REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'enseignement supérieur et de la recherche scientifique

Université Larbi Ben M'Hidi Oum El Bouaghi

Faculté des sciences exactes et science de la nature et de la vie

Département de mathématique d'informatique et

Cours : Méthodes Algorithmiques

Chapitre 2 : Diviser pour régner



Proposé par Dr. Kenza Belhouchette

Edition 1.0

2024/ 2025

Introduction au Paradigme : Diviser pour Régner

Le paradigme **"Diviser pour régner"** est une approche algorithmique puissante et largement utilisée pour résoudre des problèmes complexes en informatique. Il repose sur un principe simple : au lieu d'aborder directement un problème dans son ensemble, on le divise en sous-problèmes plus petits et plus simples à traiter. Ces sous-problèmes sont ensuite résolus de manière récursive, et leurs solutions sont combinées pour donner la solution finale du problème initial.

Cette méthode permet de simplifier considérablement la conception d'algorithmes pour des problèmes de grande taille, tout en offrant des gains d'efficacité dans de nombreux cas. Le paradigme "diviser pour régner" est au cœur de nombreux algorithmes classiques, tels que le tri-fusion (Merge Sort), la recherche dichotomique (Binary Search) et l'algorithme de Strassen pour la multiplication de matrices.

En plus d'être une approche élégante et modulaire, "diviser pour régner" permet souvent d'améliorer la complexité temporelle des algorithmes, les rendant plus performants face à de grandes quantités de données.

Au sein de ce chapitre, nous explorerons en détail comment le paradigme "Diviser pour régner" permet de résoudre efficacement une large gamme de problèmes, tout en examinant les avantages et les limites de cette méthode.

1. Définition

Le paradigme **"Diviser pour régner"** est un concept algorithmique qui vise à résoudre un problème en le décomposant en plusieurs sous-problèmes plus petits et indépendants. Chaque sous-problème est résolu récursivement, et les solutions obtenues sont ensuite combinées pour former la solution finale du problème initial. Ce paradigme est particulièrement efficace pour les problèmes complexes et volumineux, car il permet d'aborder chaque partie du problème de manière plus simple et modulaire.

2. Formulation mathématique

La définition mathématique du paradigme **"Diviser pour régner"** repose sur la résolution de problèmes par récurrence. Un problème de taille n est divisé en a sous-problèmes de taille n\b

où $a \geq 1$ et $b > 1$.

Chaque sous-problème est ensuite résolu de manière récursive. Le coût total de la méthode peut être exprimé par une relation de récurrence de la forme :

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$$

où :

- T(n) est le temps d'exécution pour résoudre un problème de taille n,
- a·T représente le coût pour résoudre les a sous-problèmes de taille n\b,
- $O(n^d)$est le coût de la combinaison des solutions des sous-problèmes.

3. Principe de l'algorithme : Diviser pour Régner

L'approche diviser pour régner repose sur trois étapes fondamentales :

3.1 La division

Dans cette première étape, le problème de départ est divisé en sous-problèmes plus petits et indépendants, qui sont de même nature que le problème initial. La division doit être effectuée de manière à ce que les sous-problèmes soient plus faciles ou plus rapides à résoudre que le problème global.

Nature de la division : Selon la structure du problème, cette division peut se faire de manière simple (en deux parties, par exemple, pour une recherche dichotomique), ou en plusieurs sous-problèmes (comme dans le cas du tri fusion ou de l'algorithme de Strassen pour la multiplication des matrices).

Objectif de la division : L'objectif est de réduire la complexité du problème à chaque étape en décomposant celui-ci en sous-problèmes plus petits jusqu'à atteindre un point où la solution devient évidente ou triviale (comme résoudre un tableau à un seul élément dans un tri récursif).

3.2 Résolution des sous-problèmes

Après avoir divisé le problème, chaque sous-problème est résolu de manière récursive. Si le sous-problème est encore trop grand pour être résolu directement, il est à son tour divisé suivant le même schéma.

Résolution récursive : La plupart des algorithmes utilisant cette approche sont récursifs. Cela signifie que chaque sous-problème est résolu en appliquant la même stratégie de division, résolution et combinaison jusqu'à atteindre une condition de terminaison (souvent appelée **cas de base**).

