# Chapitre 4

# LA STRATÉGIE DIVISER POUR RÉGNER (DIVIDE&CONQUER)

## 1. Méthode Diviser pour Régner

**Définition 1.**
Diviser pour régner est une méthode de programmation récursive qui consiste à diviser un problème en sous-problèmes, résoudre les sous-problèmes et recombiner les résultats. dont le calcul de la complexité génère des équations de récurrence de partitions qu'on peut résoudre facilement.

**Les étapes de la stratégie Diviser pour Régner**

La stratégie d'un algorithme « Diviser pour Régner » repose sur trois étapes à chaque niveau de la récursivité **:**

1. **Étape Diviser** : on divise le problème initial en un certain nombre de sous problèmes.
2. **Étape Régner** : on règne sur les sous-problèmes en les résolvant de manière récursive. Si la taille d'un sous-problème est suffisamment réduite, on peut toutefois le résoudre directement, cas de la base de la récurrence.
3. **Étape Combiner** : on forme la solution du problème initial en combinant les solutions partielles (les solutions des sous-problèmes).

**Schéma général d'un algorithme de type Diviser pour Régner**

```
Fonction DiviserRégner(P : Problème) : Solution
Début
  Si (la taille de P est petit) Alors
      CasdeBase(P);
  Sinon
```

```
    Diviser P en N sous-problèmes P1, P2, …, PN.
    Pour (i← 1 à N ) Faire
        Résoudre récursivement Pi : Si ← DiviserRégner(Pi);
    finPour;
    S ← Combiner(S1, S2, …, SN);      /*  S : Solution  */
  finSi;
Fin.
```

- CasdeBase : procédure effectuant le traitement de la base de la récursivité.
- Si ← DiviserRégner(Pi) : appel récursif pour un sous-problème.
- Combiner : procédure effectuant la combinaison des solutions partielles en la solution globale.

## 2. Diviser pour Régner : tri fusion

Le Tri Fusion utilise une stratégie différente : on divise le tableau à trier en deux parties (de tailles égales), que l'on trie, puis on interclasse les deux tableaux triés ainsi obtenus. La stratégie sous-jacente est du type Diviser pour Régner : on divise un problème sur une donnée de « grande taille » en sous-problèmes de même nature sur des données de plus petite taille

- On applique récursivement cette division jusqu'à arriver à un problème de très petite taille (objets élémentaires) et facile à traiter;
- On recombine les solutions des sous-problèmes pour obtenir la solution au problème initial.

**Input** : A list of n numbers $\sigma_1, \sigma_2, ..., \sigma_n$

**Output** : A permutation $\sigma_1, \sigma_2, ..., \sigma_n$ of the input such that $\sigma_1 \leq \sigma_2 \leq ... \leq \sigma_n$

Le but du jour est de voir la stratégie "Diviser pour régner" par l'exemple du tri-fusion. On discutera de l'implémentation des données sans forme de listes ou de tableau.

La stratégie est la suivante :

1. **Divide**. Eclater la donnée.
2. **Conquer**. Contracter les données.
3. **Combine**. Spécifique au problème…

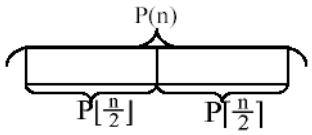



**Figure 4.** La fonction Divide pour éclater la donnée.

→ Si on a un tableau en entrée, on retourne une liste d'indices.
→ Si on a une liste en entrée, on retourne une liste d'éléments.

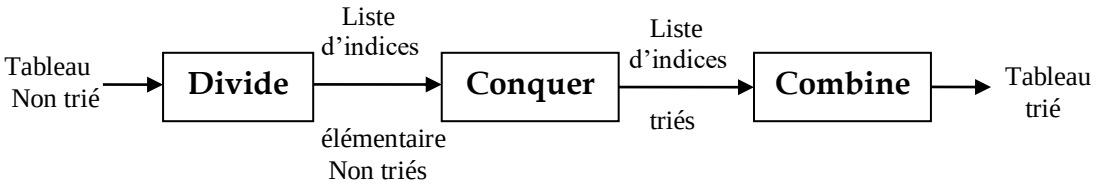



**Figure 5.** La stratégie Diviser pour Régner (Divide&Conquer)

## A. Les clauses de la stratégie Diviser pour Régner : tri fusion

1. **i=j → Tri(L,i,j)=i.∅**
2. **i<j →Tri(L,i,j)=C(L,Tri(T,I,(i+j)/2),Tri(T,(i+j)/2+1,j))**

