SOMMAIRE

INTRODUCTION

L'écriture des algorithmes peut se faire suivant plusieurs paradigmes parmi lesquels **l'approche gloutonne**, qui est une approche qui vise à faire à chaque étape le choix localement meilleur. Elle donne rapidement des résultats qui ne sont pas toujours optimaux. Plusieurs algorithmes peuvent être implémentés avec cette approche à l'instar du **problème du rendu de la monnaie** qui fera l'objet de ce travail. Tout d'abord, nous décrirons ce que c'est que le problème du rendu de monnaie, puis nous proposerons un algorithme de résolution suivi de son implémentation en C++ et des résultats obtenus après plusieurs tests.

1. Description et typologie du problème

Étant donné un système de monnaie (pièces et billets), on veut rendre une somme donnée de façon optimale, c'est-à-dire avec le nombre minimal de pièces (billets considérés comme pièces) tout en supposant que l'on a autant de pièces que possible pour chaque valeur. Le système de monnaie peut être identifié à une liste de n nombres qui représente la valeur de chaque pièce du système, par exemple **[25, 50, 100, 500, 1000, 2000, 5000, 10000]** pour la zone CEMAC. On souhaite écrire une fonction **rendu()** prenant en argument la liste des valeurs de pièces de monnaie et un entier **S**, correspondant à la somme d'argent à rendre, et renvoyant un tableau de longueur **n** indiquant le nombre de pièces à rendre pour chaque pièce du système, par exemple : **rendu([25, 50, 100], 150)** renverra **[0, 1, 1]** pour rendre **150** avec **[25, 50, 100]** comme système de pièces, qui signifie **0** pièce de 25, **1** piece de 50 et **1** piece de 100.

En pratique, sans s'en rendre compte généralement, tout individu met en œuvre un algorithme glouton. Il choisit d'abord la plus grande valeur de pièces autant que possible ainsi de suite jusqu'à la pièce de valeur minimale.

2. Quelques applications dans le monde réel du problème

Le problème de rendu de monnaie est un problème très connu dans le monde du commerce qui vise à optimiser le nombre de pièces à rendre pour un remboursement. A l'exemple du :

- Retour d'argent dans un guichet automatique
- Remboursement de la clientèle dans un supermarché

Bien que cet algorithme soit optimal lorsque le système de monnaie est canonique (l'ensemble de pièces permet de faire une décomposition minimale), il peut aussi s'appliquer à d'autres domaines qui se formalisent de la même manière que le rendu de monnaie. Un exemple concret serait :
On a un volume de terre V qu'on veut déplacer pour un chantier donné, on dispose de plusieurs types de camions, ceux de type 1, (contenance v1), type 2 (contenance v2), type 3 (contenance v3), etc…
On veut minimiser le nombre de camions à utiliser pour transporter la terre. Certes, le système dans lequel on se trouve n'est pas toujours canonique, mais ceci est un exemple d'extension de l'algorithme de rendu de monnaie pour un problème quotidien.

3. Formalisation du problème

On dispose d'un système de pièces **S=(C1, C2, ..., Cn)** tel que : **C1 > C2 > ... > Cn >= 1** et un entier positif **V**. Chaque pièce a une valeur **Ci** avec i **€ 1, 2, ..., n** ; le problème de rendu de monnaie est un problème d'optimisation combinatoire qui consiste à trouver un n-uplet d'entiers positifs **T=(X1, X2, ..., Xn)** qui minimise le nombre de pièces à rendre, soit :

$$\begin{cases} \sum_{i=1}^{n} x_i \\ s.c \\ \sum_{i=1}^{n} x_i c_i = v \end{cases}$$

où - **V** représente la somme de monnaie à rembourser.
- **Xi** le nombre de pièces utilisées pour rembourser.

La quantité à minimiser est le nombre total de pièces rendues, la condition à vérifier est qu'il faut rendre la somme **V**. On note MS(V) le nombre minimal de pièces d'un système S(V).

4. Proposition d'un algorithme

L'algorithme du rendu de monnaie varie en fonction du paradigme utilisé, dans notre cas il s'agit de l'approche gloutonne qui suit le principe suivant :

- Trier les pièces d'argent par ordre décroissant de valeurs.
- Pour chaque pièce on cherche le nombre maximum de pièces pouvant servir à rembourser la somme.

