Université Assane SECK de Ziguinchor

Unité de Formation et de Recherche des Sciences et Technologies

Département d'Informatique

CONCEPTION DES ALGORITHMES

Licence 1 Math - Physique – Informatique

Janvier 2021

ydieng@univ-zig.sn

Il existe de nombreuses façons de concevoir un algorithme. Le tri par insertion utilise une approche incrémentale qui après avoir trié le sous-tableau A[1 . . j − 1], on insère l'élément A[j] au bon emplacement pour produire le sous-tableau trié A[1 . . j].

Cette section va présenter une autre approche de conception appelée « diviser pour régner ». Grace à cette technique nous proposons un algorithme de tri avec un temps d'exécution du cas le plus défavorable très inférieur à celui du tri par insertion. L'un des avantages des algorithmes diviser-pour-régner est que leurs temps d'exécution sont souvent faciles à déterminer, via des techniques que nous présenterons.

1 - MÉTHODE DIVISER-POUR-RÉGNER

Nombre d'algorithmes utiles sont d'essence récursive : ils s'appellent eux-mêmes, de manière récursive, une ou plusieurs fois pour traiter des sous-problèmes très similaires. Ces algorithmes suivent généralement une approche diviser pour régner :

- d'abord ils séparent le problème en plusieurs sous-problèmes semblables au problème initial mais de taille moindre
- ensuite ils résolvent les sous-problèmes de façon récursive,
- enfin ils combinent toutes les solutions pour produire la solution du problème original.

1.1 - Le 3 étapes du paradigme diviser-pour-régner

1) **Diviser** le problème en un certain nombre de sous-problèmes.
2) **Régner** sur les sous-problèmes en le résolvant de manière récursive. Si la taille d'un sous-problème est suffisamment réduite, on peut le résoudre directement.
3) **Combiner** les solutions des sous-problèmes pour produire la solution du problème originel.

1.2 - Exemple du tri par fusion

Cet algorithme suit fidèlement la méthodologie diviser-pour-régner.

1) **Diviser** la suite de n éléments à trier en deux sous-suites de n/2 éléments chacune.
2) **Régner** en Triant les deux sous-suites de manière récursive en utilisant le tri par fusion.
3) **Combiner/Fusionner** les deux sous-suites triées pour produire la réponse triée.

La récursivité s'arrête quand la séquence à trier a une longueur 1: en effet il n'y a plus rien à faire puisqu'une suite de longueur 1 est déjà triée. La clé de voûte de l'algorithme est la fusion de deux séquences triées dans l'étape « combiner ». Cette fusion est faite grâce à une

procédure auxiliaire **FUSION(*A, p, q, r*) .** *A* étant un tableau et *p*, *q* et *r* étant des indices numérotant des éléments du tableau tels que $p \leq q < r$. La procédure suppose que les sous-tableaux $A[p\,..\,q]$ et $A[q+1\,..\,r]$ sont triés. Elle les ***fusionne*** pour en faire un même sous-tableau trié, qui remplace le sous-tableau courant $A[p\,..\,r]$. Notre procédure **FUSION** a une durée $\Theta(n)$, où $n = r - p + 1$ est le nombre d'éléments fusionnés.

Principe de fonctionnement du fusion : Supposons que l'on ait deux piles **A** et **B** de cartes posées sur la table. Supposons que chaque pile est triée de façon à ce que la carte la plus faible soit en haut. On veut fusionner les deux piles pour obtenir une pile unique triée **R**, dans laquelle les cartes seront à l'envers. Le principe de fusion est :

1. Choisir la plus faible des deux cartes occupant les sommets respectifs des deux piles **A** et **B** :
   - Retirer la de sa pile (ce qui a va exposer une nouvelle carte), puis placer la à l'envers sur la pile résultat **R**.
2. Répéter cette étape jusqu'à épuisement de l'une des piles **A** et **B**.

Après quoi il suffit de prendre la pile qui reste (**A** ou **B**) et de la placer à l'envers sur la pile résultat **R**.