Cas de base : La récursion continue jusqu'à ce que les sous-problèmes soient suffisamment petits pour être résolus directement sans besoin de division supplémentaire. Le cas de base est généralement trivial, comme un tableau de taille 1 dans un algorithme de tri ou un cas où la division ne produit plus de sous-problèmes significatifs.

3.3 Combinaison des solutions partielles

Une fois les sous-problèmes résolus, les résultats obtenus sont combinés pour former la solution du problème initial. Cette étape est cruciale, car elle doit se faire efficacement pour ne pas annuler les gains obtenus lors des étapes de division et de résolution.

Fusion des résultats : Par exemple, dans le tri fusion (merge sort), une fois les sous-tableaux triés, ils sont combinés pour obtenir un tableau trié complet. Dans les algorithmes de

multiplication de matrices, les matrices partielles sont recombinées pour obtenir le produit global.

Coût de la combinaison : Le coût de cette étape varie selon les problèmes et les algorithmes. Dans certains cas, la combinaison est triviale (comme dans la recherche binaire), tandis que dans d'autres, elle peut être aussi coûteuse que les étapes précédentes, voire plus (comme dans le tri fusion).

4. Stategie de Diviser pour régner

En général, les algorithmes de diviser pour régner sont appliqués selon deux stratégies principales :

- La première stratégie est **la récursion sur les données** : où les données sont directement décomposées en un certain nombre de partitions. Cela donne lieu à des sous-problèmes qui sont résolus récursivement à l'aide de la même fonction. À la fin, les résultats obtenus sont combinés afin de calculer la solution complète.
- La deuxième stratégie est **la récursion sur les résultats**, qui effectue d'abord un prétraitement avant de couper les données en partitions. Ensuite, elle résout récursivement les sous-problèmes résultants en utilisant la même fonction. Enfin, elle combine les résultats obtenus afin de calculer la solution complète.

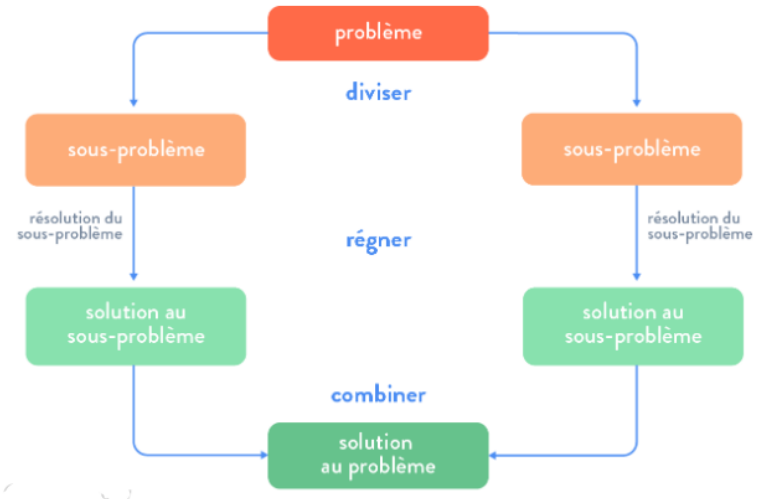


Fig 1. Stratégie diviser pour régner

5. La récursivité

La récursivité est une technique de programmation et un concept algorithmique où une fonction s'appelle elle-même pour résoudre un problème. Elle est souvent utilisée pour simplifier la conception et l'implémentation d'algorithmes, surtout pour les problèmes qui peuvent être divisés en sous-problèmes similaires.

Concepts de Base de la Récursivité

1. **Définition :** Une fonction est dite récursive si elle s'appelle elle-même directement ou indirectement. La récursivité est un outil puissant pour résoudre des problèmes complexes en les décomposant en parties plus simples.
2. **Cas de Base :** Chaque fonction récursive doit avoir un cas de base, qui est une condition dans laquelle la fonction renvoie une valeur sans appeler elle-même. Cela empêche la récursion de continuer indéfiniment.
3. **Appels Récursifs :** Si le cas de base n'est pas atteint, la fonction s'appelle elle-même avec des paramètres modifiés. Cela doit réduire progressivement la taille du problème, menant finalement au cas de base.

