1. **Diviser pour Régner**

* **Algorithme de Strassen**

Pour effectuer le produit de deux matrice de taille 2X2 il faut faire 8 multiplications. Plus les tailles des matrices sont grandes plus ce nombre évolue soit N3 pour des matrices de taille N. Volker Strassen a proposer un algorithme de type DPR pour qui réduit le nombre de multiplications nécessaire pour le produit de deux matrices. En utilisant cet algorithme, le nombre de multiplications pour le produit de deux matrices de taille 2x2 passerait de 8 à 7. Pour des matrices de taille N on aura N2.81. Cet algorithme est utilisé pour des matrice dont la taille est une puissance de 2.

Fonction Strassen (A,B : Matrice, n :Entier) : Matrice

Debut

Si n=1 alors

Retourner A\*B

Sinon

E1 ← Strassen(A[1][2] – A[2][2] , B[2][1] + B[2][2] , n/2)

E2 ← Strassen(A[1][1] + A[2][2] , B[1][1] + B[2][2] , n/2)

E3 ← Strassen(A[1][1] – A[2][1] , B[1][1] + B[1][2] , n/2)

E4 ← Strassen(A[1][1] + A[1][2] , B[2][2] , n/2)

E5 ← Strassen(A[1][1] , B[1][2] – B[2][2] , n/2)

E6 ← Strassen(A[2][2] , B[2][1] – B[1][1] , n/2)

E7 ← Strassen(A[2][1] + A[2][2] , B[1][1] , n/2)

C[1][1] ← E1 + E2 - E4 + E6

C[1][2] ← E6 + E5

C[2][1] ← E6 + E7

C[2][2] ← E2 – E3 + E5 – E7

Retourner C

fsi

* **Algorithme de Karatsuba**

Pour effectuer le produit de deux nombres de n chiffre en utilisant la méthode naïve, on fait **n2** multiplication élémentaires, le coût en calcul d’un tel produit est **T(n)=O(n2)**.

L’algorithme de Karatsuba est une application du principe « Diviser pour régner » qui permet d’améliorer le coût des multiplications des grands nombres. Le coût de cette méthode est **T(n)=O(n1.585)**

Fonction karatsuba(a,b : Entier) :Entier

Debut

n ← max(E(log2a) , E(log2b)) + 1

si n <= 1 alors

retourner a\*b

sinon

k=n/2

a0 ← a mod 2^k

a1 ← a/2^k

b0 ← b mod 2^k

b1 ← b/2^k

x ← karatsuba(a0 , b0)

y ← karatsuba(a1 , b1)

z ← karatsuba(abs(a1 – a0) , abs(b1 – b0))

retourner y\*2^(2\*k) + (x + y – sign(a1 – a0)\*sign(b1 – b0)\*z)\*2^k + x

fsi

fonction abs(n :Entier) :Entier

Debut

Si n >= 0 alors

Retourner n

Sinon

Retourner 0 – n

Fsi

Fonction sign(n : Entier) : Entier

Debut

Si n >= 0 alors

Retourner 1

Sinon

Retourner -1

Fsi

Fin

1. **Glouton**

* **Sac à dos**

On a un sac à dos de poids maximale **P** et n objets **(vi, Pi)** de valeurs et de poids diverses. On souhaite remplir le sac de façon à cumuler un maximum de valeur sans dépasser sa capacité. Quel est le bénéfice maximum que l’on puisse faire ?

Pour résoudre ce problème de manière gloutonne, on tri les objets par ordre décroissant de et on choisit les objets jusqu’à ce que le poids libre ne puisse plus recevoir d’objet.

Algorithme Sac\_a\_dos

Debut

X←trier(X) //par odre décroissant de

W←0 , Valeur←0

Sac←Φ

i←1

Tanque P-W > 0 et i<=n faire

Si P-W>=Poids[i] alors

W←W + Poids[i]

valeur←valeur + V[i]

Enfiler(Sac,X[i])

Fsi

i←i+1

Ftq

Fin

* **Rendu de monnaie**

Nous considérons n pièces de différentes valeurs **vi**. Notons **N(x)** le nombre minimum de pièces pour obtenir x.

Algorithme Monnaie

Debut

X←trier(X) //par odre décroissant de vi

R←S

Piece←Φ

i←1

Tanque R > 0 et i<=n faire

Si R>=X[i] alors

R←R - X[i]

Enfiler(Piece,X[i])

Fsi

i←i+1

Ftq

Fin

* Location de voiture

1. **Programmation dynamique**

* **Sac à dos**

On a un sac à dos de poids maximale **W** et **n** objets **(vi, wi)** de valeurs et de poids diverses. On souhaite remplir le sac de façon à cumuler un maximum de valeur sans dépasser sa capacité. Quel est le bénéfice maximum que l’on puisse faire ?