Complexité de fusion : Au niveau du calcul, chaque étape fondamentale prend un temps constant, vu que l'on se contente de comparer les deux cartes du haut. Comme on effectue au plus ***n*** étapes fondamentales, la fusion prend une durée **Θ(n)**.

1.3 - L'algorithme du tri par fusion

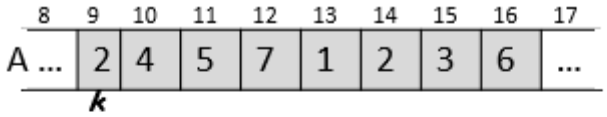
Le principe est de placer en bas de chaque pile une carte ***sentinelle*** contenant une valeur spéciale. Cette valeur nous permettra de ne pas vérifier à chaque fois si l'une des piles est vide. (Nous utiliserons ∞ comme valeur sentinelle.) Ainsi, chaque fois qu'il y a apparition d'une carte portant la valeur ∞, elle ne peut pas être la carte la plus faible sauf si les deux piles exposent en même temps leurs cartes sentinelle. Vu qu'on a exactement $r-p+1$ cartes sur la pile de sortie, nous pourrons arrêter lorsque ce nombre d'étapes sera effectué.

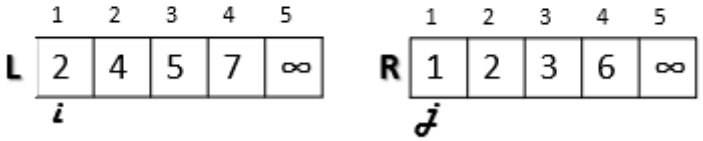
```
FUSION(A, p, q, r)
1  n1 ← q − p + 1
2  n2 ← r − q
3  créer tableaux L[1 . . n1 + 1] et R[1 . . n2 + 1]
4  pour i ← 1 à n1
5      faire L[i] ← A[p + i − 1]
6  pour j ← 1 à n2
7      faire R[j] ← A[q + j]
8  L[n1 + 1] ← ∞
9  R[n2 + 1] ← ∞
10 i ← 1
11 j ← 1
12 pour k ← p à r
13     faire si L[i] ⩽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

- La ligne 1 calcule la longueur $n1$ du sous-tableau $A[p..q]$ et la ligne 2 calcule la longueur $n2$ du sous-tableau $A[q + 1..r]$.
- On crée des tableaux L et R de longueurs $n1 + 1$ et $n2 + 1$, respectivement, en ligne 3.
