

# Analyse d'algorithmes – Dossier intermédiaire

Huffman – Round Robin – Coefficient binomial



Baptiste Thomas – Cédric Klodzinski – Kevin Valente  
ISIMA, F2, JANVIER 2022

## Table des matières

Table des figures.....	3
I – Codage de Huffman .....	4
A) Principe et intérêt.....	4
B) Langage et dépendance du code.....	6
C) Analyse du code .....	6
D) Test .....	8
II – Algorithme de Round Robin .....	9
A) Principe et intérêt.....	9
B) Dépendance externe du code .....	9
C) Analyse du code .....	9
D) Test .....	10
III – Coefficient Binomial .....	12
A) Principe et intérêt.....	12
B) Langage et dépendance du code.....	12
C) Analyse du code .....	12
D) Test .....	14

## Table des figures

Figure 1 : Initialisation de l'arbre par Huffman .....	4
Figure 2 : Étape intermédiaire du codage de Huffman .....	5
Figure 3 : Arbre final obtenu avec Huffman .....	5
Figure 4 : Fonctionnement de l'attribution des codes binaires avec Huffman .....	6
Figure 5 : Résultat d'exécution du programme avec la chaîne de l'exemple.....	8
Figure 6 : illustration du Round About de Round Robin .....	9

## I – Codage de Huffman

Cédric K. (Programme) & Kevin V. (analyse)

### A) Principe et intérêt

Le codage de Huffman est un algorithme de compression de données sans perte. Il permet, pour simplifier, de réduire une chaîne de caractères en un code beaucoup plus léger. Pour ce faire, on va associer à chaque caractère un code binaire. Afin de réduire la taille de la chaîne, on fait correspondre les nombres binaires les plus petits aux caractères les plus fréquents.

Ce codage est utilisé dans plusieurs cas, notamment pour la compression des fichiers en .zip.

Le résultat de l'algorithme se présente sous la forme d'un arbre, dont voici le principe de construction, au travers d'un exemple. Nous prendrons ici le mot « aaabésséddé ». La première étape consiste à déterminer les fréquences de chaque caractère. Ici, on obtient :

a	b	d	é	s
3	1	2	3	2

On trie alors ces éléments par fréquence dans une file de priorité. Il s'agit d'une file dont les éléments sont ordonnés en fonction d'une valeur qui leur est associée, et ce de manière croissante selon cette valeur. Ici, on a pour la suite a-é-s-b-d :

b	d	s	a	é
1	2	2	3	3

Chaque caractère représente un nœud du futur arbre. Le procédé consiste à créer un nœud vide auquel on attachera les deux lettres les moins fréquentes présentes dans la file (le moins fréquent des deux étant la feuille de gauche). Le nœud créé prend alors pour valeur la somme des deux fréquences de ses feuilles. Ici, on a donc :

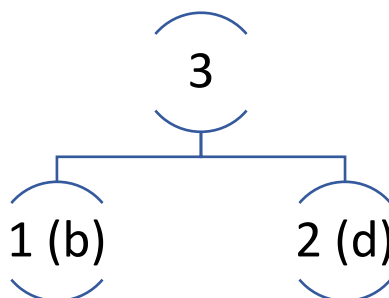


Figure 1 : Initialisation de l'arbre par Huffman

On peut alors défiler les deux lettres « b » et « d » de la file et ajouter le nouveau nœud, de valeur 3.

On continue alors la construction de l'arbre en suivant la même logique. Les deux nœuds à traiter sont donc le caractère « s » de fréquence 2 et le nœud de valeur 3 issu de la précédente itération. On obtient alors l'arbre suivant :

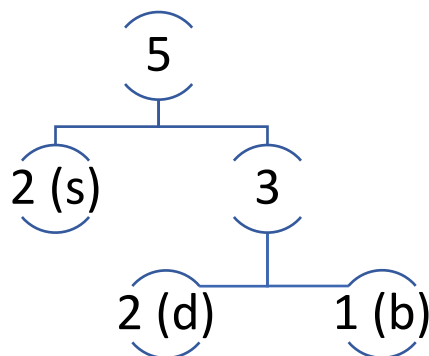


Figure 2 : Étape intermédiaire du codage de Huffman

Après cette deuxième itération, la file contient les valeurs suivantes :

a	é	Nœud vide
3	3	5

Pour la troisième itération, les deux nœuds à traiter sont alors les lettres « a » et « é », dont la somme des fréquences fait 6. L'arbre est alors le suivant :

