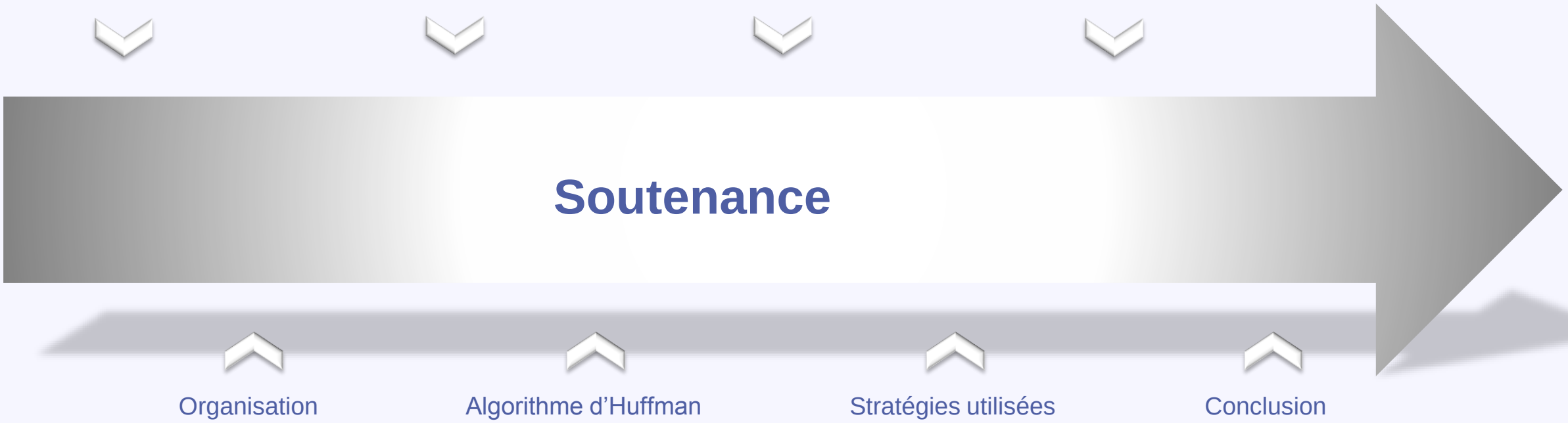
S2.01 Développement d’une application

Webpagesaver

RAHARISON Leong Foc Sing Andhy
DEMARIA Matthieu
ABERT—SIFFERLEN Tom
GOMEZ CORONA Kevin

Sommaire

#1


Cahier des charges
Architecture
Formats de fichier
Études des résultats
Soutenance
Organisation
Algorithme d’Huffman
Stratégies utilisées
Conclusion

Cahier des charges

#2

Cahier des charges

#3

Finalité : Créer une application permettant le téléchargement et la compression d'une page web

Objectifs :

- O1 : Créer une librairie de compression
  - E1.1 : A l'aide de l'algorithme d'Huffman
  - E1.2 : Sans utiliser de librairies externes

- O2 : Créer le logiciel de téléchargement de page web
  - E2.1 : Télécharger l'ensemble des ressources de la page web
  - E2.2 : Implémenter la librairie pour rendre l'application fonctionnelle

Fonctionnalités :

