

# Les données sont $D^1$

Étude de méthodes de compression sans pertes

Thomas BAGREL

Lycée Henri POINCARÉ, Nancy

TIPE session 2018

# Aperçu

## Régularités et gains

- Théorie

- ZIP récursif

## Composantes de la compression

### Codage

- Problème résolu

- Codage d'Huffman

- Codage arithmétique

- Inefficacité de HUFFMAN seul – pourquoi

### Modèles généraux

- BITWISE ENCODER et PPM

- BITWISE PPM et BITWISE PPM FLAT

### Impact de la transformée

- BWT régularise nos données

- La BWT en action

# Régularités et gains

## Théorie

Analogie : sources de données  $\leftrightarrow$  fonctions

# Régularités et gains

## Théorie

Analogie : sources de données  $\leftrightarrow$  fonctions

Compression des données sans pertes

# Régularités et gains

## Théorie

Analogie : sources de données  $\leftrightarrow$  fonctions

Compression des données sans pertes

- exploiter les régularités des données

# Régularités et gains

## Théorie

Analogie : sources de données  $\leftrightarrow$  fonctions

Compression des données sans pertes

- ▶ exploiter les régularités des données
- ▶ données aléatoires : pas de gain

# Régularités et gains

## Théorie

Analogie : sources de données  $\leftrightarrow$  fonctions

Compression des données sans pertes

- ▶ exploiter les régularités des données
- ▶ données aléatoires : pas de gain

Théorème. (Entropie de SHANNON)

$$H(S) = - \sum_{i=1}^n p_i \log_2(p_i)$$

$H(S)$  : nb. de bits moyen par symbole de la source

# Régularités et gains

## Théorie

Analogie : sources de données  $\leftrightarrow$  fonctions

Compression des données sans pertes

- ▶ exploiter les régularités des données
- ▶ données aléatoires : pas de gain

Théorème. (Entropie de SHANNON)

$$H(S) = - \sum_{i=1}^n p_i \log_2(p_i)$$

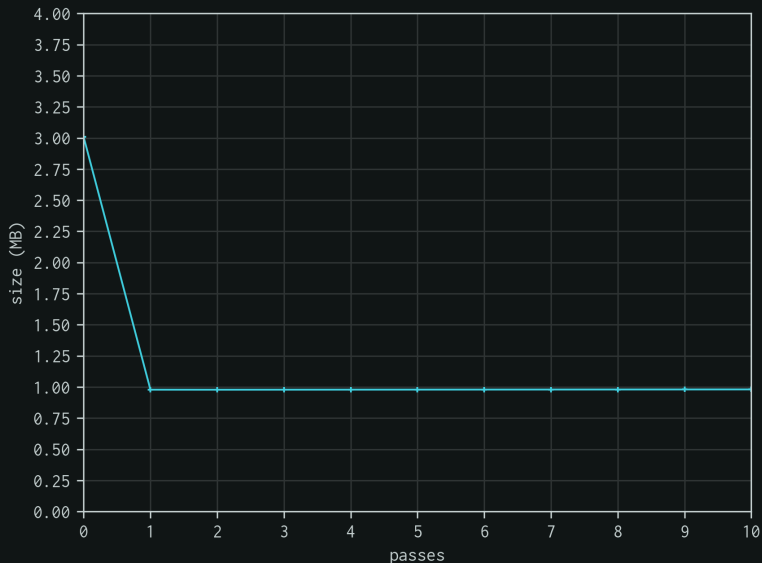
$H(S)$  : nb. de bits moyen par symbole de la source

- ▶ une fois les données compressées (dérivées) une fois, plus aucune régularité



# Régularités et gains

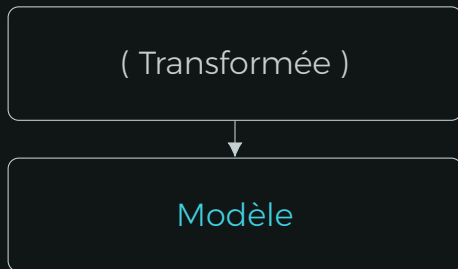
ZIP recursif



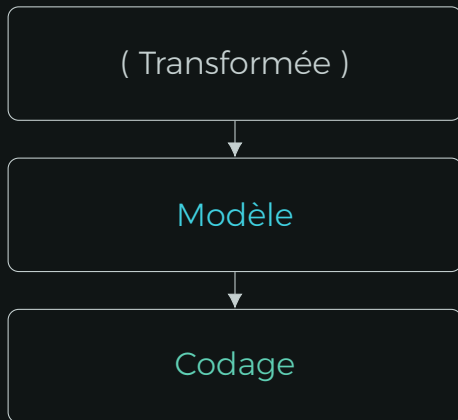
# Composantes de la compression

( Transformée )