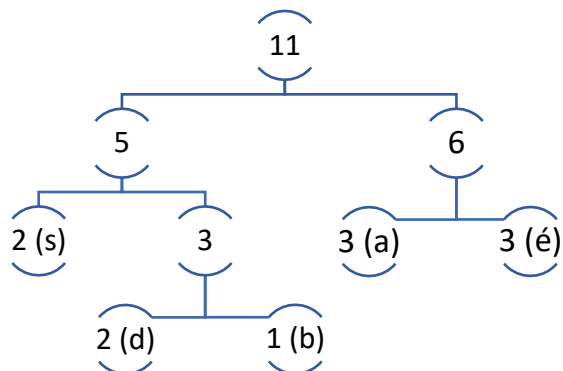


Figure 3 : Arbre final obtenu avec Huffman

La lecture de l'arbre se fait ensuite assez simplement. Chaque feuille de l'arbre représentant un caractère, il suffit de parcourir l'arbre depuis la racine jusqu'à une feuille pour établir le codage de ce caractère. Descendre à gauche signifie ajouter un 0 au code, tandis que descendre à droite correspond à ajouter un 1, comme le montre le schéma suivant :

s	a	é	d	b
00	10	11	010	011

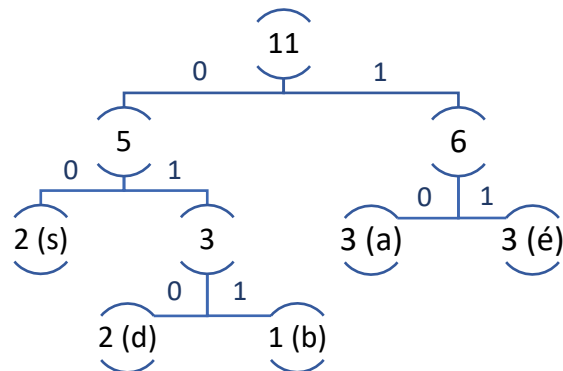


Figure 4 : Fonctionnement de l'attribution des codes binaires avec Huffman

Ici, en prenant par exemple le caractère « b », on voit bien qu'il nous faut descendre une fois à gauche, puis deux fois à droite. Le code binaire de « b » est donc 011.

## B) Langage et dépendance du code

Le programme fourni est écrit en C# avec le Framework .NET. Il respecte la charte de style préconisée par Microsoft. Le programme nécessite la version 6 de .NET, car les files de priorité (*priority queue*) ne sont disponibles qu'à partir de cette version du Framework.

## C) Analyse du code

**Type de programme** : application console

**Entrée** : une chaîne de caractères, tapée au clavier par l'utilisateur

**Sortie** : l'arbre du codage de Huffman. En cas de chaîne vide en entrée, une exception est levée.

Dans ce programme, nous utilisons une classe « HuffmanNode » relativement simple, qui représente un nœud de l'arbre et contient le caractère, sa fréquence et les deux nœuds enfants de droite et de gauche.

```
internal class HuffmanNode
{
    public char? Character { get; set; }
    public int Frequency { get; set; }
    public HuffmanNode? LeftNode { get; set; }
    public HuffmanNode? RightNode { get; set; }
}
```

La première étape du programme consiste en la lecture de la chaîne et en la gestion du cas null. Une exception est levée dans ce cas précis.

```
Console.WriteLine("Enter text to compress :");
string? toEnc = Console.ReadLine();

// Ensuring the entered string is not null
if (toEnc is null)
{
    throw new Exception("Null string");
}
```