Divide

**.....**

3. **L[i]≤ L[j] →C(L,i.I,j.J)=i.C(L,I,j.J)**
4. **L[i]> L[j] →C(L,i.I,j.J)=j.C(L,i.I,J)**
5. **C(L,∅ .J)=J**
6. **C(L,I, ∅)=I**

Conquer

Indices triés

**.....**

7. **TF(L)=TF1(L,Tri(L,1,n),C.i)**
8. **TF1(L,l.c,C.i)=TF1(L,l,C[i]← L[c],i+1)**
9. **TF1(L, ∅,C.i)=C**

Reconstitution du Tableau C à partir de la liste d'indice l.c.

Pour reconstituer un tableau, on récupère la liste d'indice triés l.c, le tableau d'origine L est un tableau résultat C. Tant qu'il y a des indices dans la liste, alors le tableau résultat contient l'élément de cet indice dans le tableau d'origine.

Et si au lieu d'un tableau, on a une liste ? Comment la couper en deux ? On ne veut surtout pas calculer la taille. Une façon plus simple est de prendre les pairs d'un côté et les impairs de l'autre. Ainsi :

$F(\emptyset) = \emptyset$

$F(C, \emptyset)=C$

$|L| \geq 2 \rightarrow F(L)=Comb(\underbrace{\underbrace{F(L_{impaire})}_{Divide},\underbrace{F(L_{paire})}_{Divide}}_{Conquer})$

Comb : Combine, la fonction de rassemblement

## B. La stratégie Diviser pour Régner pour Trier un tableau d'éléments

Les règles des trois sous fonction sont les suivantes:

- **La fonction Divide**

  i = j → Fп(T, i, j) = i.Ø (selon le problème)

  i < j → Fп(T, i, j) = C(T, Fп(T, i, (i+j)/2), Fп(T, (i+j)/2 +1, j))

- **La fonction Conquer**

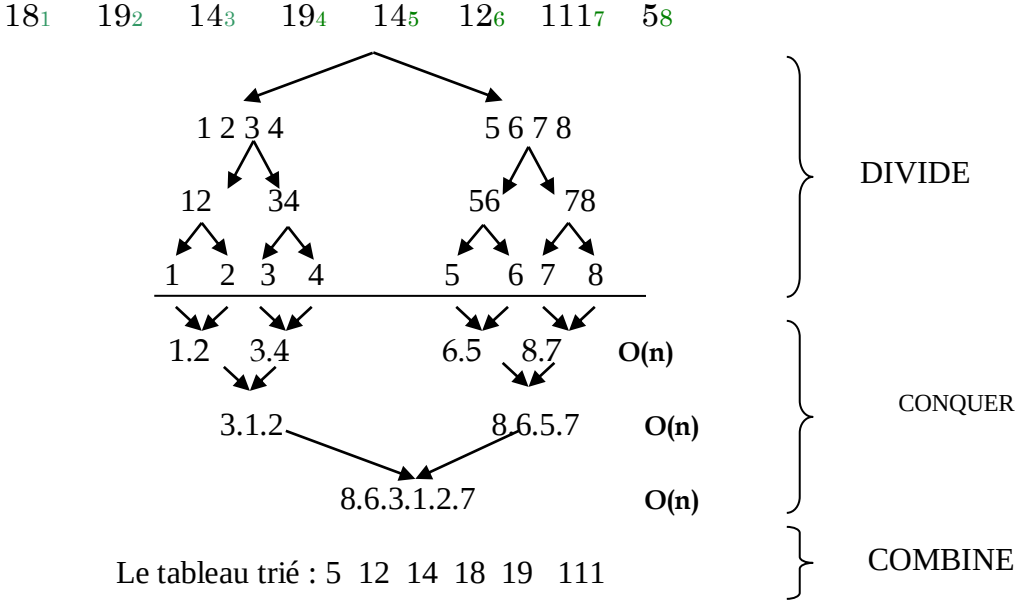
C est la fonction Conquer qui contracte les données. Elle est déclinée sur 3 types d'entrées :

C(Ø, l2) → l2, C(l1, Ø) → l1, et C(c1.l1, c2.l2) qui consistera à des comparaisons.