Structure d'une Fonction Récursive

Une fonction récursive typique comprend les éléments suivants :

```
def fonction_recursive(paramètres):
    if condition_de_base:
        # Cas de base
        return valeur
    else:
        # Appel récursif
return fonction_recursive(nouveaux_paramètres)
```

Avantages de la Récursivité

1. **Simplicité :** Les solutions récursives sont souvent plus simples et plus faciles à comprendre que les solutions itératives, surtout pour des problèmes complexes comme les arbres et les graphes.
2. **Facilité d'implémentation :** De nombreux algorithmes, comme le tri rapide, la recherche binaire, et les algorithmes de traversée d'arbres, sont naturellement récursifs.

Inconvénients de la Récursivité

1. **Performance :** Les appels récursifs peuvent consommer plus de mémoire et de temps d'exécution que les solutions itératives, surtout pour des problèmes avec une profondeur de récursion importante (par exemple, le calcul de Fibonacci avec une implémentation naïve).
2. **Limite de profondeur :** Les langages de programmation ont souvent une limite de profondeur de récursion, ce qui peut entraîner un débordement de pile si la fonction est appelée trop de fois.

Optimisation de la Récursivité

Pour améliorer les performances des algorithmes récursifs, certaines techniques peuvent être appliquées :

1. **Mémorisation :** Stocker les résultats des appels de fonction récursifs dans une structure de données (comme un dictionnaire) pour éviter de recalculer les mêmes valeurs. Cela est particulièrement utile dans des problèmes comme la suite de Fibonacci.

2. **Récursion terminale :** Réécrire la fonction récursive pour qu'elle soit une récursion terminale, où l'appel récursif est la dernière opération de la fonction, permettant à certains langages de programmation d'optimiser l'utilisation de la pile.

6. Exemples illustratifs

5.1 Probleme de Hanoi

Le problème de Hanoi consiste à déplacer un ensemble de disques de différentes tailles d'une tour à une autre, en suivant des règles spécifiques. Ce casse-tête se base sur trois tours et un nombre variable de disques.

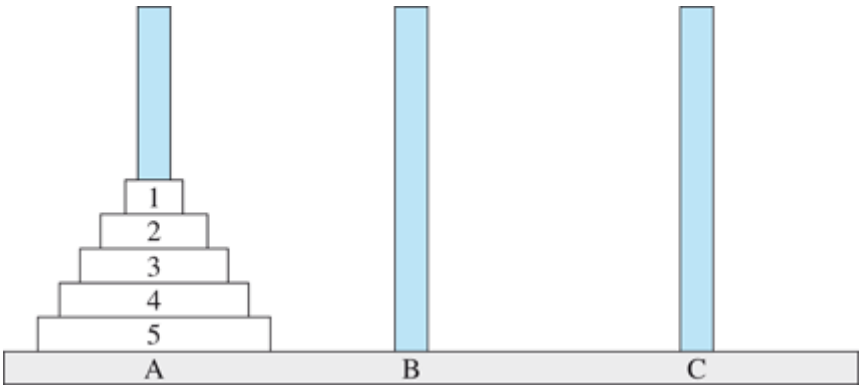


Fig 2. Problème de Hanoi

Description des Éléments

1. **Tours :** Il y a trois tours étiquetées A, B et C.
2. **Disques :** Les disques sont empilés sur la tour A, le plus petit disque étant au sommet. Les disques peuvent être de différentes tailles, et le but est de déplacer tous les disques vers la tour C.

Règles de Déplacement

- Un seul disque peut être déplacé à la fois.
- Un disque ne peut être placé que sur une tour vide ou sur un disque plus grand.

Stratégie de Résolution

L'algorithme de Hanoi utilise une approche récursive pour résoudre le problème. La stratégie consiste à décomposer le problème en sous-problèmes plus simples.

Étapes de l'Algorithme

1. **Déplacement de n-1 disques :** Pour déplacer n disques de la tour A à la tour C, commencez par déplacer les n-1 disques supérieurs de A à B, en utilisant la tour C comme intermédiaire.
2. **Déplacement du disque n :** Une fois que les n-1 disques sont en B, déplacez le disque n (le plus grand) de la tour A à la tour C.

3. **Retour des n-1 disques :** Déplacez ensuite les n-1 disques de B à C, en utilisant la tour A comme auxiliaire.

Formulation Récursive

L'algorithme peut être exprimé de manière récursive comme suit :

- **Cas de base :** Si le nombre de disques est 1, déplacez simplement le disque de la tour source à la tour cible.
- **Cas général :** Pour n disques :

- Déplacez n-1 disques de la tour source à la tour auxiliaire.
- Déplacez le disque n de la tour source à la tour cible.
- Déplacez n-1 disques de la tour auxiliaire à la tour cible.