Afin de calculer la fréquence de chaque caractère, nous créons une liste composée de tuples caractère-entier. On vérifie si un caractère est déjà apparu, si oui, on incrémente sa valeur, sinon on l'ajoute à la liste.

```
// List of character frequencies
List<(char, int)> frequencies = new List<(char, int)>();

// For each character in entered string, count occurrences and add to frequencies
foreach (char character in toEnc)
{
    int found = frequencies.FindIndex(pair => pair.Item1.Equals(character));

    if (found != -1)
    {
        frequencies[found] = (frequencies[found].Item1,
                                frequencies[found].Item2 + 1);
    }
    else
    {
        frequencies.Add((character, 1));
    }
}
```

On peut alors créer notre PriorityQueue à partir de la liste des fréquences :

```
// Character priority queue
PriorityQueue<HuffmanNode, int> priorities = new PriorityQueue<HuffmanNode, int>();

// For each frequency, create a new node in Huffman tree
foreach ((char, int) value in frequencies)
{
    HuffmanNode node = new HuffmanNode() { Character = value.Item1,
        Frequency = value.Item2, LeftNode = null, RightNode = null };

    priorities.Enqueue(node, value.Item2);
}
```

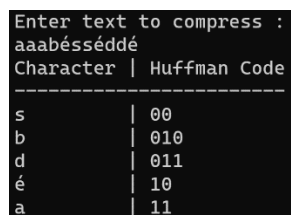
Il ne reste plus qu'à remonter l'arbre depuis la racine :

```
// While there are nodes in priorities, dequeue 2 nodes and store them as new
// node children (new root), then enqueue this new node
while (priorities.Count > 1)
{
    HuffmanNode left = priorities.Dequeue();
    HuffmanNode right = priorities.Dequeue();
    HuffmanNode nNode = new HuffmanNode() { Character = null, Frequency =
    left.Frequency + right.Frequency, LeftNode = left, RightNode = right };

    root = nNode;
    priorities.Enqueue(nNode, nNode.Frequency);
}
```

#### D) Test

Concernant ce programme, les tests sont assez simples. Un premier consiste seulement à vérifier le bon fonctionnement de l'exception. Un second consiste à tester les résultats avec la chaîne donnée précédemment en exemple, afin de vérifier que le résultat est correct.



```
Enter text to compress :
aaabésséddé
Character | Huffman Code
-----|-----
s         | 00
b         | 010
d         | 011
é         | 10
a         | 11
```

Figure 5 : Résultat d'exécution du programme avec la chaîne de l'exemple

Les résultats sont conformes à l'exemple plus haut. Certains caractères, comme « é » et « a », ont leur code échangé mais cela s'explique par leur fréquence d'apparition identique : l'ordre de traitement en cas d'égalité des fréquences peut différer entre les exécutions. Cela n'a cependant pas d'impact sur la taille du code généré. La taille d'une chaîne compressée ne change donc pas.

On notera également que ce test a également permis de tester un caractère spécial, le « é ».



## II – Algorithme de Round Robin

Kevin V. (Programme) & Baptiste T. (analyse)

### A) Principe et intérêt

L'algorithme Round-Robin permet de distribuer du temps de calcul entre plusieurs processus dans un système d'exploitation. Il s'agit donc d'un algorithme d'ordonnancement.

Dans le cas de Round-Robin, on définit un quantum de temps. Les processus sont placés dans une file et exécutés chacun son tour, et ce pendant le quantum de temps au maximum.

On utilise le principe du tourniquet, où un seul processus passe en phase de calcul pour un quantum de temps. De plus, d'autres règles entrent en vigueur pour assurer le bon fonctionnement de l'algorithme. Ainsi, pour éviter la famine, les nouveaux processus sont ajoutés en fin de file. De plus, un processus ne peut pas dépasser le quantum de temps en phase de calcul.