Add

Remove

List

View

Organisation

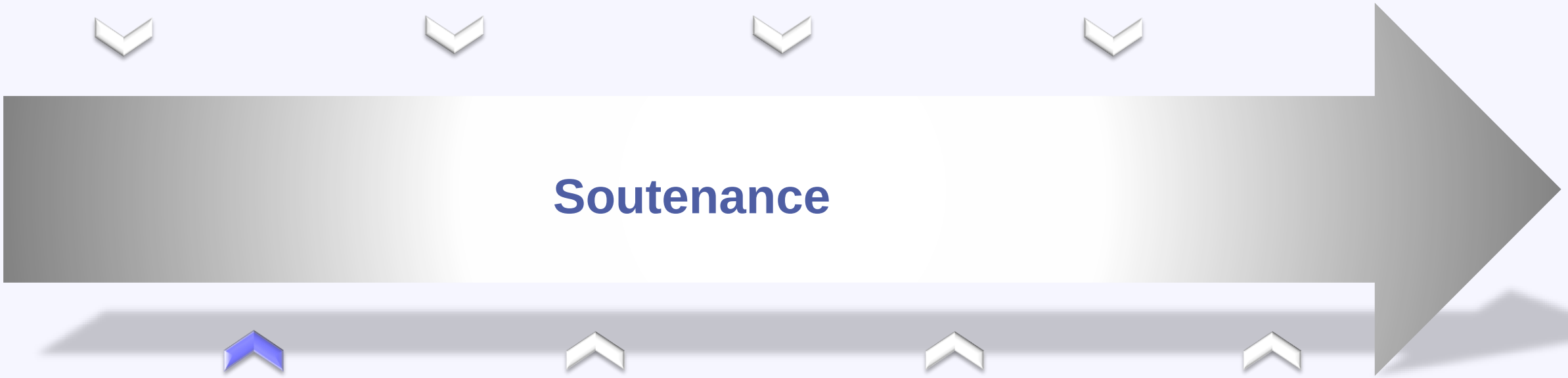
#4

Cahier des charges

Architecture

Formats de fichier

Études des résultats



Organisation

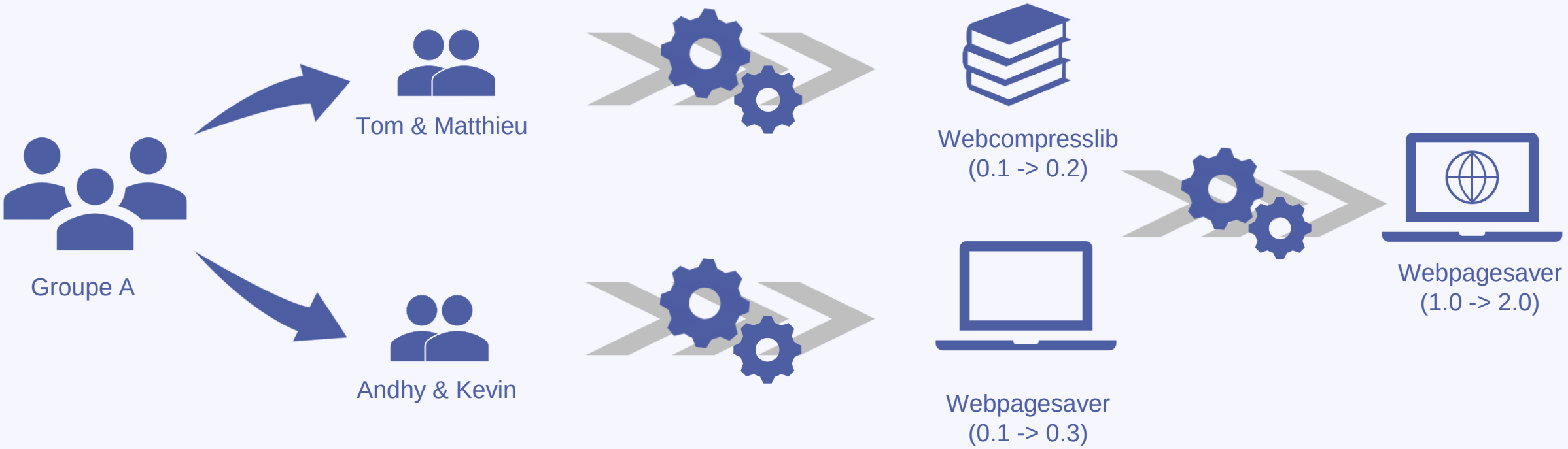
Algorithme d’Huffman

Stratégies utilisées

Conclusion

Organisation :