# Composantes de la compression



# Composantes de la compression



# Codage

## Problème résolu

Le codage, contrairement aux apparences, est un problème **résolu**

# Codage

## Problème résolu

Le codage, contrairement aux apparences, est un problème **résolu**

- Si les  $p_i$  sont connus, la limite de compression théorique est donnée par SHANNON

# Codage

## Problème résolu

Le codage, contrairement aux apparences, est un problème **résolu**

- ▶ Si les  $p_i$  sont connus, la limite de compression théorique est donnée par SHANNON
- ▶ HUFFMAN permet d'**approcher** cette limite

# Codage

## Problème résolu

Le codage, contrairement aux apparences, est un problème **résolu**

- ▶ Si les  $p_i$  sont connus, la limite de compression théorique est donnée par SHANNON
- ▶ HUFFMAN permet d'**approcher** cette limite
- ▶ Le codage arithmétique l'**atteint**



# Codage

## Codage d'Huffman

Codage d'HUFFMAN pour turlututu

u | 4

l | 1

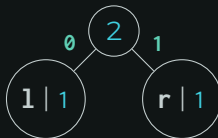
r | 1

t | 3

# Codage

## Codage d'Huffman

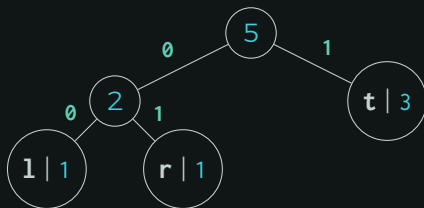