Complexité de l'Algorithme

Le nombre minimum de mouvements requis pour résoudre le problème avec n disques est donné par la formule :

$$M(n)=2^n-1$$

Cela indique que la complexité croît exponentiellement avec le nombre de disques.

Exemple d'Exécution

Prenons un exemple avec trois disques :

1. **Appel initial :** hanoi(3, 'A', 'C', 'B')

- Déplacer les 2 disques de A à B (avec C comme intermédiaire).
- Déplacer le disque 3 de A à C.
- Déplacer les 2 disques de B à C (avec A comme intermédiaire).

Les mouvements pour cet exemple seraient :

- Déplacer le disque 1 de A à C.
- Déplacer le disque 2 de A à B.
- Déplacer le disque 1 de C à B.
- Déplacer le disque 3 de A à C.
- Déplacer le disque 1 de B à A.
- Déplacer le disque 2 de B à C.
- Déplacer le disque 1 de A à C.

5.1 Tri par Fusion (Merge Sort)

Le tri par fusion est un excellent exemple de la technique "diviser pour régner". Il fonctionne de la manière suivante :

1. **Diviser** : Divisez le tableau en deux moitiés.
2. **Récursivité** : Appliquez le tri par fusion de manière récursive à chaque moitié jusqu'à ce que chaque sous-tableau contienne un seul élément (un tableau de taille un est déjà trié).
3. **Combiner** : Fusionnez les deux sous-tableaux triés pour former un tableau trié.

Algorithme de Tri par Fusion (Merge Sort)

```
Algorithme : Tri fusion
tri_fusion(tableau) :
    Si la longueur de tableau est > 1 alors :
        # Séparer
        milieu = longueur // 2
        gauche = tableau[0:milieu]  # Première moitié
        droite = tableau[milieu:]    # Seconde moitié

        # Diviser
        tri_fusion(gauche)          # Trier la première moitié
        tri_fusion(droite)          # Trier la seconde moitié

        # Combiner
        fusion(tableau, gauche, droite)
Algorithme fusion
fusion(tableau, gauche, droite) :
    i, j, k = 0
    tant que i < longueur de gauche et j < longueur de droite faire :
        Si gauche[i] < droite[j] alors :
            tableau[k] = gauche[i]
            i = i + 1
        Sinon :
            tableau[k] = droite[j]
            j = j + 1
```

```
        k = k + 1

    # Ajouter les éléments restants de gauche
    tant que i < longueur de gauche faire :
        tableau[k] = gauche[i]
        i = i + 1
        k = k + 1

    # Ajouter les éléments restants de droite
    tant que j < longueur de droite faire :
        tableau[k] = droite[j]
        j = j + 1
        k = k + 1
```

Exemple illustratif de l'algorithme de Tri par Fusion

Trier le tableau [38, 27, 43, 3, 9, 82, 10]

1. Initialisation :

   [38, 27, 43, 3, 9, 82, 10]

2. Division :

   [38, 27, 43] | [3, 9, 82, 10]

   [38] [27, 43] | [3, 9] [82, 10]

   [27] [43] | [3] [9] [82] [10]

3. Fusion :

   - Fusion de [27] et [43] → [27, 43]
   - Fusion de [38] et [27, 43] → [27, 38, 43]
   - Fusion de [3] et [9] → [3, 9]
   - Fusion de [82] et [10] → [10, 82]
   - Fusion de [3, 9] et [10, 82] → [3, 9, 10, 82]

4. Fusion finale :

   - Fusion de [27, 38, 43] et [3, 9, 10, 82] → [3, 9, 10, 27, 38, 43, 82]

5. Résultat final trié :

[3, 9, 10, 27, 38, 43, 82]

Complexité en Temps

1. **Étape de Division** :

   À chaque étape de récursion, l'algorithme divise le tableau en deux parties égales. Pour un tableau de taille nnn, on continue à diviser jusqu'à obtenir des sous-tableaux de taille 1. Le nombre de divisions nécessaires pour atteindre des sous-tableaux de taille 1 est de $\log_2(n)$, car chaque division divise la taille du tableau par deux.

2. **Étape de Fusion** :

   À chaque niveau de récursion, tous les éléments sont parcourus pour fusionner les sous-tableaux. Cela prend O(n) opérations par niveau.