- La boucle **pour** des lignes 4–5 copie le sous-tableau $A[p . . q]$ dans $L[1 . . n1]$ et la boucle **pour** des lignes 6–7 copie le sous-tableau $A[q + 1. . r]$ dans $R[1 . . n2]$.
- Les lignes 8–9 placent les sentinelles aux extrémités des tableaux L et R.
- Les lignes 10–17, effectuent les $r – p + 1$ étapes fondamentales en conservant l'invariant de boucle que voici :
  - Au début de chaque itération de la boucle **pour** des lignes 12–17, le sous-tableau $A[p . . k – 1]$ contient les $k – p$ plus petits éléments de $L[1 . . n1 + 1]$ et $R[1 . . n2 + 1]$, en ordre trié.

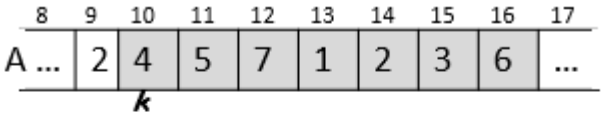
i. ***Exemple*** :

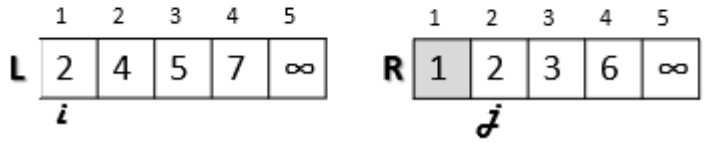
1) Quel est l'élément qui doit être en position k = 9 ? C'est le plus petit entre L[i=1] et R[j=1].



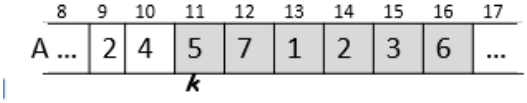


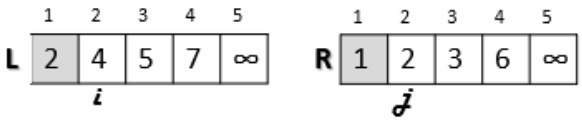
2) Quel est l'élément qui doit être en position k = 10 ? C'est le plus petit entre L[i=1] et R[j=2].



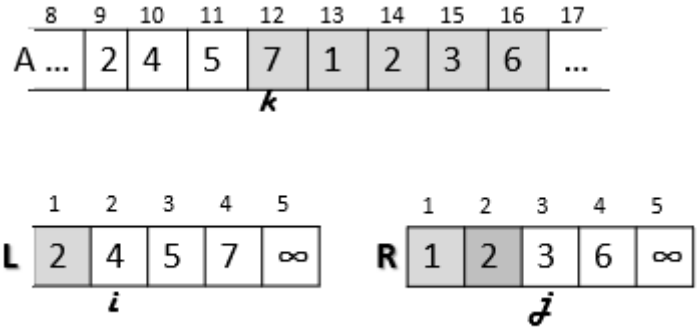


3) Quel est l'élément qui doit être en position k = 11 ? C'est le plus petit entre L[i=2] et R[j=2].

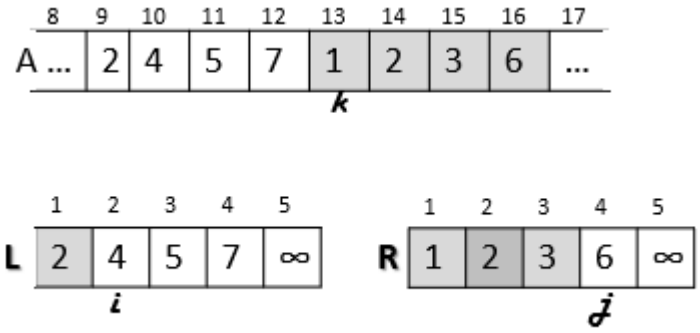




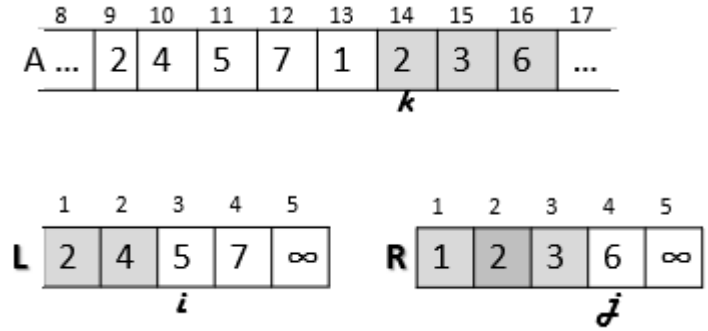
4) Quel est l'élément qui doit être en position k = 12 ? C'est le plus petit entre L[i=2] et R[j=3].



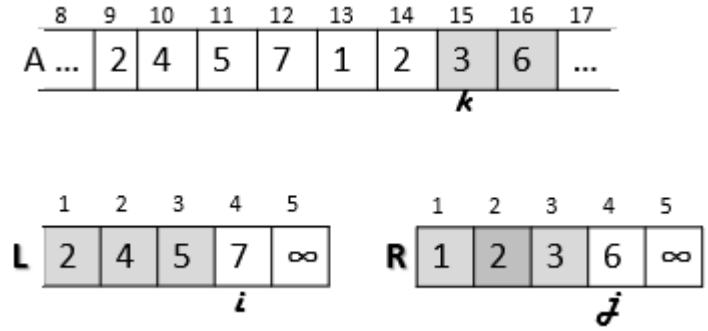
5) Quel est l'élément qui doit être en position k = 13 ? C'est le plus petit entre L[i=2] et R[j=4].



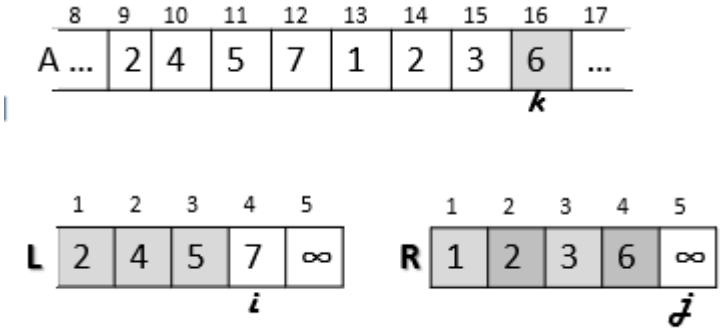
6) Quel est l'élément qui doit être en position k = 13 ? C'est le plus petit entre L[i=2] et R[j=4].