Codage d'HUFFMAN pour turlututu



# Codage

## Codage d'Huffman

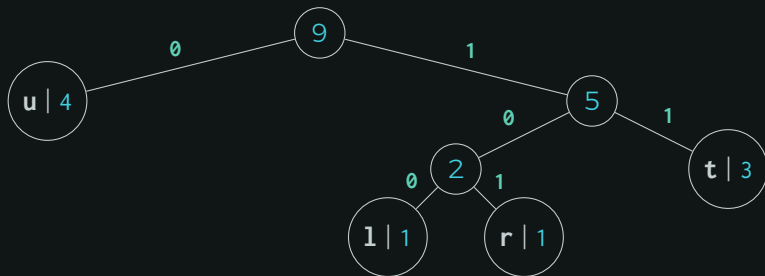
Codage d'HUFFMAN pour turlututu



# Codage

## Codage d'Huffman

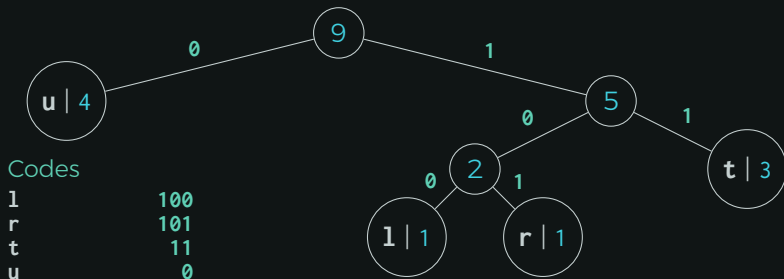
Codage d'HUFFMAN pour turlututu



# Codage

## Codage d'Huffman

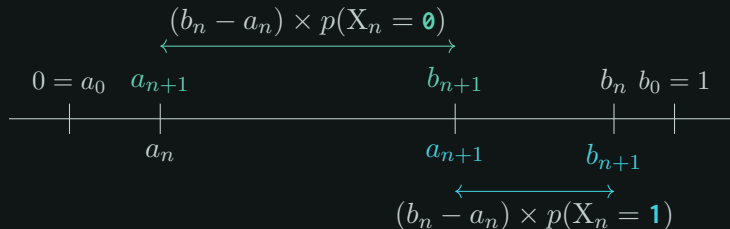
Codage d'HUFFMAN pour turlututu



# Codage

## Codage arithmétique

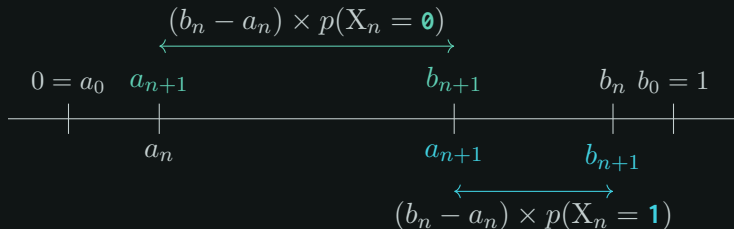
Codage arithmétique  $p(X_n = 0) = 0,65$



# Codage

## Codage arithmétique

Codage arithmétique  $p(X_n = 0) = 0,65$

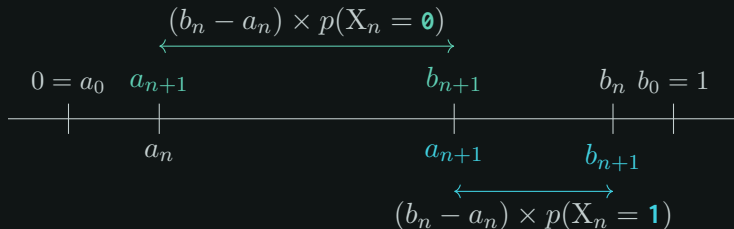


- Sous-intervalles proportionnels aux  $p_i$

# Codage

## Codage arithmétique

Codage arithmétique  $p(X_n = 0) = 0,65$



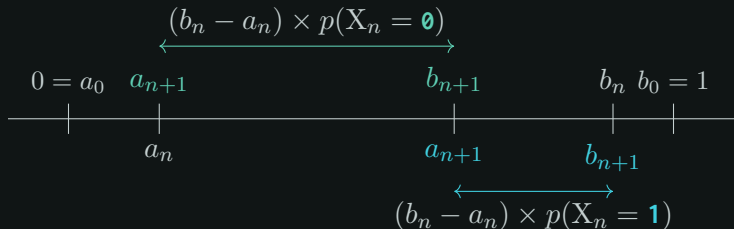
- ▶ Sous-intervalles proportionnels aux  $p_i$
- ▶ En pratique, avec des  $p_i$  non fixes, gérer plus de quelques symboles est compliqué



# Codage

## Codage arithmétique

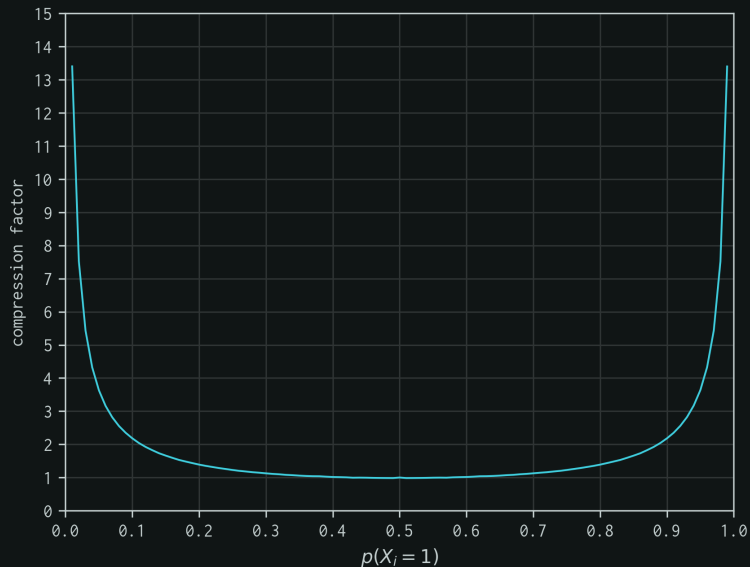
Codage arithmétique  $p(X_n = 0) = 0,65$