3. **Complexité Totale** :

   Puisque nous avons $\log_2(n)$,niveaux de récursion et que chaque niveau nécessite O(n) opérations pour la fusion, la complexité totale en temps est donnée par :

$$O(n \log n)$$

5.2 Recherche Binaire

La recherche binaire est une méthode rapide pour localiser un élément spécifique dans un tableau ordonné. Elle procède en coupant systématiquement en deux la plage de recherche à chaque itération, ce qui diminue le nombre d'éléments à vérifier. À chaque étape, on compare l'élément central de l'intervalle restant avec l'élément recherché, puis on ajuste l'intervalle en fonction du résultat. Il est important que le tableau soit trié au préalable, sinon l'algorithme ne sera pas efficace.

Pseudocode pour la recherche binaire

```
Fonction RechercheBinaire(A, cible, gauche, droite):
    Si gauche > droite:
        Retourner -1  // Élément non trouvé
    mid = (gauche + droite) // 2
    Si A[mid] == cible:
        Retourner mid  // Élément trouvé
    Sinon Si A[mid] < cible:
        Retourner RechercheBinaire(A, cible, mid + 1, droite)  // Cherche dans la moitié droite
```

Sinon:

Retourner RechercheBinaire(A, cible, gauche, mid - 1) // Cherche dans la moitié gauche

Exemple illustratif de Recherche Binaire

Tableau d'entrée :

A = [1, 3, 5, 7, 9, 11]

Élément à chercher :

cible = 7

Étapes de la recherche binaire :

1. Initialisation :

gauche = 0, droite = 5 (indices du tableau).

2. Première itération :

mid = (0 + 5) // 2 = 2

Comparer A[2] = 5 avec 7.

5 < 7, donc ajuster gauche à mid + 1 = 3.

3. Deuxième itération :

gauche = 3, droite = 5.

mid = (3 + 5) // 2 = 4

Comparer A[4] = 9 avec 7.

9 > 7, donc ajuster droite à mid - 1 = 3.

4. Troisième itération :

gauche = 3, droite = 3.

mid = (3 + 3) // 2 = 3

Comparer A[3] = 7 avec 7.

A[3] == 7, donc retourner 3.

Résultat final :

L'élément 7 a été trouvé à l'indice 3 dans le tableau.

5.3 Trouver le Maximum dans un Tableau

L'algorithme de recherche du maximum dans un tableau utilise le principe de "diviser pour régner" pour diviser le tableau en sous-tableaux et trouver le maximum dans chacun d'eux, puis combine ces résultats pour obtenir le maximum global.

Si le tableau contient un seul élément, cet élément est le maximum.

Pseudocode Pour l'algorithme le maximum d'un tableau

```
Fonction TrouverMaximum(A, gauche, droite):
    Si gauche == droite:
        Retourner A[gauche]  // Cas de base : un seul élément
    mid = (gauche + droite) // 2
    maxG = TrouverMaximum(A, gauche, mid)  // Maximum dans la première moitié
    maxD = TrouverMaximum(A, mid + 1, droite)  // Maximum dans la deuxième moitié


// Retourner le plus grand des deux maximums trouvés
SI maxG > maxD ALORS
RETOURNER maxG SINON
 RETOURNER maxD
FIN SI
 FIN FONCTION
```

Exemple illustratif de recherche le maximum d'un tableau

Tableau : A = [3, 1, 4, 1, 5, 9, 2, 6]

gauche = 0, droite = 7

1. Première division :

   milieu = (0 + 7) // 2 = 3

   Sous-tableau gauche : [3, 1, 4, 1] (indice 0 à 3)

   Sous-tableau droit : [5, 9, 2, 6] (indice 4 à 7)

2. Sous-tableau gauche [3, 1, 4, 1] :

milieu = (0 + 3) // 2 = 1

Sous-tableau gauche : [3, 1] (indice 0 à 1)

Sous-tableau droit : [4, 1] (indice 2 à 3)

a. Sous-tableau gauche [3, 1] :

milieu = (0 + 1) // 2 = 0

Sous-tableau gauche : [3] (indice 0) → Cas de base, retourne 3

Sous-tableau droit : [1] (indice 1) → Cas de base, retourne 1

Comparaison : max(3, 1) = 3

b. Sous-tableau droit [4, 1] :

milieu = (2 + 3) // 2 = 2

Sous-tableau gauche : [4] (indice 2) → Cas de base, retourne 4

Sous-tableau droit : [1] (indice 3) → Cas de base, retourne 1

Comparaison : max(4, 1) = 4

Comparaison globale pour [3, 1, 4, 1] :