L'algorithme est donc le suivant:

Algorithme : Rendu De Monnaie

Entrées : **V** : La somme à rembourser
T: Le tableau des pièces disponibles

Sortie: **S** : Le tableau contenant pour chaque pièce le nombre de fois qu'elle a été utilisée

Variables: **i** : entier

Début:

```
Trier T dans l'ordre décroissant de valeurs des pièces
i = 1
Tant que V > 0 et i <= n faire
        S[i] ← V / T[i]                    //division entière
        V ← V - S[i] * T[i]
        i ← i + 1
Fintantque
Fin
```

5. Evaluation détaillée de la complexité de l'algorithme

Calculer la complexité d'un algorithme c'est mesurer le temps et l'espace mémoire nécessaire pour son exécution indépendamment de son implémentation.
En ce qui concerne notre algorithme, la complexité en temps se calcule comme suit:

- Le tri en complexité O (n (log n))
- La condition tantque :
  $T(n) = \Sigma(Tc + Ta) + Tc = \Sigma(2 + 3) + 1$
  $T(n) = 5n + 1$
- on a donc T(n) = O (n (log n) + 5n +1)

On en conclut que **T(n) € O (n (log n))**.

La complexité en espace est la taille de la mémoire utilisée par l'algorithme. Ainsi toutes les entrées, les variables et les sorties sont nécessaires pour ce calcul.

- V de taille 1
- T de taille n
- S de taille n
- i de taille 1

La complexité en espace est donc T(n) = 2n+2, **T(n) € O(2n + 2)**

6. Implémentation en C++

Il s'agit simplement d'une implémentation en C++ de l'algorithme proposé en **4. :**

```
vector<int> MinNumCoins(int somme, int pieces[], int n)
{
    sort(pieces, pieces + n, greater<>());

    vector<int> result(n);

    int i = 0;

    while (somme > 0 & i<n) {

        result[i] = somme / pieces[i];

        somme = somme - result[i] * pieces[i];

        i = i+1;
    }

    return result;
}
```

Suivi de l'appel de la fonction dans la fonction principale **main() :**

1 - L'utilisateur entre le nombre de pièces de son système

2 - L'utilisateur entre chacune des valeurs des pièces de son système

3 - L'utilisateur entre la valeur de la somme à rembourser

Ces valeurs récupérées sont passées en paramètres à l'appel de la fonction, une fois la matrice de résultat retournée, toujours dans la fonction **main()** l'on affiche la solution plus lisiblement qu'un simple vecteur de nombres.

```
int main()
{
    cout << "Entrez le nombre de pieces disponibles : ";
    int n, s;
    cin >> n;
    int c[n];
    for (int i = 0; i < n; i++) {
        cout << "Entrez la valeur de la piece " << i+1 << " : ";
        cin >> c[i];
    }
    cout << "Entrez la somme à rembourser : ";
    cin >> s;

    int len_c = (sizeof(c) / sizeof(c[0]));
    sort(c, c + len_c);

    //Appel de la fonction
    vector<int> r(n);
    r = MinNumCoins(s, c, len_c);

    // Affichage du resultat
    for (int i = 0; i < n; i++){
        if (r[i] != 0)
        {
            cout << " " << c[i] << " (" << r[i] << " fois) \t";
        }
    }
    return 0;
}
```

7. Tableau d'expérimentations de l'implémentation

Le tableau qui suit renseigne sur un ensemble de résultats obtenus après exécution de l'algorithme sur plusieurs cas :

Ensemble pieces	somme à rembourser	Somme remboursée	Solution retournée	Optimale ?
25, 50, 100, 500, 1000, 2000, 5000, 10000	4550 F	4550	2000(2) + 500(1) + 50(1)	Oui
25, 50, 100, 500, 1000, 2000, 5000, 10000	8775 F	8775	5000(1) + 2000(1) + 1000(1) + 500(1) + 100(2) + 50(1) + 25(1)	Oui
25, 50, 100, 500, 1000, 2000, 5000, 10000	3175 F	3175	2000(1) + 1000(1) + 100(1) + 50(1) + 25(1)	Oui
1, 2, 5, 10, 20, 50, 100, 500, 1000	45 €	45 €	20(2) + 5(1)	Oui
1, 2, 5, 10, 20, 50, 100, 500, 1000	98 €	98 €	50 (1) + 20(2) + 5(1) + 2(1) + 1(1)	Oui
4,3,1	6	6	4(1) + 1(2)	Non car [3(2)] fais moins de pièces
2,5,10	66	65	10(6) + 5(1) **solution exacte = 10(6) + 2(3)**	Pas de solution (66#65)

Ces expérimentations permettent de mieux comprendre le fonctionnement de l'algorithme glouton sur le problème du **rendu de monnaie.** Il retourne pour certains cas, la solution optimale mais pour d'autres pas ; De plus, pour certains cas il ne retourne pas la solution (dernier cas par exemple).

CONCLUSION

Le problème de rendu de monnaie est un problème de la vie courante. Il était question au cours de ce devoir de le comprendre, puis de proposer une solution gloutonne pour sa résolution. Nous avons constaté après une série de tests que cet algorithme bien qu'étant rapide ne retourne pas toujours la solution optimale, et dans certains cas, ne retourne pas la solution attendue. Il serait peut être plus intéressant de penser à le résoudre avec une approche de **programmation dynamique.**