- ▶ Sous-intervalles proportionnels aux  $p_i$
- ▶ En pratique, avec des  $p_i$  non fixes, gérer plus de quelques symboles est compliqué
- ▶ Implémenté avec des entiers pour représenter  $[0; 1[$

# Codage

## Codage arithmétique



# Codage

Inefficacité de HUFFMAN seul – pourquoi

HUFFMAN seul

# Codage

Inefficacité de HUFFMAN seul – pourquoi

HUFFMAN seul

En pratique, efficacité de 32 %

# Codage

Inefficacité de HUFFMAN seul – pourquoi

HUFFMAN seul

En pratique, efficacité de 32 %

En appliquant directement SHANNON aux fréquences d'apparition, on commet des erreurs

# Codage

## Inefficacité de HUFFMAN seul – pourquoi

HUFFMAN seul

En pratique, efficacité de 32 %

En appliquant directement SHANNON aux fréquences d'apparition, on commet des erreurs

- ▶ un symbole n'est pas indépendant des précédents

# Codage

## Inefficacité de HUFFMAN seul – pourquoi

HUFFMAN seul

En pratique, efficacité de 32 %

En appliquant directement SHANNON aux fréquences d'apparition, on commet des erreurs

- ▶ un symbole n'est pas indépendant des précédents
- ▶ par exemple, en Français,  $q \rightarrow u$  est plus fréquent que  $q \rightarrow z$

# Codage

## Inefficacité de HUFFMAN seul – pourquoi

### HUFFMAN seul

En pratique, efficacité de 32 %

En appliquant directement SHANNON aux fréquences d'apparition, on commet des erreurs

- ▶ un symbole n'est pas indépendant des précédents
- ▶ par exemple, en Français,  $q \rightarrow u$  est plus fréquent que  $q \rightarrow z$
- ▶ en quelque sorte, on oublie le caractère lipschitzien de nos données



# Codage

## Inefficacité de HUFFMAN seul – pourquoi

### HUFFMAN seul

En pratique, efficacité de 32 %

En appliquant directement SHANNON aux fréquences d'apparition, on commet des erreurs

- ▶ un symbole n'est pas indépendant des précédents
- ▶ par exemple, en Français,  $q \rightarrow u$  est plus fréquent que  $q \rightarrow z$
- ▶ en quelque sorte, on oublie le caractère lipschitzien de nos données

Il faut donc un modèle

# Modèles généraux

BITWISE ENCODER et PPM

PPM

# Modèles généraux

## BITWISE ENCODER et PPM

### PPM

- ▶ Le contexte est constitué des  $N$  symboles précédents

# Modèles généraux

## BITWISE ENCODER et PPM

### PPM

- ▶ Le contexte est constitué des  $N$  symboles précédents
- ▶ Un jeu de  $p_i$  pour chaque contexte différent

# Modèles généraux

## BITWISE ENCODER et PPM

### PPM

- ▶ Le contexte est constitué des  $N$  symboles précédents
- ▶ Un jeu de  $p_i$  pour chaque contexte différent
- ▶ Si le contexte n'a jamais été rencontré, on retombe sur un contexte d'ordre  $N - 1$  (*order fallback*) etc.

# Modèles généraux

## BITWISE ENCODER et PPM

### PPM

- ▶ Le contexte est constitué des  $N$  symboles précédents
- ▶ Un jeu de  $p_i$  pour chaque contexte différent
- ▶ Si le contexte n'a jamais été rencontré, on retombe sur un contexte d'ordre  $N - 1$  (*order fallback*) etc.
- ▶ Codage arithmétique

# Modèles généraux

## BITWISE ENCODER et PPM

### PPM

- ▶ Le contexte est constitué des  $N$  symboles précédents
- ▶ Un jeu de  $p_i$  pour chaque contexte différent
- ▶ Si le contexte n'a jamais été rencontré, on retombe sur un contexte d'ordre  $N - 1$  (*order fallback*) etc.
- ▶ Codage arithmétique

### Inconvénients et solutions

# Modèles généraux

## BITWISE ENCODER et PPM

### PPM

- ▶ Le contexte est constitué des  $N$  symboles précédents
- ▶ Un jeu de  $p_i$  pour chaque contexte différent
- ▶ Si le contexte n'a jamais été rencontré, on retombe sur un contexte d'ordre  $N - 1$  (*order fallback*) etc.
- ▶ Codage arithmétique

