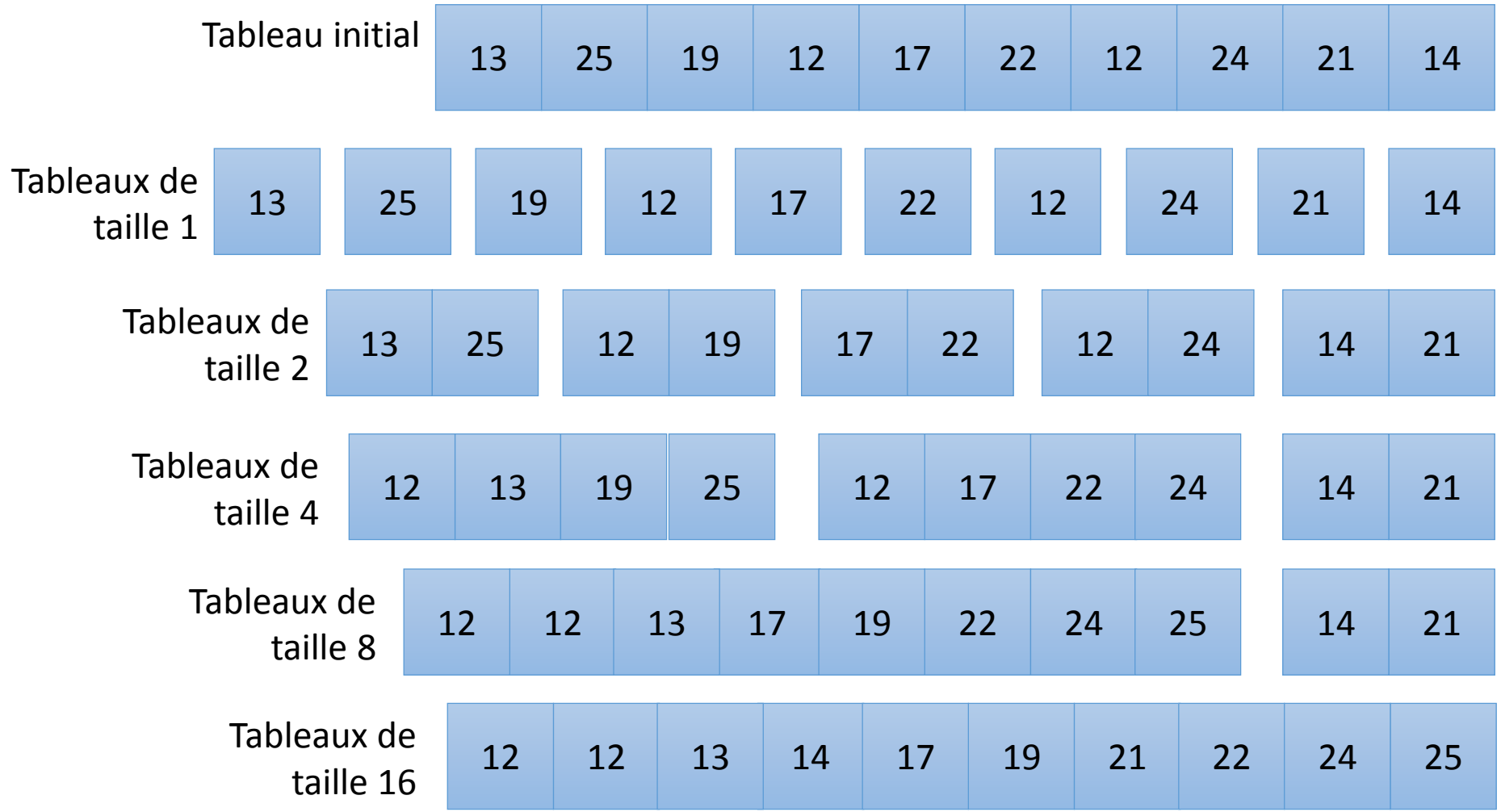
Complexité

- La complexité dans le pire des cas est $O(n^2)$
- La complexité en moyenne est O(n.log(n))
- En pratique, il s'agit d'un tri rapide

Tri fusion

- Le tri fusion a été inventé par J. von Neumann en 1945
- Principe :
  - On divise le tableau non-trié de n entiers en n tableaux (triés) de 1 entiers
  - On considère les tableaux triés deux à deux, et on les fusionne pour produire des tableaux triés deux fois plus longs
  - On procède récursivement jusqu'à obtenir un unique tableau trié

Exemple de tri fusion



Complexité

- La complexité dans le pire des cas est O(n.log(n))
- Il s'agit donc d'un algorithme rapide (y compris dans le pire des cas)
- En pratique, il est souvent légèrement plus lent que le tri rapide

Conclusions

- Un programme peut être prouvé formellement
- Il existe de nombreux algorithmes de tris
  - Des versions efficaces sont souvent implémentées dans les bibliothèques standards