Comparaison : max(3, 4) = 4

3. Sous-tableau droit [5, 9, 2, 6] :

milieu = (4 + 7) // 2 = 5

Sous-tableau gauche : [5, 9] (indice 4 à 5)

Sous-tableau droit : [2, 6] (indice 6 à 7)

a. Sous-tableau gauche [5, 9] :

milieu = (4 + 5) // 2 = 4

Sous-tableau gauche : [5] (indice 4) → Cas de base, retourne 5

Sous-tableau droit : [9] (indice 5) → Cas de base, retourne 9

Comparaison : max(5, 9) = 9

b. Sous-tableau droit [2, 6] :

milieu = (6 + 7) // 2 = 6

Sous-tableau gauche : [2] (indice 6) → Cas de base, retourne 2

Sous-tableau droit : [6] (indice 7) → Cas de base, retourne 6

Comparaison : max(2, 6) = 6

Comparaison globale pour [5, 9, 2, 6] :

Comparaison : max(9, 6) = 9

4. Comparaison finale pour tout le tableau [3, 1, 4, 1, 5, 9, 2, 6] :

Comparaison : max(4, 9) = 9

Le maximum du tableau est 9.

6.4 Algorithme de Strassen (multiplication de matrices)

L'algorithme de Strassen est une méthode algébrique efficace pour la multiplication de matrices carrées, introduite par Volker Strassen en 1969. Cet algorithme réduit le nombre d'opérations nécessaires pour effectuer une multiplication de matrices par rapport à la méthode classique. Il est basé sur la stratégie du *diviser pour régner*.

Contexte

Pour deux matrices carrées A et B de taille n x n, l'approche classique de multiplication matricielle nécessite $O(n^3)$ opérations de multiplication scalaire, ce qui devient inefficace pour de grandes matrices. L'algorithme de Strassen parvient à améliorer cette complexité en la réduisant à environ $O(n^{2.81})$, ce qui est significatif pour des matrices de grande dimension.

Principe de l'algorithme

Strassen repose sur une décomposition des matrices en sous-matrices de taille n/2 x n/2, réduisant le nombre de multiplications par une approche récursive.

Soit A et B deux matrices n x n (où n est une puissance de 2 pour simplifier l'explication). Elles peuvent être décomposées en quatre sous-matrices :

$$A= \begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix} \quad \text{ET B=} \quad \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix}$$

La multiplication classique de matrices nécessiterait huit produits de ces sous-matrices. Strassen, cependant, utilise une stratégie astucieuse pour n'en effectuer que sept.

Les sept produits intermédiaires de Strassen sont définis comme suit :

M1 = (A11 + A22)(B11 + B22)

M2 = (A21 + A22) B11

M3 = A11 (B12 - B22)

M4 = A22 (B21 - B11)

M5 = (A11 + A12) B22

M6 = (A21 - A11)(B11 + B12)

M7 = (A12 - A22)(B21 + B22)

À partir de ces produits intermédiaires, les sous-matrices du produit final C = A x B sont obtenues en combinant les résultats de manière efficace :

C11 = M1 + M4 - M5 + M7

C12 = M3 + M5

C21 = M2 + M4

C22 = M1 - M2 + M3 + M6

Complexité

L'intérêt principal de l'algorithme de Strassen réside dans sa réduction du nombre de multiplications requises, passant de 8 à 7. Cette optimisation a un impact direct sur la complexité temporelle :

- Multiplication classique : $O(n^3)$

- Algorithme de Strassen : $O(n^{\log_2 7}) \approx O(n^{2.81})$

Algorithme de Strassen

```
function Strassen(A, B):
    n = dimension(A)  // taille des matrices A et B
    if n == 1:
        return A[0][0] * B[0][0]
    // Diviser A et B en sous-matrices
    A11, A12, A21, A22 = divide(A)
    B11, B12, B21, B22 = divide(B)
    // Calculer les 7 produits intermédiaires
    M1 = Strassen(A11 + A22, B11 + B22)
    M2 = Strassen(A21 + A22, B11)
    M3 = Strassen(A11, B12 - B22)
    M4 = Strassen(A22, B21 - B11)
    M5 = Strassen(A11 + A12, B22)
    M6 = Strassen(A21 - A11, B11 + B12)
    M7 = Strassen(A12 - A22, B21 + B22)
```

```
// Combiner les résultats pour former la matrice résultante
C11 = M1 + M4 - M5 + M7
C12 = M3 + M5
C21 = M2 + M4
C22 = M1 - M2 + M3 + M6
// Retourner la matrice C
return combine(C11, C12, C21, C22)
```