### Inconvénients et solutions

- ▶ Encoder 256 symboles avec des  $p_i$  variables est dur à implémenter sans pertes



# Modèles généraux

## BITWISE ENCODER et PPM

### PPM

- ▶ Le contexte est constitué des  $N$  symboles précédents
- ▶ Un jeu de  $p_i$  pour chaque contexte différent
- ▶ Si le contexte n'a jamais été rencontré, on retombe sur un contexte d'ordre  $N - 1$  (*order fallback*) etc.
- ▶ Codage arithmétique

### Inconvénients et solutions

- ▶ Encoder 256 symboles avec des  $p_i$  variables est dur à implémenter sans pertes
- ▶ On pourrait donc encoder bit par bit au lieu de symbole par symbole

# Modèles généraux

BITWISE PPM et BITWISE PPM FLAT

BITWISE PPM

# Modèles généraux

BITWISE PPM et BITWISE PPM FLAT

BITWISE PPM

- ▶ Idée personnelle

# Modèles généraux

## BITWISE PPM et BITWISE PPM FLAT

### BITWISE PPM

- ▶ Idée personnelle
- ▶ Utiliser une PPM (d'ordre max. 3 octets) avec un encodage bit par bit

# Modèles généraux

## BITWISE PPM et BITWISE PPM FLAT

### BITWISE PPM

- ▶ Idée personnelle
- ▶ Utiliser une PPM (d'ordre max. 3 octets) avec un encodage bit par bit
- ▶ 49 % d'efficacité!

# Modèles généraux

## BITWISE PPM et BITWISE PPM FLAT

### BITWISE PPM

- ▶ Idée personnelle
- ▶ Utiliser une PPM (d'ordre max. 3 octets) avec un encodage bit par bit
- ▶ 49 % d'efficacité!
- ▶ Mais l'ordre *fallback* occasionne des pertes évitables...

# Modèles généraux

## BITWISE PPM et BITWISE PPM FLAT

### BITWISE PPM

- ▶ Idée personnelle
- ▶ Utiliser une PPM (d'ordre max. 3 octets) avec un encodage bit par bit
- ▶ 49 % d'efficacité!
- ▶ Mais l'ordre *fallback* occasionne des pertes évitables...

### BITWISE PPM FLAT

# Modèles généraux

## BITWISE PPM et BITWISE PPM FLAT

### BITWISE PPM

- ▶ Idée personnelle
- ▶ Utiliser une PPM (d'ordre max. 3 octets) avec un encodage bit par bit
- ▶ 49 % d'efficacité!
- ▶ Mais l'*order fallback* occasionne des pertes évitables...

### BITWISE PPM FLAT

- ▶ On retire seulement l'*order fallback* en utilisant à la place des probabilités neutres (0.5 : 0.5)



# Modèles généraux

## BITWISE PPM et BITWISE PPM FLAT

### BITWISE PPM

- ▶ Idée personnelle
- ▶ Utiliser une PPM (d'ordre max. 3 octets) avec un encodage bit par bit
- ▶ 49 % d'efficacité!
- ▶ Mais l'*order fallback* occasionne des pertes évitables...

### BITWISE PPM FLAT

- ▶ On retire seulement l'*order fallback* en utilisant à la place des probabilités neutres (0.5 : 0.5)
- ▶ Performances améliorées : 63 % d'efficacité!

