

Objectifs

- À la fin de cette séance, vous devriez être capable de :
- appliquer l’algorithme de Huffman sur un exemple simple ;
 - évaluer la qualité d’un code compresseur ;
 - utiliser la méthode *Diviser pour Régner* pour résoudre un problème.

Exercice 1 : Huffman (30 min)

On dispose d’un texte écrit avec un alphabet $\mathcal{A} = \{A, B, C, D, E, F, G, H\}$. Les proportions de ces symboles dans le texte sont données en pourcentage dans le tableau suivant :

Symbole	A	B	C	D	E	F	G	H
Proportion (%)	2	10	4	18	12	16	32	6

1. Codage de longueur fixe

Donnez la taille du codage de longueur fixe nécessaire pour coder cet alphabet \mathcal{A} .
 Calculez l’entropie $\mathcal{H}(p) = -\sum_{i=1}^k p_i \log_2 p_i$ de cette distribution.
 Pourquoi le code de longueur fixe n’est-il (probablement) pas le codage le plus court en moyenne pour cet alphabet ?

2. Codage de longueur variable

Construire avec l’algorithme de Huffman un codage de longueur variable pour \mathcal{A} . Donner l’arbre de codage correspondant et le tableau des codes. Calculer la longueur moyenne du codage et commenter le résultat en une phrase.

3. Transmission de message

À faire par deux : simulez une transmission de message à l’aide de votre code de Huffman.
 — L’un d’entre vous joue le rôle d’émetteur : il choisit un mot de 5 à 10 lettres sur l’alphabet \mathcal{A} et le code.
 — L’autre joue le rôle du récepteur : il décode le message binaire reçu.
 Vérifiez que tout s’est passé comme prévu. Sinon, expliquez pourquoi et résolvez le problème.
 Échangez les rôles et recommencez.

Correction de l’exercice 1

1. L’alphabet \mathcal{A} possède 8 lettres, il faut donc au moins $\log_2(8) = 3$ bits pour coder ces 8 symboles (code de longueur fixe). On calcule l’entropie associée à cette distribution de probabilité.

<i>Symbole</i>	<i>%</i>	<i>Proba</i>	$-\log_2(p_i)$	$-\log_2(p_i)p_i$
A	2	0,02	5,64	0,11
B	10	0,10	3,32	0,33
C	4	0,04	4,64	0,18
D	18	0,18	2,47	0,44
E	12	0,12	3,05	0,36
F	16	0,16	2,64	0,42
G	32	0,32	1,64	0,52
H	6	0,06	4,05	0,24
<i>s =</i>	100		$H(p) =$	2,63

On trouve que $H(p) = 2,63$ c’est à dire qu’il faut au minimum et en moyenne 2.63 bits pour coder un symbole de l’alphabet alors qu’il en faudrait 3 pour un code de longueur fixe. Il est donc probable qu’on y gagnera à avoir un code de longueur variable qui compressera le texte.

2. Après avoir fait tourner Huffman (on peut avoir plusieurs solutions) on obtient

<i>Symbole</i>	<i>%</i>	<i>Proba</i>	<i>code</i>	<i>longueur</i>	<i>longueur theorique</i>
<i>A</i>	2	0,02	01000	5	5,64
<i>B</i>	10	0,10	010	3	3,32
<i>C</i>	4	0,04	01001	5	4,64
<i>D</i>	18	0,18	00	2	2,47
<i>E</i>	12	0,12	100	3	3,05
<i>F</i>	16	0,16	101	3	2,64
<i>G</i>	32	0,32	11	2	1,64
<i>H</i>	6	0,06	0101	4	4,05
<i>s =</i>	100		<i>L(h) =</i>	2,68	<i>H(p) = 2,63</i>

On retrouve bien $H(p) \leq L(h) \leq H(p) + 1$. Ici l'algorithme produit un résultat très proche de la valeur minimale théorique.

3. À faire en binôme.
4. Il suffit de parcourir l'arbre, de mémoriser les chemins parcourus sous forme d'une suite de bits, et d'afficher ces suites au niveau des feuilles. La mémorisation du chemin de la racine jusqu'au noeud courant se fait au moyen d'un argument supplémentaire *s*.