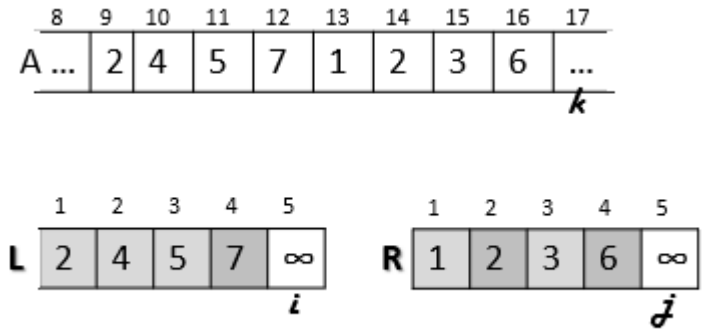
7) Quel est l'élément qui doit être en position k = 15 ? C'est le plus petit entre L[i=4] et R[j=4].



8) Quel est l'élément qui doit être en position k = 16 ? C'est le plus petit entre L[i=4] et R[j=5].



9) Quel est l'élément qui doit être en position k = 17 ? C'est le plus petit entre L[i=5] et R[j=5].



1.4 - Correction de l'algorithme du tri par fusion

Initialisation: Nous allons montrer que l'invariant est vrai avant la première itération de la boucle pour des lignes 12–17. En effet, avant la première itération de la boucle, on a $k = p$, de sorte que le sous-tableau $A[p \,..\, k-1]$ est vide. Ce sous-tableau vide contient les $k - p = 0$ plus petits éléments de L et R ; et, comme $i = j = 1$, $L[i]$ et $R[j]$ sont les plus petits éléments de leurs tableaux à ne pas avoir été copiés dans A.

Conservation: Nous allons montrer que chaque itération de la boucle conserve l'invariant. En effet, supposons que $L[i] \leq R[j]$. Alors L[i] est le plus petit élément qui n'a pas encore été copié dans A. Comme A[p . . k−1] contient les k−p plus petits éléments, alors après que la ligne 14 a copié L[i] dans A[k], le sous-tableau A[p . . k] contient les k−p+1 plus petits éléments. Incrémenter k (dans l'actualisation de la boucle **pour**) et i (en ligne 15) recrée l'invariant pour l'itération suivante. Si l'on a L[i] > R[j], alors les lignes 16–17 font l'action idoine pour conserver l'invariant.