# Modèles généraux

## BITWISE PPM et BITWISE PPM FLAT

### BITWISE PPM

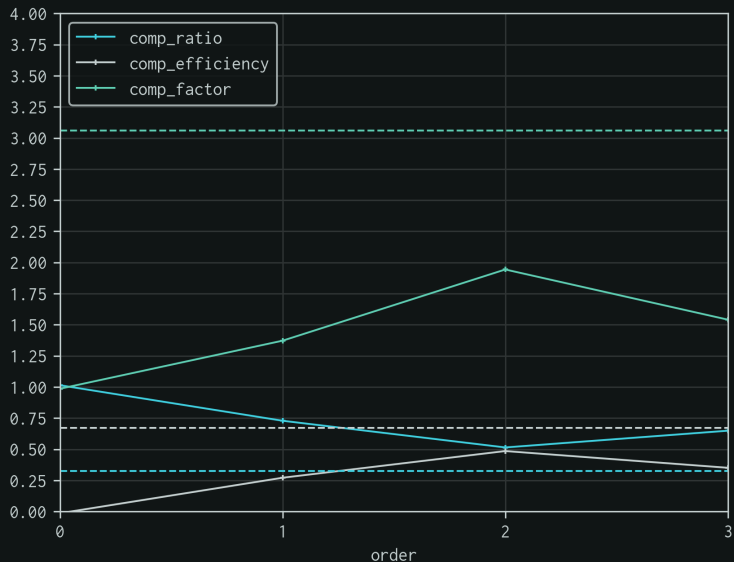
- ▶ Idée personnelle
- ▶ Utiliser une PPM (d'ordre max. 3 octets) avec un encodage bit par bit
- ▶ 49 % d'efficacité!
- ▶ Mais l'*order fallback* occasionne des pertes évitables...

### BITWISE PPM FLAT

- ▶ On retire seulement l'*order fallback* en utilisant à la place des probabilités neutres (0.5 : 0.5)
- ▶ Performances améliorées : 63 % d'efficacité!
- ▶ Simplement un *bitwise encoder* d'ordre fixe (0 - 28 bits) (existait déjà)

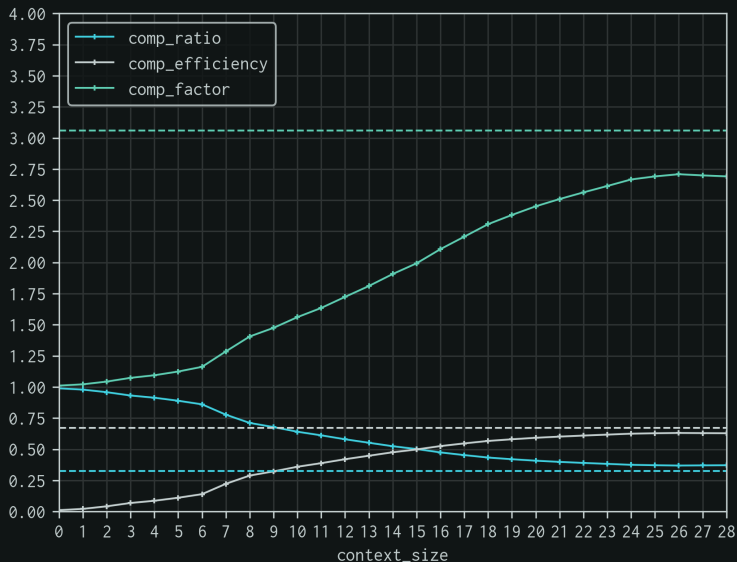
# Modèles généraux

## BITWISE PPM



# Modèles généraux

## BITWISE PPM FLAT



# Impact de la transformée

BWT régularise nos données

BWT

# Impact de la transformée

BWT régularise nos données

BWT

- ▶ Transformation bijective

# Impact de la transformée

BWT régularise nos données

BWT

- ▶ Transformation bijective
- ▶ Tend à placer côte-à-côte les mêmes symboles





# Impact de la transformée

BWT régularise nos données

BWT

- ▶ Transformation bijective
- ▶ Tend à placer côte-à-côte les mêmes symboles

t	u	r	l	u	t	u	t	
	t	u	r	l	u	t	u	t
u		t	u	r	l	u	t	u
t	u		t	u	r	l	u	t
u	t	u		t	u	r	l	u
t	u	t	u		t	u	r	l
u	t	u	t	u		t	u	r
l	u	t	u	t	u		t	u
r	l	u	t	u	t	u		t
u	r	l	u	t	u	t	u	

→ sort →

	t	u	r	l	u	t	u	t	u
l	u	t	u	t	u		t	u	r
r	l	u	t	u	t	u		t	u
t	u		t	u	r	l	u	t	u
t	u	r	l	u	t	u	t	u	
t	u	t	u		t	u	r	l	u
u		t	u	r	l	u	t	u	t
u	r	l	u	t	u	t	u		t
u	t	u		t	u	r	l	u	t
u	t	u	t	u		t	u	r	l