Bien que cet algorithme soit plus rapide pour des matrices de grande taille, il comporte certaines limitations pratiques. Notamment, la décomposition et la gestion des additions et soustractions supplémentaires peuvent rendre cette approche moins efficace pour des matrices de petite taille.

Les principaux avantages et inconvénients de l'algorithme de Strassen

Les avantages

- Réduction de la Complexité Temporelle
- Pour des matrices de grande dimension, Strassen est généralement plus rapide que l'algorithme classique, car il réduit le nombre total d'opérations nécessaires, rendant l'algorithme plus performant.
- Dans certaines applications, comme le traitement d'image et la résolution de systèmes d'équations linéaires, la réduction de la complexité peut conduire à des améliorations de performances notables.

Inconvénients

- L'algorithme de Strassen est plus complexe à implémenter que la méthode classique, ce qui peut rendre son utilisation difficile pour certains développeurs.
- Bien que l'algorithme réduise le nombre de multiplications, il introduit un plus grand nombre d'additions et de soustractions, ce qui peut compenser les gains réalisés dans des scénarios où les opérations d'addition sont coûteuses.
- Pour des matrices de petite taille, la méthode classique peut être plus efficace en raison de la surcharge liée à l'algorithme de Strassen et de la gestion des sous-matrices.

Application avancées de l'algorithme de Strassen

1. Traitement d'Images

- Filtres et Transformées : L'algorithme de Strassen est utilisé dans les opérations de filtrage d'images et de transformation (comme la transformation de Fourier), où la multiplication de grandes matrices est fréquente.
- Compression d'Images : Dans des algorithmes comme JPEG, Strassen peut être utilisé pour effectuer des multiplications de matrices dans le cadre des opérations de compression et de décompression.

2. Résolution de Systèmes d'Équations Linéaires

Strassen est appliqué dans des méthodes de résolution rapide pour des systèmes d'équations linéaires, en particulier ceux qui peuvent être formulés sous forme matricielle. Cela inclut des applications dans la modélisation physique, la mécanique des fluides et d'autres domaines d'ingénierie.

3. Calcul Scientifique et Ingénierie

Dans les simulations numériques, comme celles utilisées dans la dynamique des fluides ou la mécanique des structures, où de grands systèmes d'équations doivent être résolus efficacement.

4. Apprentissage Automatique et Intelligence Artificielle

Les réseaux de neurones, en particulier les réseaux de neurones profonds, impliquent de nombreuses multiplications matricielles. L'utilisation de Strassen peut accélérer les calculs lors de l'entraînement et de l'inférence.

5. Graphes et Traitement de Données

Les algorithmes de Strassen peuvent être appliqués à la multiplication de matrices d'adjacence dans les graphes, ce qui est utile pour des applications en théorie des graphes, telles que la recherche de chemins ou la détermination des composants connexes.

Conclusion

En conclusion, la méthode "Diviser pour régner" constitue un pilier essentiel de l'algorithmique en permettant la résolution efficace de problèmes complexes. En fragmentant un problème global en sous-problèmes plus simples, cette approche permet d’optimiser les ressources, de réduire la complexité de traitement, et d’accélérer le calcul. Des exemples comme l'algorithme de Strassen pour la multiplication de matrices ou la recherche binaire illustrent bien les gains en performance et en efficacité obtenus grâce à cette méthode.

Ainsi, "Diviser pour régner" ne se limite pas à une technique parmi d’autres; elle offre une base solide pour aborder et concevoir des algorithmes avancés. Elle constitue également un point de départ pour approfondir des méthodes complémentaires, telles que la programmation dynamique et les algorithmes de tri. Maîtriser cette approche est donc fondamental pour quiconque souhaite s’initier à la résolution algorithmique de problèmes complexes en informatique.