Terminaison: Nous allons montrer que l'invariant fournit une propriété utile pour prouver la conformité de la procédure quand la boucle se termine. À la fin de la boucle, on a $k = r + 1$. D'après l'invariant, le sous-tableau $A[p \,..\, k-1]$, qui est $A[p \,..\, r]$, contient les $k - p = r - p + 1$ plus petits éléments de $L[1 \,..\, n1 + 1]$ et $R[1 \,..\, n2 + 1]$, dans l'ordre trié. Les tableaux

L et *R*, à eux deux, contiennent $n1 + n2 + 2 = r - p + 3$ éléments. Tous les éléments, sauf les deux plus forts, ont été copiés dans *A*, et ces deux plus gros éléments ne sont autres que les sentinelles.

1.5 - Complexité en temps de FUSION

Pour voir que la procédure FUSION s'exécute en $\Theta(n)$ temps, avec $n = r - p + 1$, observez que chacune des lignes 1–3 et 8–11 prend un temps constant et que les boucles **pour** des lignes 4–7 prennent $\Theta(n1 + n2) = \Theta(n)$ temps, et qu'il y a n itérations de la boucle **pour** des lignes 12–17, chacune d'elles prenant un temps constant.

1.6 - Utilisation dans TRI-FUSION

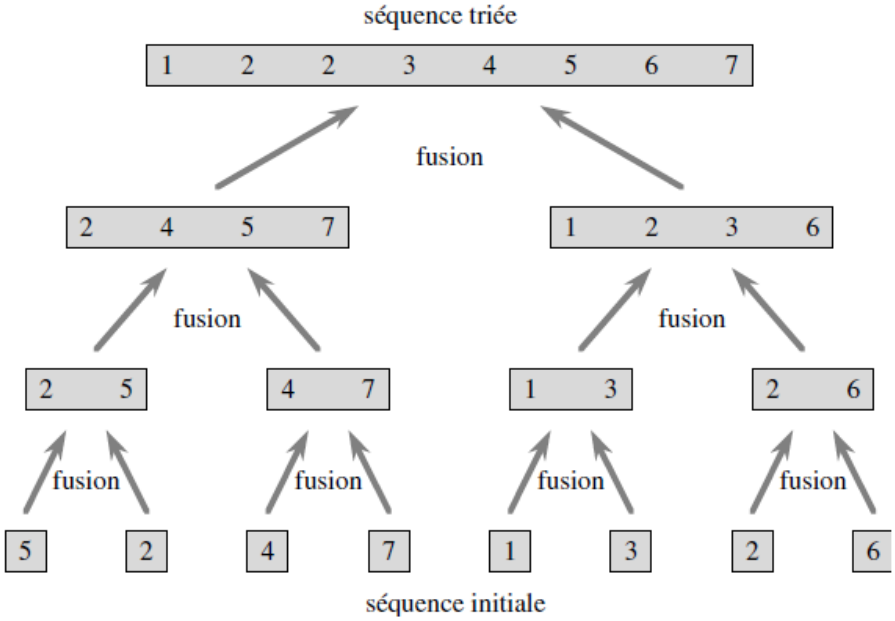
La procédure **FUSION** peut maintenant être employée comme sous-routine de l'algorithme du tri par fusion. La procédure **TRI-FUSION(*A, p, r*)** trie les éléments du sous-tableau $A[p \,..\, r]$. Si $p \geq r$, le sous-tableau a au plus un seul élément et il est donc déjà trié. Sinon, l'étape diviser se contente de calculer un indice q qui partitionne $A[p \,..\, r]$ en deux sous-tableaux tels que $A[p \,..\, q]$, contenant $n/2$ éléments, et $A[q + 1 \,..\, r]$, contenant $n/2$ éléments.

1.7 - L'algorithme TRI-FUSION