- ▶  $\text{BWT}(\text{tur lututu}) = \text{uruu|utttl}$

# Impact de la transformée

BWT régularise nos données

BWT

- ▶ Transformation bijective
- ▶ Tend à placer côte-à-côte les mêmes symboles

t	u	r	l	u	t	u	t	u		
	t	u	r	l	u	t	u	t	u	
u		t	u	r	l	u	t	u	t	
t	u		t	u	r	l	u	t	u	
u	t	u		t	u	r	l	u	t	
t	u	t	u		t	u	r	l	u	
u	t	u	t	u		t	u	r	l	
l	u	t	u	t	u		t	u	r	
r	l	u	t	u	t	u		t	u	
u	r	l	u	t	u	t	u		t	

→ sort →

	t	u	r	l	u	t	u	t	u	
l	u	t	u	t	u		t	u	r	
r	l	u	t	u	t	u		t	u	
t	u		t	u	r	l	u	t	u	
t	u	r	l	u	t	u	t	u		
t	u	t	u		t	u	r	l	u	
u		t	u	r	l	u	t	u	t	
u	r	l	u	t	u	t	u		t	
u	t	u		t	u	r	l	u	t	
u	t	u	t	u		t	u	r	l	

- ▶  $\text{BWT}(\text{tur lututu}) = \text{ur uu|utttl}$
- ▶ Transforme  $C_m^0$  en  $C^0$

# Impact de la transformée

La BWT en action

Avec la RLE

# Impact de la transformée

La BWT en action

Avec la RLE

- ▶ Très simple : `uruu|utttl` devient `1u1r2u1|1u3t1l`

# Impact de la transformée

La BWT en action

Avec la RLE

- ▶ Très simple : `uruu|utttl` devient `1u1r2u1|1u3t1l`
- ▶ Efficacité de 41 %!

# Impact de la transformée

## La BWT en action

Avec la RLE

- ▶ Très simple : `uruu|utttl` devient `1u1r2u1|1u3t1l`
- ▶ Efficacité de 41 %!

Avec BITWISE PPM FLAT

# Impact de la transformée

## La BWT en action

### Avec la RLE

- ▶ Très simple : `uruu|utttl` devient `1u1r2u1|1u3t1l`
- ▶ Efficacité de 41 %!

### Avec BITWISE PPM FLAT

- ▶ On gagne (seulement) 2 % d'efficacité supplémentaire

# Impact de la transformée

## La BWT en action

### Avec la RLE

- ▶ Très simple : `uruu|utttl` devient `1u1r2u1|1u3t1l`
- ▶ Efficacité de 41 %!

### Avec BITWISE PPM FLAT

- ▶ On gagne (seulement) 2 % d'efficacité supplémentaire
- ▶ Permet d'atteindre mon record personnel :  
65 % d'efficacité  
facteur de compression **2,85x**  
35 % de la taille initiale



# Impact de la transformée

## La BWT en action

### Avec la RLE

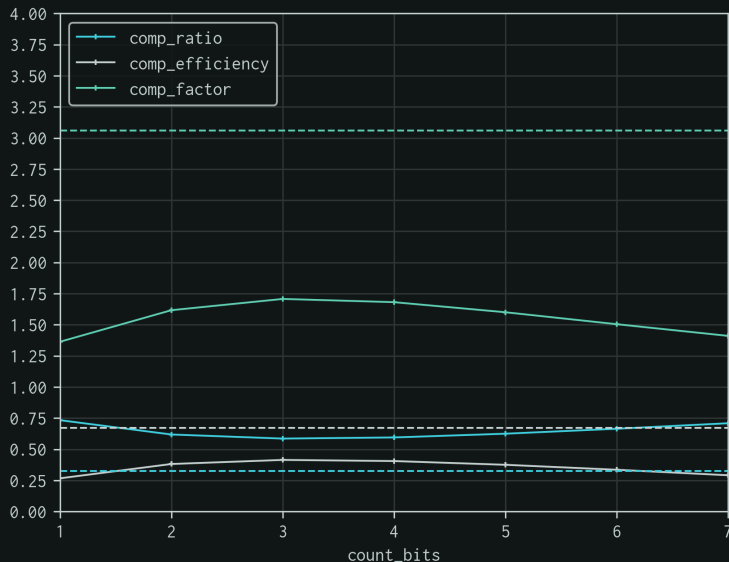
- ▶ Très simple : `uruu|utttl` devient `1u1r2u1|1u3t1l`
- ▶ Efficacité de 41 %!