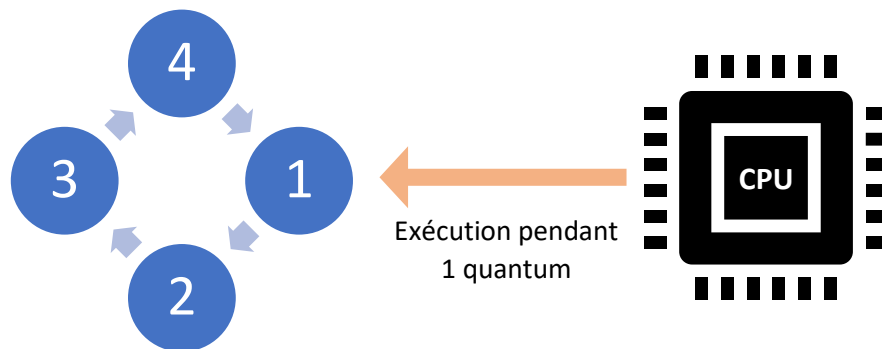


Figure 6 : illustration du Round About de Round Robin

### B) Dépendance externe du code

Le programme fourni est écrit en C# avec le Framework .NET. Il respecte la charte de style préconisée par Microsoft. Le programme nécessite la version 6 de .NET, car des changements de structure empêchent la rétrocompatibilité.

### C) Analyse du code

**Type de programme** : application console

**Entrée** : un tableau de processus et des temps d'exécution correspondant à entrer dans le code.

**Sortie** : Plusieurs données relatives au temps d'attente et d'exécution.

On place, pour chaque processus, un numéro dans un tableau et le temps d'exécution correspondant dans un autre tableau (même index). On indique le nombre total de processus ainsi que le quantum de temps choisi.

Le programme est ensuite composé de 2 fonctions. La première, « AverageTime », permet le calcul de différentes données, comme le temps d'attente moyen des processus avant leur première exécution, ainsi que le temps avant la fin de leur exécution. C'est ici aussi que l'on lance la fonction d'ordonnancement.

```

public void AverageTime(int[] processes, int n, int[] burstTime, int quantum)
{
    int tmp, totalWaiting = 0, totalTurnAround = 0;
    // Time before the end of execution
    int[] turnAroundTime = new int[n];
    int[] waitingTime = new int[n];

    WaitingTime(processes, n, burstTime, waitingTime, quantum);

    // we process the turnaround time and the waiting cumuled time
    for (int i = 0 ; i < n ; i++)
    {
        // we gather the waiting time
        tmp = waitingTime[i];
        turnAroundTime[i] = burstTime[i] + tmp;
        totalWaiting = totalWaiting + tmp; //temps cumulé d'attente
        totalTurnAround = totalTurnAround + turnAroundTime[i];
    }
}

```

La deuxième fonction, « **WaitingTime** », est la plus intéressante. Elle gère la distribution du temps de calcul.

On boucle tant que les processus ne sont pas terminés.

Dans cette boucle, on parcourt toute la liste des processus, en mettant à jour le temps de calcul restant pour ceux qui ne sont pas terminés. Pour mettre à jour le temps, on somme simplement le quantum de temps au temps total et au temps d'exécution restant. Une fois le processus terminé, on calcule le temps d'attente à partir du temps actuel et du temps d'exécution total.

## D) Test

Lors de la phase de test, une redondance de code a été supprimée et donc le code a été simplifié. Ceci m'a également permis de corriger une erreur qui faisait arrêter le programme dès qu'un processus était fini.

En termes de test, une comparaison entre les résultats du programme et des résultats obtenus à la main a été effectuée. Afin d'afficher les résultats, le code suivant est utilisé :

```

public void WaitingTime(int[] processes, int n, int[] burstTime, int[] waitingTime, int
quantum)
{
    // updating exécution time
    int[] remainingBurstTime = new int[n];
    for (int i = 0; i < n; i++)
    {
        remainingBurstTime[i] = burstTime[i];
    }
    // Actual time and temporary variable
    int t = 0 , tmp;
    bool loop = true;

    // while they are processes
    while (loop)
    {
        // for each process
        for (int i = 0; i < n; i++)
        {
            tmp = remainingBurstTime[i];
            //if not ended
            if (tmp > 0)
            {
                // + time process
                loop = true ;
            }
            else
            {
                loop = false ;
            }
        }
    }
}

```

Sur plusieurs essais, les résultats étaient conformes.