```
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)
```

Fonctionnement du TRI-FUSION : Pour trier toute la séquence A = A[1], A[2], . . . , A[n], on fait l'appel initial TRI-FUSION(A, 1, longueur[A]) (ici aussi, longueur[A] = n). L'algorithme consiste à fusionner des paires de séquences à 1 élément pour former des séquences triées de longueur 2 ensuite à fusionner des paires de séquences de longueur 2 pour former des séquences triées de longueur 4, etc.

Ce processus continue jusqu'à ce qu'il y ait fusion de deux séquences de longueur n/2 pour former la séquence triée définitive de longueur n.



1.8 - Analyse des algorithmes diviser-pour-régner

Si on a un algorithme récursif et un problème de taille n alors le temps d'exécution peut souvent être décrit par une équation de récurrence, ou récurrence, à partir du temps d'exécution pour des entrées de taille moindre. On peut alors se servir d'outils mathématiques pour résoudre la récurrence et trouver des bornes pour les performances de l'algorithme. Une récurrence pour le temps d'exécution d'un algorithme diviser-pour-régner s'appuie sur les trois étapes du paradigme de base. En effet, soit $T(n)$ le temps d'exécution d'un problème de taille n.

1) Si la taille du problème est suffisamment petite, disons $n \leq c$ pour une certaine constante c, la solution directe prend un temps constant que l'on écrit $\Theta(1)$.
2) Supposons que l'on divise le problème en a sous-problèmes, la taille de chacun étant $1/b$ de la taille du problème initial. Pour le tri par fusion, tant a que b valent 2, mais nous verrons beaucoup d'algorithmes diviser-pour-régner dans lesquels $a \neq b$.
3) Si l'on prend un temps $D(n)$ pour diviser le problème en sous-problèmes et un temps $C(n)$ pour construire la solution finale à partir des solutions aux sous-problèmes, on obtient la récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leqslant c\,, \\ aT(n/b) + D(n) + C(n) & \text{sinon}\,. \end{cases}$$

Analyse du tri par fusion : Même si le pseudo-code de TRI-FUSION s'exécute correctement quand le nombre d'éléments n'est pas pair, notre analyse fondée sur la récurrence

sera simplifiée si nous supposons que la taille du problème initial est une puissance de deux. Chaque étape diviser génère alors deux sous-séquences de taille n/2 exactement. Au chapitre 4, nous verrons que cette supposition n'affecte pas l'ordre de grandeur de la solution de la récurrence.

Nous raisonnerons comme suit pour mettre en œuvre la récurrence pour $T(n)$, (temps d'exécution du cas le plus défavorable du tri par fusion de n nombres). Le tri par fusion d'un seul élément prend un temps constant ; avec $n > 1$ éléments, on segmente le temps d'exécution de la manière suivante.

- **Diviser :** L'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant. Donc $D(n) = \Theta(1)$.
- **Régner :** On résout récursivement deux sous-problèmes, chacun ayant la taille $n/2$, ce qui contribue pour $2T(n/2)$ au temps d'exécution.
- **Combiner :** Nous avons déjà noté que la procédure FUSION sur un sous-tableau à n éléments prenait un temps $\Theta(n)$, de sorte que $C(n) = \Theta(n)$.

Quand on ajoute les fonctions $D(n)$ et $C(n)$ pour l'analyse du tri par fusion, une fonction qui est $\Theta(n)$ et une fonction qui est $\Theta(1)$, cette somme est une fonction linéaire de n, à savoir $\Theta(n)$. L'ajouter au terme $2T(n/2)$ de l'étape régner donne la récurrence pour $T(n)$, temps d'exécution du tri par fusion dans le cas le plus défavorable :

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

Au chapitre 4, nous verrons le « théorème général » grâce auquel on peut montrer que $T(n)$ est $\Theta(n \lg n)$, où $\lg n$ désigne $\log_2 n$. Comme la fonction logarithme croit plus lentement que n'importe quelle fonction linéaire, pour des entrées suffisamment grandes, alors le tri par fusion, avec son temps d'exécution $\Theta(n \lg n)$, est plus efficace que le tri par insertion dont le temps d'exécution vaut $\Theta(n^2)$ dans le cas le plus défavorable.

En attendant de voir le théorème général nous pouvons comprendre intuitivement pourquoi la solution de la récurrence du slide précédente est $T(n) = \Theta(n \lg n)$.

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

La constante c représentant le temps requis pour résoudre des problèmes de taille 1, ainsi que le temps par élément de tableau des étapes diviser et combiner. Pour des raisons de commodité, on suppose que n est une puissance exacte de 2.