#5



Architecture

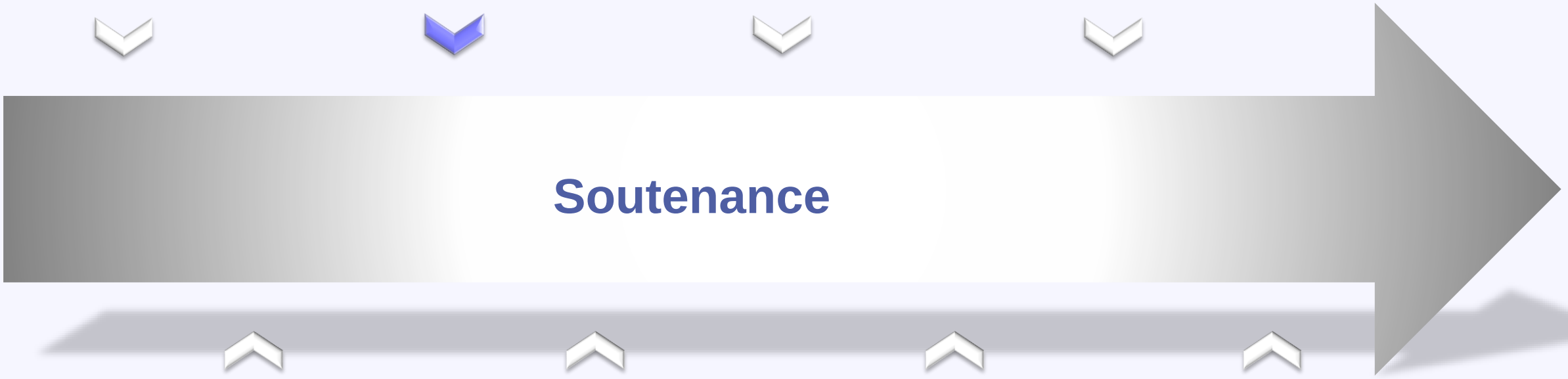
#6

Cahier des charges

Architecture

Formats de fichier

Études des résultats



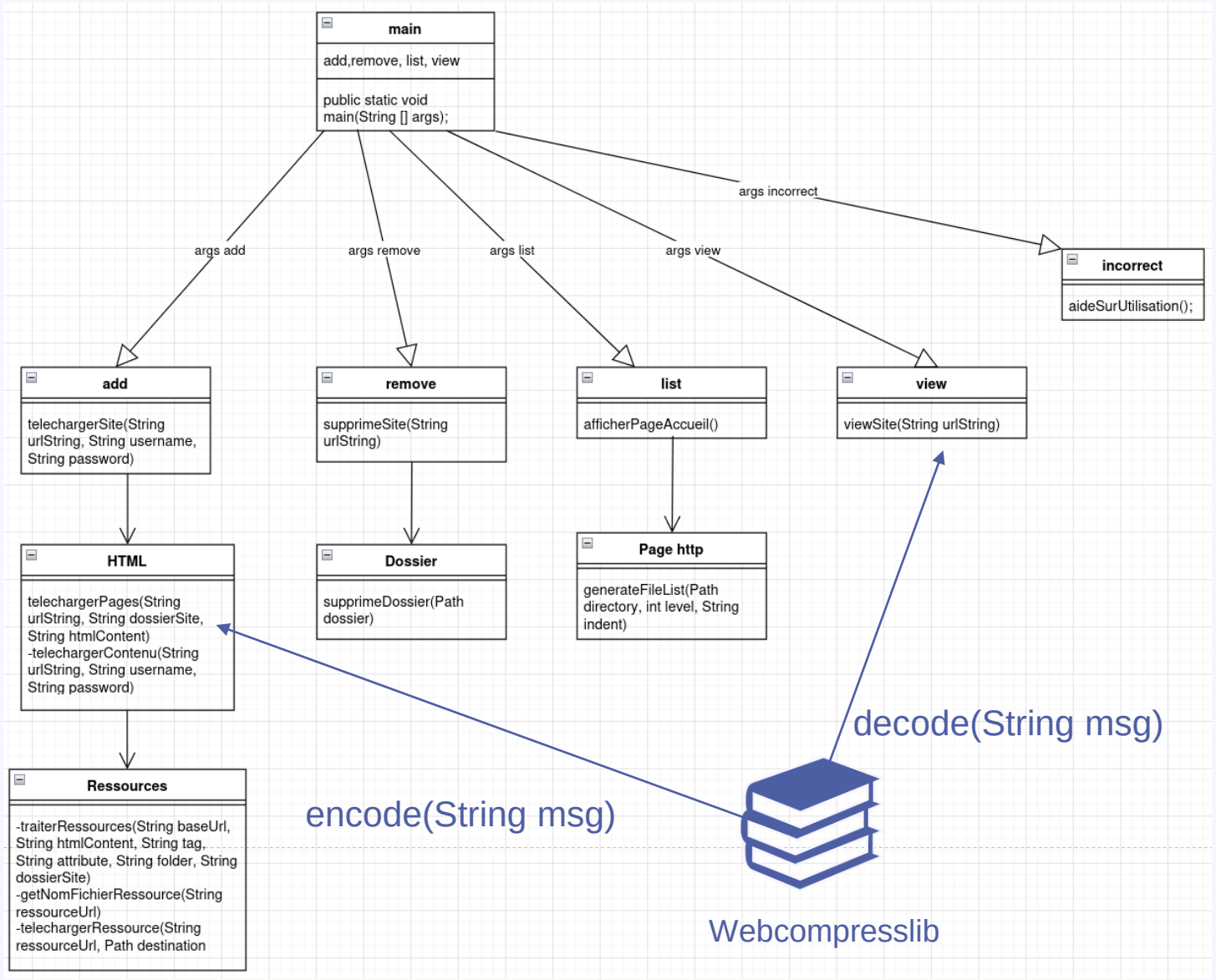
Organisation