On désigne par **Gij** le gain maximum généré par le choix des **i** premiers objets dont la somme des poids ne dépasse pas **j.** En calculant **Gij** la séquence peut être divisée en deux : les i-1 premiers objets et l’objet i. l’objet i est soit choisi, soit ignoré dans **Gij.** S’il est choisi, avant de l’inclure on doit de rassurer que son poids ne dépasse pas la capacité j du sac. Si tel est le cas, alors il contribue à la solution optimale par son gain vi et on a alors :

**Gij = Gi-1 , j-wi + vi**

Sinon

**Gij =Gi-1** , j

On déduit donc :

**Gij = max{ Gi-1** ,j , **Gi-1 , j-wi + vi }**

**G0j = 0**

**Gi0 = 0**

**Gij = Gi-1** ,j si j<wi et i>0

On a donc l’algorithme

Algorithme Sac\_a\_dos

Debut

Pour i de 1 à n faire

G[i,0] ← 0

Fpour

Pour j de 1 à W faire

G[0,j] ← 0

Fpour

Pour i de 1 à n faire

Pour j de 1 à W faire

Si j >= wi alors

G[i,j] ← max{ G[i-1, j-wi] + vi , G[i-1, j] }

Sinon

G[i,j] ← G[i-1, j]

Fsi

Fpour

Fpour

Fin

* **Multiplication de chaîne de matrices**

Soit n matrices **M1, M2, …, Mn**; chaque matrice Mi possède di-1 ligne et di le nombre de colonne. Le problème est d’effectuer **M1 x M2 x … x Mn** en un minimum d’opérations de multiplications, c’est-à-dire trouver le parenthésage adéquat.

On note **mi,j** le nombre minimum de multiplications scalaire pour le calcul de **Mi x … x Mj**

Supposons que la meilleur façon de parenthéser **Mi x … x Mj** soit (**Mi x … x Mk) x (Mk+1 x … x Mj)**

(**Mi x … x Mk)** est une matrice de taille **di-1 x dk** et nécessite **mi,k** multiplication scalaire.

**(Mk+1 x … x Mj)** est une matrice de taille **dk x dj** et nécessite **mk+1,j** multiplication scalaire. Le nombre total de multiplication scalaire est **mi,k + mk+1,j +**  **di-1 x dk x dj**

On a donc : **mi,j = min{ mi,k + mk+1,j +**  **di-1 x dk x dj }**

Algorithme Chaine\_matrice

Debut

Pour i de 1 à n faire

M[i,i] ← 0

Fpour

Pour l de 2 à n faire

Pour i de 1 à n-l+1 faire

J ← i+l-1

M[i,j] ← ∞

Pour k de i à j-1 faire

q ← M[i,k] + M[k+1,j] + di-1\*dk\*dj

Si q < M[i,j] alors

M[i,j] ← q

S[i,j] ← k

Fsi

Fpour

Fpour

Fpour

Fin

* **Alignement global de séquences**

C’est un problème utilisé en bio-informatique surtout dans la recherche de séquence génétique. Il se pose comme suit : On dispose de deux séquences de caractères, et l’on cherche à les aligner de manière à maximiser la similarité.

Algorithme Chaine\_matrice

Debut

S[0,0] ← 0

Pour i de 1 à n faire

S[i,0] ← S[i-1,0] + gap

Fpour

Pour j de 1 à m faire

S[0,j] ← S[0, j-1] + gap

Fpour

Pour i de 1 à n faire

Pour j de 1 à m faire

Si S[i-1, j-1]>=S[i-1,j] ou S[i-1, j-1]>=S[i,j-1] alors

Si A[i] = B[j] alors

S[i,j] ← S[i-1, j-1] + match

Sinon

S[i,j] ← S[i-1, j-1] + mismatch

Fsi

Ligne[i,j] ← i-1

Colonne[i,j] ← j-1

Sinon

Si S[i-1,j] >= S[i,j-1] alors

S[i,j] ← S[i-1,j] + gap

Ligne[i,j] ← i-1

Colonne[i,j] ← j

Sinon

S[i,j] ← S[i,j-1] + gap

Ligne[i,j] ← i

Colonne[i,j] ← j-1

Fsi

Fsi

Fpour

Fpour

Fin