### Avec BITWISE PPM FLAT

- ▶ On gagne (seulement) 2 % d'efficacité supplémentaire
- ▶ Permet d'atteindre mon record personnel :  
65 % d'efficacité  
facteur de compression **2,85x**  
35 % de la taille initiale  
ZIP : **3.06x** | record (ZPAQ) : **5.43x**

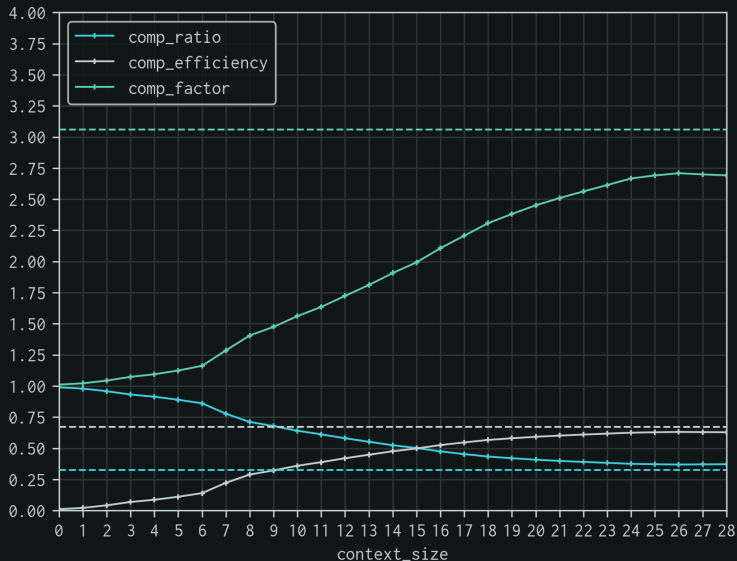
# Impact de la transformée

BWT + RLE



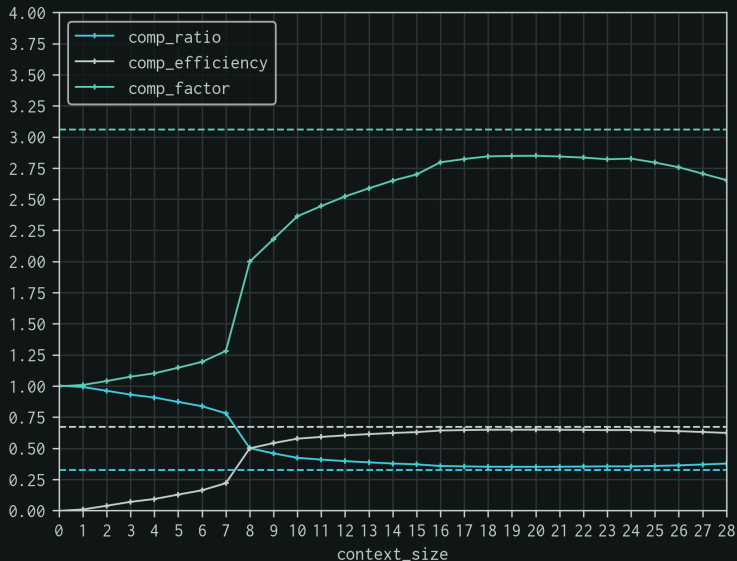
# Impact de la transformée

BITWISE PPM FLAT seul (déjà vu)



# Impact de la transformée

BWT + BITWISE PPM FLAT



# Conclusion

Merci pour votre attention !

## Sources principales

- ▶ *Data Compression Explained*, Matt MAHONEY  
[mattmahoney.net/dc/dce.html](http://mattmahoney.net/dc/dce.html)
- ▶ *Suffix Array by Induced Sorting* (pour la BWT),  
G. NONGS. ZHANG, W. H. CHAN  
[code.google.com/archive/p/ge-nong/](http://code.google.com/archive/p/ge-nong/)

## Tout le dossier disponible

- ▶ <https://github.com/tbagrel1/tipe>

---

Thomas BAGREL - Lycée Henri POINCARÉ, Nancy