Algorithme d’Huffman

Stratégies utilisées

Conclusion

Architecture WebPagesSaver



#7

webpagesaver

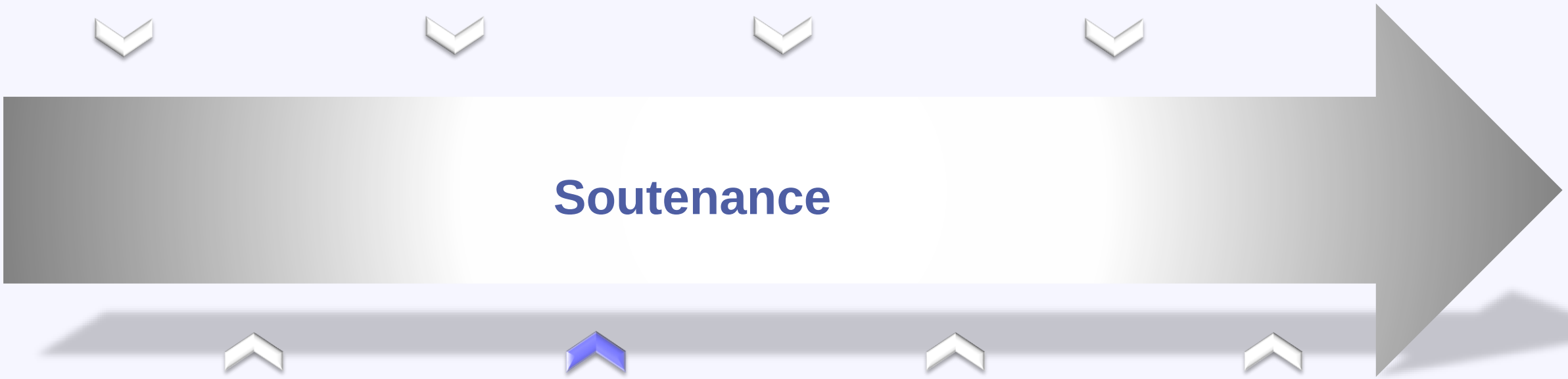
.classpath

javac -cp

webcompresslib