### III – Coefficient Binomial

Baptiste T. (Programme) & Cédric K. (analyse)

#### A) Principe et intérêt

Les coefficients binomiaux sont très utilisés en mathématiques. On peut, par exemple, citer les lois binomiales de probabilités ou encore le dénombrement. Un coefficient binomial se traduit couramment par «  $k$  parmi  $n$  » et donne le nombre de parties de  $k$  éléments dans un ensemble de  $n$  éléments. La formule pour le calculer et sa notation sont les suivantes :

$$\binom{n}{k} = C_n^k = \frac{n!}{k! (n - k)!}$$

Dans le programme que nous présentons, nous verrons que nous avons quelque peu manipulé cette formule.

#### B) Langage et dépendance du code

Le programme fourni est écrit en C#. Pour se prémunir des problèmes de dépassement de capacité, on utilise des types *ulong*, soit des entiers non signés sur 64 bits. On pourra quand même noter que dans le cas où la capacité est dépassée, les valeurs sont fausses, mais le programme ne produit pas d'erreurs.

#### C) Analyse du code

**Type de programme** : application console

**Entrée** : deux entiers.

**Sortie** : un entier correspondant au coefficient binomial.

Le code est composé de deux fonctions :

- **Factorial**, qui calcule la factorielle  $x!$  d'un entier  $x$  via une boucle réalisant la multiplication de 1 à  $n$ . Ainsi, on a  $o(x)$  opérations.

```
ulong factorial(int x)
{
    ulong result = 1;
    while (x > 1)
    {
        result *= (ulong)x;
        x--;
    }

    return result;
}
```

- **CoefBinomial**, qui calcule le coefficient binomial pour deux entiers  $n$  et  $k$ . On utilise la formule  $\frac{n!}{k!(n-k)!}$  dans le cas  $k \leq n$ , la complexité est donc de  $o(2n)$ . Dans le cas où  $k > n$ , la fonction retourne 0 en  $o(1)$ .

```
ulong CoefBinomial(int n, int k)
{
    if (n >= 0 && k >= 0 && k <= n)
        return factorial(n) / (factorial(k) * factorial(n - k));
    else
        return 0;
}
```

Des versions optimisées sont aussi disponibles :

- **FactorialPQ**, qui calcule la factorielle entre deux entiers  $p$  et  $q$  via une boucle réalisant la multiplication de  $p$  à  $q$ . Le résultat est équivalent à  $\frac{q!}{(p-1)!}$  et permet de simplifier le calcul du coefficient. On a  $o(q - p)$  opérations ici.

```
ulong factorialPQ(int p, int q)
{
    if (p > q || p == 0)
        throw new Exception();

    ulong result = (ulong)p;
    while (q > p)
    {
        result *= (ulong)q;
        q--;
    }

    return result;
}
```

- **CoefBinomialOpti**, qui calcule le coefficient binomial pour deux entiers  $n$  et  $k$ . On utilise la formule  $\frac{n!}{k!}$ , dans le cas  $k < n$ , avec  $\frac{n!}{k!}$  calculé via **FactorialPQ**. La complexité est donc de  $o(2(n - k))$ . Dans le cas où  $k = n$ , on retourne 1 en  $o(1)$  et pour  $k > n$ , on retourne 0 en  $o(1)$  également.

```
ulong CoefBinomialOpti(int n, int k)
{
    if (n >= 0 && k >= 0)
    {
        if (k < n)
            return factorialPQ(k + 1, n) / factorialPQ(n - k);
        else if (k == n)
            return 1;
        else
            return 0;
    }
    else
        return 0;
}
```

## D) Test

Pour tester l'algorithme, nous allons utiliser les cas suivants :

- $n = 0, k = 0$  : le coefficient vaut 1, comme attendu
- $n < k$  : pour  $n = 10$  et  $k = 20$ , on a un coefficient de 0, comme attendu
- $n = k$  : pour  $n = k = 10$ , on a un coefficient de 1, comme attendu
- $n > k$  : pour  $n = 20$  et  $k = 10$ , on a un coefficient de 184756, comme attendu
- $n$  ou  $k$  négatif : pour  $n=-1$  et  $k$  positif,  $n$  positif et  $k=-1$  et  $n=k=-1$ , on a un coefficient de 0, comme attendu