- **La fonction Combine**

Reconstituer la solution de problème on utilisant la solution trouvée de la fonction Conquer.

**Exemple :** application D&C pour le tri d'un tableau T, T=[18 19 14 19 14 12 111 5]




La relation de récurrence est T(1) = O(1), T(n) = 2T(n/2) + O(Tc).

Si O(Tc) est linéaire alors la complexité est en O(n.log(n)) ; s'il est constant, la complexité est linéaire.

## 3. Diviser pour Régner : produit de deux matrices

On considère deux matrices carrées (d'entiers) d'ordre n, M et N. Le produit de M par N est une matrice carrée C définie par :

$$Ci,j=\sum_{k=1}^{n} Mi,k \times Nk,j$$

1. Donner un algorithme calculant le produit de deux matrices représentées sous forme d'un tableau à deux dimensions. Calculer la complexité de cet algorithme. Doit-on préciser dans quels cas (pire cas, meilleur des cas, cas moyen) cette complexité est obtenue?

2. Modifier l'algorithme précédent lorsque la matrice M est de dimension (m,n) et la matrice N de dimension (n,p). Quelle est alors la complexité de l'algorithme?

On suppose que n est une puissance de 2. On découpe les matrices M et N en 4 blocs (n/2)x(n/2) comme suit:

$$M=\begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad N=\begin{pmatrix} E & F \\ G & H \end{pmatrix}$$ Et donc le produit est: $$MN=\begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

Soient maintenant :

P1 = A(F - H);
P2 = (A + B)H;
P3 = (C + D)E;
P4 = D(G - E);
P5 = (A + D)(E +H);
P6 = (B - D)(G + H);
P7 = (A - C)(E + F).

3. Avec les Pi en utilisant uniquement l'addition et la soustraction, Exprimer les expressions suivantes :

AE+BG;
AF+BH;
CE+DG et
CF+DH

4. En déduire un algorithme pour calculer le produit de matrices et évaluer sa complexité.

### A. Algorithme produit matriciel de dimension (n,n)×(n,n)

L'algorithme calculant le produit des deux matrices M et N est le suivant:

```
Algorithme ProduitMatrices(M,N : tableau de deux dimension [1..n][1..n])
Var
i, j,k : entier ;
Début
    Pour i ← 1 à n pas +1 faire
      Pour  j ← 1 à n pas +1 faire
          C[i, j] ← 0;
          Pour k ← 1 à n faire
              C[i, j]←C[i, j]+A[i,k]×B[k, j];
          finpour;
      finpour;
    finpour ;
  Fin.
```

**Complexité**

Si l'on s'intéresse au nombre a(n) d'additions (ou de multiplications) effectuées par l'algorithme, on obtient de façon immédiate que $a(n)= n^3$.

→ L'algorithme est en $\Theta(n^3)$, Il n'ya pas ni meilleur ni pire des cas.

**B. Algorithme produit matriciel de dimension (m,n)×(n,p)**

L'algorithme modifié est le suivant:

```
Algorithme ProduitMatrices(M: tableau de deux dimension [1..m][1..n] ; N: tableau
de deux dimension [1..n][1..p] )
Var
i, j,k :entier;
Début
    Pour i ← 1 à m pas +1 faire
      Pour j ← 1 à p pas +1 faire
          C[i, j] ← 0;
          Pour k ← 1 à n pas +1 faire
              C[i, j]←C[i, j]+A[i,k]×B[k, j];
          finpour;
      finpour;
   finpour;
Fin.
```

**Complexité**

Le nombre d'additions (ou de multiplications) effectuées est cette fois-ci a(n)= npm. En déduit que l'algorithme est en $\Theta(npm)$.