Algorithme d’Huffman

#8



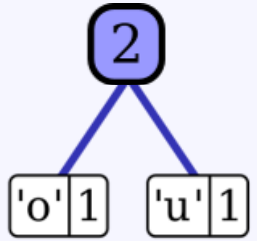
Algorithme d’Huffman

- Construction de l’arbre

Tableau d’occurrences

e	n	…
4	2	…

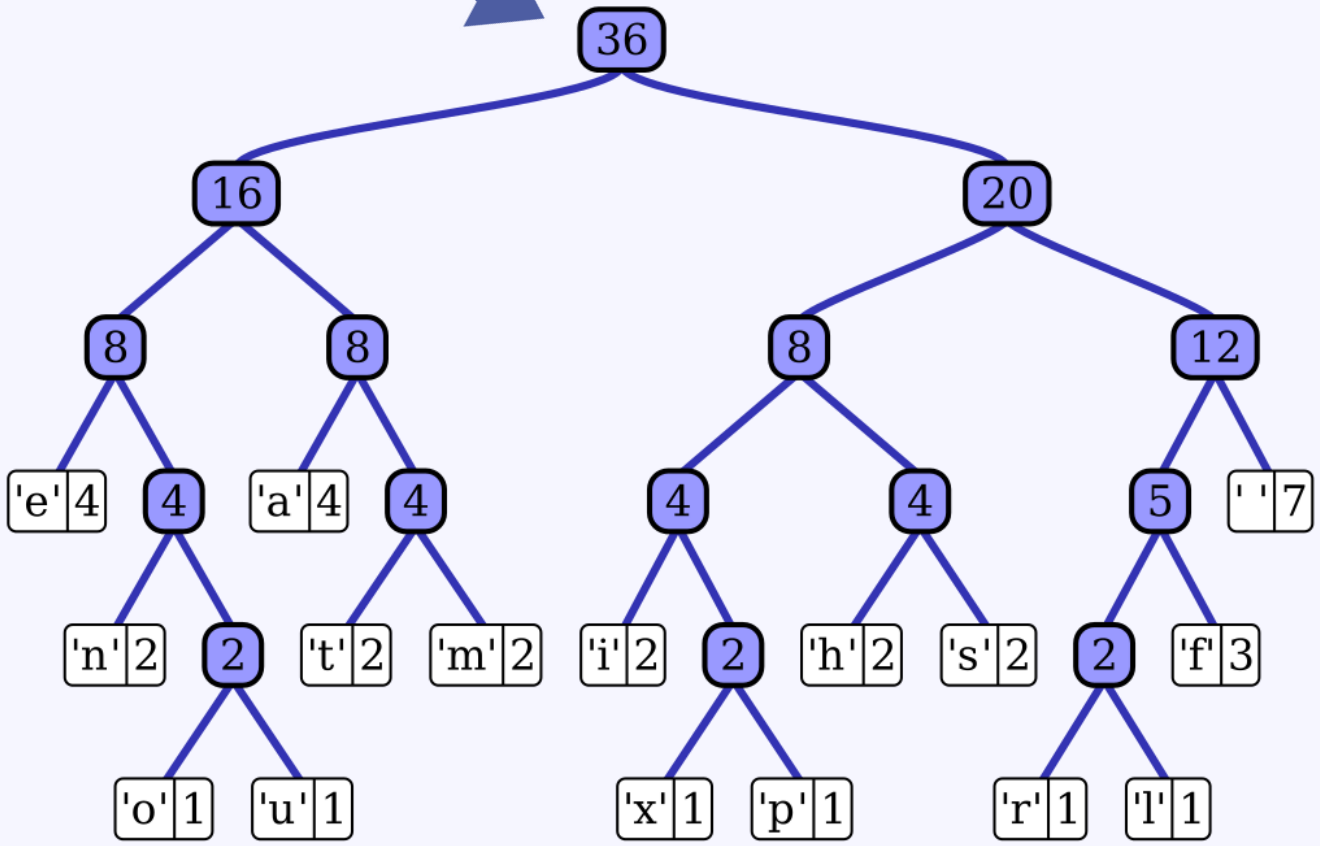