Soit `TableCodes` le tableau à construire, indexé par les symboles.

```

CalculerCodes(A : Arbre de Huffman, s : séquence de bits)
si A est une feuille alors
    TableCodes(symbole de A) <- s
sinon
    CalculerCodes(fils gauche de A, s.0)
    CalculerCodes(fils droit de A, s.1)

```

Exercice 2 : Majoritaire

Définition : un élément x est dit *majoritaire* dans un multi-ensemble E de n éléments, si et seulement si le nombre d'occurrences de x dans E est strictement supérieur à $n/2$.

Dans la suite E est un tableau indexé de 0 à $n - 1$.

1. Algorithme naïf

```

existe_majoritaire <- faux
pour i de 0 à n-1
  c := 0
  j := 0
  tant que j < n et non(existe_majoritaire) faire
    si E[j]=E[i] alors c := c+1
    si (c > n/2) alors
      existe_majoritaire := vrai
      majo := E[i]
    j := j+1

```

Décrivez (en deux lignes maximum) son fonctionnement.

Quel est le coût au mieux et au pire de cet algorithme ?

2. Diviser pour régner

On remarque les propriétés suivantes (que l'on admet pour l'instant).

Si $E = E_1 \cup E_2$ avec E_1 et E_2 de mêmes tailles à un élément près (dans le cas où E comporte un nombre impair d'éléments), alors :

- si x est majoritaire dans E , alors x est majoritaire dans E_1 ou dans E_2 .
- si x est majoritaire dans E_1 **et** dans E_2 , alors il est majoritaire dans E .

En utilisant ces propriétés, écrivez une procédure `majoritaire(i, j)` qui explore récursivement les deux moitiés de tableau, et cherche s'il existe un élément majoritaire dans $E[i..j]$. Elle renvoie :

- $(-, 0)$ s'il n'y a pas de majoritaire dans $E[i..j]$.
- (x, c) si x est majoritaire avec un nombre d'occurrences égal à c .

3. Complexité

Déterminez la complexité de cette procédure récursive, en vous inspirant de l'algorithme vu en cours.

4. Vers un algorithme récursif

Démontrer les propriétés suivantes pour justifier la correction de l'algorithme récursif :

- il y a **au plus** un élément majoritaire.

Si $E = E_1 \cup E_2$ avec E_1 et E_2 de taille n' (si $n = 2 \times n'$) ou de tailles n' et $n' + 1$ si $n = 2 \times n' + 1$, alors :

- si x est majoritaire dans E , alors x est majoritaire dans E_1 ou dans E_2 .
- si x est majoritaire dans E_1 et dans E_2 , alors il est majoritaire dans E .

Correction de l'exercice 2

1. — Au minimum, la boucle **tant que** s'exécutera $n/2 + 1$ fois (le temps que c dépasse $n/2$), et dans le meilleur des cas cela se produit pour $i = 1$ et $j = n/2 + 1$. Les itérations suivantes de la boucle **for** sont alors triviales et ne comparent plus d'éléments de E .
Le coût au mieux de l'algorithme est donc de $n/2$ comparaisons entre éléments de E , plus $\mathcal{O}(n)$ opérations sur les indices ; il correspond au cas où le premier élément du tableau est majoritaire et où ses occurrences sont tassées à gauche.
- Au pire, chaque occurrence de la boucle **tant que** subit un maximum d'itérations, ce qui se produit si j va de 0 à $n - 1$ systématiquement. Le coût est alors clairement de n^2 comparaisons (et autant d'opérations sur les indices).
Pour qu'un tel cas se produise, il faut que c ne dépasse jamais $n/2$, autrement dit qu'aucun élément ne soit majoritaire. Un exemple de pire cas serait un tableau dont tous les éléments sont distincts.