## C. La relation des expressions

Avec les (Pi=Xi) en utilisant uniquement l'addition et la soustraction.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} X \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix} = \begin{pmatrix} P6+P5-P2+P4 & P2+P1 \\ P4+P3 & P5-P7+P1-P3 \end{pmatrix}$$

## D. Algorithme de Strassen de produit de deux matrices

Supposons que la taille des matrice soit une puissance de 2: $N=2^n$, alors on peut toujours diviser récursivement les matrices en 4 de la manière suivante :

$A= \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}$ avec Aij de taille $N/2=2^{n-1}$

On peut alors écrire un programme de multiplication de 2 matrices qui utilisera les règles précédentes. Il utilisera les deux procédures suivantes :
C=MulMat(A,B,N), C=AddMat(A,B,N) et C=SubMat(A,B,N)

```
Fonction ProdMat(A, B, N)
    Début
    Si (N=1) alors    ProdMat ← A*B;
    Sinon
        début
            découper_en_4(A) ;
            découper_en_4(B) ;
        end ;
        P1  ← MulMat(A11, SubMat(B12,B22,N/2),N/2);
        P2  ← MulMat(AddMat(A11,A12,N/2),B22,N/2);
        P3  ← MulMat(AddMat(A21,A22,N/2),B11,N/2);
        P4  ←  MulMat(A22,SubMat(B21,B11,N/2),N/2);
        P5  ←  MulMat(AddMat(A11,A22,N/2),AddMat(B11,B22,N/2),N/2);
        P6  ←  MulMat(SubMat(A12,A22,N/2),AddMat(B21,B22,N/2),N/2);
        P7  ←  MulMat(SubMat(A11,A21,N/2), AddMat(B11,B12,N/2),N/2);
        C11 ← SubMat(AddMat(P6,P5,N/2),AddMat(P2,P4,N/2),N/2);
        C12 ←  AddMat(P2,P1,N/2);
        C21 ←  AddMat(P4,P3,N/2);
        C22 ←  AddMat(SubMat(P5,P7,N/2), SubMat(P1,P3,N/2),N/2);
        Recompose(C);
    End.
```

**Complexité de l'Algorithme de Strassen**

ProdMat(…, N) fait donc 7 appels à ProdMat(…, N/2) et 18 appels à AddMat(…, N/2). Il faut donc 7 multiplications et 18 additions au lieu de 8 multiplication et 4 addition.

- **Avantage** : C'est intéressant parce que le temps d'une multiplication est beaucoup plus grand que celui d'une addition.

➢ **Complexité**

- Si **n** est impair, la taille des matrices est augmenté (une ligne et une colonne nulle).
- Le temps de calcul vérifie l'équation de récurrence : **$T(n)=7T([n/2])+O(n^2)$**
  - ❖ $O(n^2)$ représente le temps des additions/soustractions et des recopies de coefficients.
  - ❖ L'équation de récurrence est de type : $T(n) = c.T(n/d) + O(n^k)$, alors: $c=7$ et $d^k = 2^2 = 4 \rightarrow c > d^k$,

$$T(n) = O(n^{\log \text{ base d de c}}) = O(n^{\log \text{ base 2 de 7}}) = O(n^{log_2^7}) = O(n^{2,80})$$

## 4. L'élément majoritaire et la dichotomie

### 4.1. Qu'est-ce que la majorité ?

Données : $S = \{\{x_1, \ldots, x_n\}\}$, $x_i$ appartenant à un univers U pas nécessairement ordonné.
Sortie : L'élément majoritaire de S ou bottom si aucune majorité ne se détache.

**Définition 2.**
Soit MAJ(x, S) le nombre d'occurrences de x dans S. Si MAJ(x, S) > |s|/2, alors c'est l'élément majoritaire.

### 4.2. Solution novice

Le principe est de compter le nombre d'occurrence de chaque élément : si cela totalise plus de la moitié des éléments du tableau, cet élément est majoritaire alors on s'arrête. Sinon, on compte le nombre d'occurrence de l'élément suivant.