Les nœuds de poids les plus faibles



Caractère

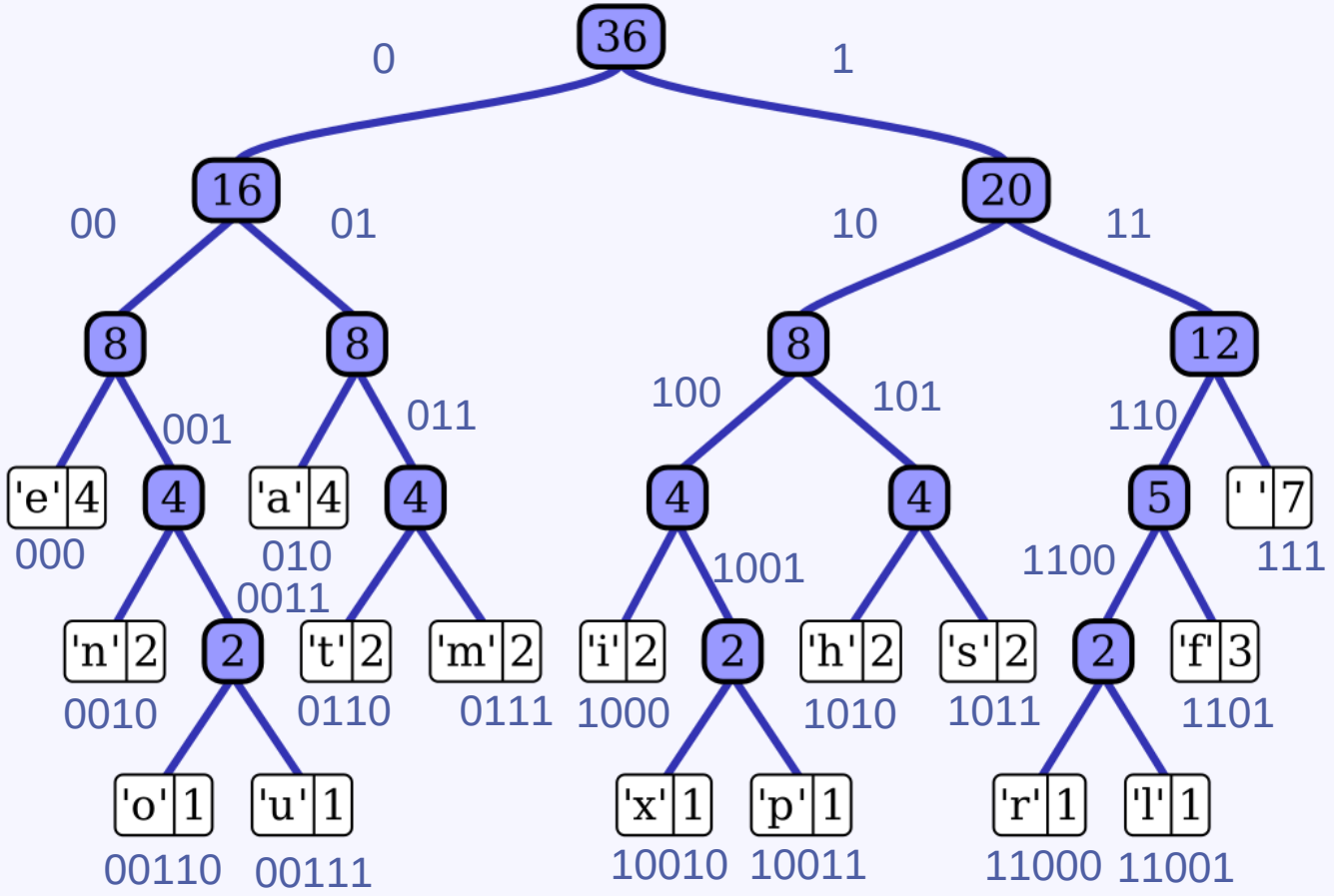
Nombre d’occurrences

Nombre de caractères total



Algorithme d’Huffman

- Création du dictionnaire