2. — Par l'absurde : s'il y avait deux éléments majoritaires distincts, ils occuperaient à eux deux $2(\lfloor n/2 \rfloor + 1)$ emplacements dans E , ce qui dépasse n .
 - Par l'absurde encore : si x est minoritaire à la fois dans E_1 et dans E_2 , alors il est présent au maximum $2\lfloor n'/2 \rfloor + 1$ dans E , et x est donc minoritaire dans E également.
 - si x est majoritaire dans E_1 et dans E_2 , alors il est présent au moins $2(\lfloor n'/2 \rfloor + 1) \geq \lfloor n/2 \rfloor + 1$ donc il est majoritaire dans E aussi.
3. On utilise une fonction $\text{occ}(x, i, j)$ qui compte le nombre d'occurrences de x dans $E[i..j]$ en comparant x à chacun des éléments.

```

fonction majoritaire(i, j)
  si i=j alors renvoyer (E[i], 1)
  sinon
    milieu := floor((i+j)/2)
    taille := j-i+1
    (x, cx) := majoritaire(i, milieu)
    (y, cy) := majoritaire(milieu+1, j)
    si x=y alors renvoyer (x, cx+cy)
    si cx <> 0 alors cx := cx + occ(x, milieu+1, j)
    si cy <> 0 alors cy := cy + occ(y, i, milieu)
    si cx > taille/2 alors renvoyer (x, cx)
    sinon si cy > taille/2 alors renvoyer (y, cy)
    sinon renvoyer (-, 0)

```

Dans les cas favorables (par exemple $E=[a, b, a, b, a, b \dots]$) l'appel $\text{majoritaire}(0, n-1)$ effectue n comparaisons.

Dans les cas défavorables, (par exemple $E=[n/2 \text{ a puis } n/4 \text{ b etc}]$) soit $C(n)$ le nombre de comparaisons pour un tableau de taille n :

$$- C(n) = 2 \times C(n/2) + n/2$$

$$- C(1) = 0$$

donc $C(n)$ est de l'ordre de $n \times \log_2(n)$.

4. On peut améliorer encore (un peu) l'algorithme en remarquant que :
 - si une seule des "moitiés" fournit un majoritaire, seul cet élément peut être majoritaire dans le tableau complet.
 - si les deux appels récursifs fournissent des majoritaires, qu'ils sont différents, avec un nombre d'occurrences différent : le seul qui puisse être majoritaire est celui qui a la plus grand nombre d'occurrences.
 - dans le cas où n est une puissance de 2, et donc les moitiés toujours de taille égale, s'il y a deux majoritaires différents avec le même nombre d'occurrences, alors il ne peut y avoir de majoritaire dans le tableau complet.

```

fonction majoritaire(i, j)
  si i=j alors renvoyer (E[i], 1)
  sinon
    milieu := floor((i+j)/2)
    taille := j-i+1
    (x, cx) := majoritaire(i, milieu)
    (y, cy) := majoritaire(milieu+1, j)
    si cx=0 et cy=0 alors renvoyer (-, 0)
    si cx=0 alors
      cy := cy + occ(y, i, milieu)
      si cy > taille/2 alors renvoyer (y, cy)
      sinon renvoyer (-, 0)
    si cy=0 alors
      cx := cx + occ(x, milieu+1, j)
      si cx > taille/2 alors renvoyer (x, cx)
      sinon renvoyer (-, 0)

```

```
// cx<>0, cy <>0
si x=y alors renvoyer (x, cx+cy)
sinon
  si cx=cy alors renvoyer (-, 0) // cas n puissance de 2
  sinon
    si cx<cy alors
      cy := cy + occ(y, i, milieu)
      si cy > taille/2 alors renvoyer (y, cy)
      sinon renvoyer (-, 0)
    sinon
      cx := cx + occ(x, milieu+1, j)
      si cx > taille/2 alors renvoyer (x, cx)
      sinon renvoyer (-, 0)
```

Dans les cas favorables, cet algo effectue $n/2$ comparaisons (par exemple pour $E=[a, b, a, b, a, b, a, b, \dots]$). Donner la complexité exacte dans les cas défavorables (i.e. exhiber un cas défavorable) semble difficile.

Une majoration du coût maximum est en $O(n \times \log_2(n))$, mais ce maximum n'est jamais atteint. Il semble cependant que l'ordre de grandeur soit le bon.