On s'arrête une fois que l'on a parcouru plus de la moitié du tableau.

⇨ Complexité de l'algorithme en $n^2$, solution utilisée est pire.

```
i← 1;    j ← 2;
Algorithme MAJ(T : tableau[1..N]d'éléments ; i, n : entier)
Début
   si (i ≤ n/2) alors
     Compter(T, i, j, n, 0) ;
   Sinon
      écrire (''aucun n'était majoritaire'') ;
   finsi ;
```

F**in**.

**Algorithme** Compter(T : tableau [1..n] d'éléments ; i, j, n, r :entier)

**Début**
**Si** (j ≤ n et T[i] = T[j]) **alors**
Compter(T, i, j+1, n, r+1) ;
**finsi** ;
**Si** (j ≤ n et T[i] ≠ T[j]) **alors**
Compter(T, i, j+1, n, r) ;
**finsi** ;
**Si** (j > n et r ≥ n/2) **alors**
écrire (''l'élément majoritaire est :'',T[i]) ;
**finsi** ;
**Si** (j > n et r < n/2) **alors**
MAJ(T, i+1, n) ;
**finsi ;**
**Fin.**

### 4.3. Solution hacker

On trie le tableau et on compte le nombre d'occurrences de l'élément du milieu. L'élément majoritaire doit occuper le milieu du tableau et apparaît au moins (n/2) fois, sinon il n'y a pas d'élément majoritaire.

⇨ Complexité de l'algorithme en nlogn.

L'idée la plus simple est de partir du centre et de s'arrêter lorsque l'élément change :
- Partir du début, arrêter de compter quand on tombe sur notre élément
- Partir de la fin, arrêter de compter quand on tombe sur notre élément
- Faire la différence des indices ainsi obtenus.

MAJ(T,i,j,x) = MAJ(T,1,n,T[(1+n)/2]) x est l'élément du centre, celui dont on teste la quantité
T[i] ≠ x ^ T[j] ≠ x → MAJ(T, i, j, x) = MAJ(T, i+1, j-1, x)
T[i] ≠ x ^ T[j] = x → MAJ(T, i, j, x) = MAJ(T, i+1, j, x)
T[i] = x ^ T[j] ≠ x → MAJ(T, i, j, x) = MAJ(T, i, j-1, x)
T[i] = x ^ T[j] = x → MAj(T, i, j, x) = (j-i+1 > n/2)

On pose un pointeur sur la fin et un sur le début. Dès que l'un des deux rencontre notre élément, on le bloque. Lorsque les deux ont rencontrés l'élément, on calcule la différence et on regarde si elle est suffisante pour nous garantir que l'élément est majoritaire.

Bonne idée : Trier le tableau et penser que l'élément du centre est le seul à pouvoir être majoritaire et d'utiliser une recherche dichotomique, on recherche donc la borne supérieure et la borne inférieure par dichotomie.

$MAJ(T, n) = (BS(T, 1, n, T[(i+1)/2]) - BI(T, 1, n, T[(i+1)/2]) + 1) \geq n / 2$

On regarde si la différence entre la borne supérieure (BS) et la borne inférieure (BI) est suffisante pour garantir qu'il y a majorité ( supérieur strictement à n/2). BS et BI sont des recherches par dichotomie.

$BI(T, g, d, x) : g \leq d \wedge T[(g+d)/2] \geq x \rightarrow BI(T, g, d, x) = BI(T, g, (g+d)/2 - 1, x)$
$g \leq d \wedge T[(g+d)/2] < x \rightarrow BI(T, g, d, x) = BI(T, (g+d)/2 + 1, d, x)$
$g > d \rightarrow BI(T, g, d, x) = g$
$BS(T, g, d, x) : g \leq d \wedge T[(g+d)/2] \geq x \rightarrow BI(T, g, d, x) = BI(T, (g+d)/2 + 1, d, x)$
$g \leq d \wedge T[(g+d)/2] < x \rightarrow BI(T, g, d, x) = BI(T, g, (g+d)/2 - 1, x)$
$g > d \rightarrow BI(T, g, d, x) = d$