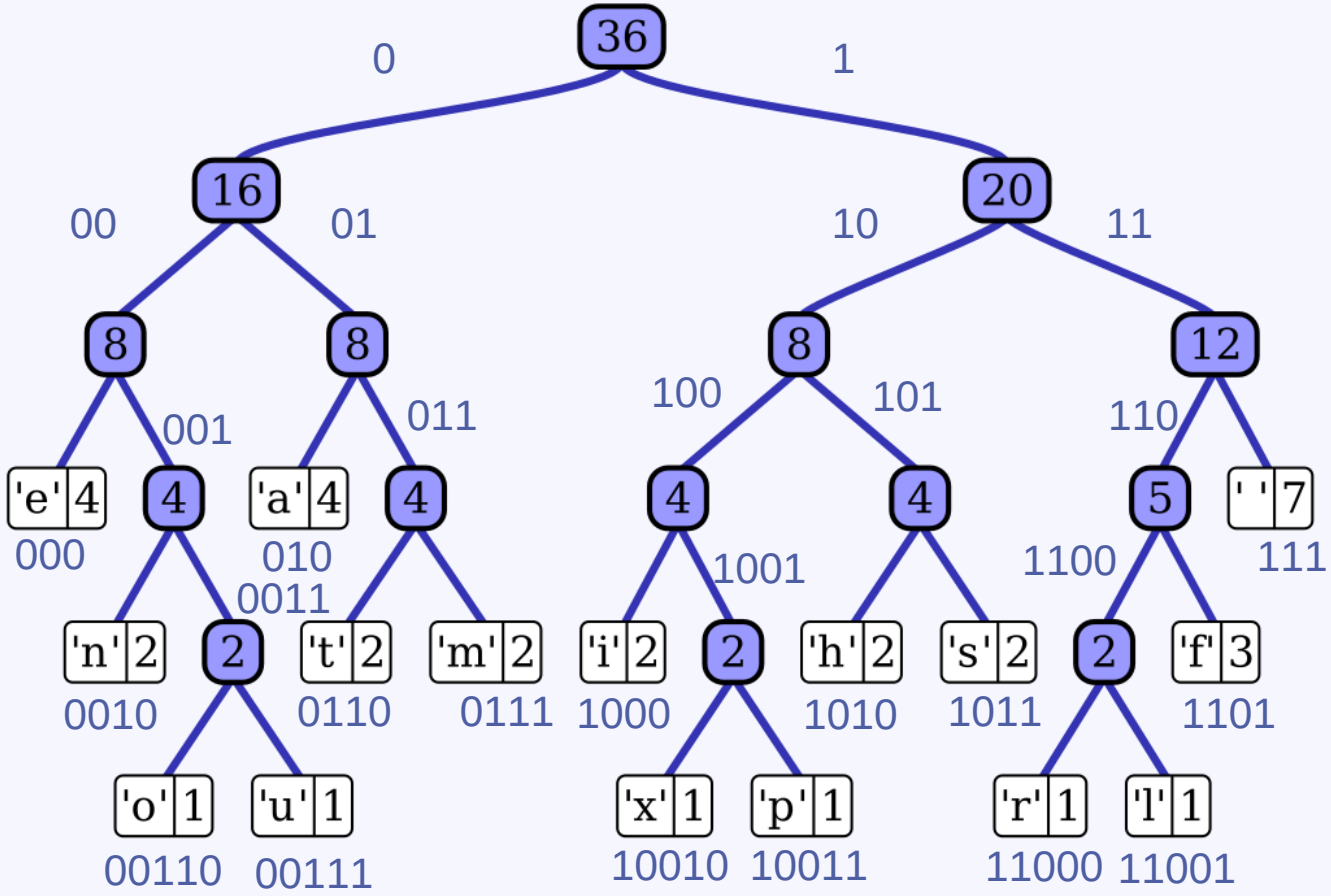
{‘e’ = ‘’000’’, ‘n’ = ‘’0010’’, …}

Algorithme d’Huffman

- Encodage du message

{‘e’ = ‘’000’’, ‘n’ = ‘’0010’’, …}
+
Texte

Binaire
(00100011101…)



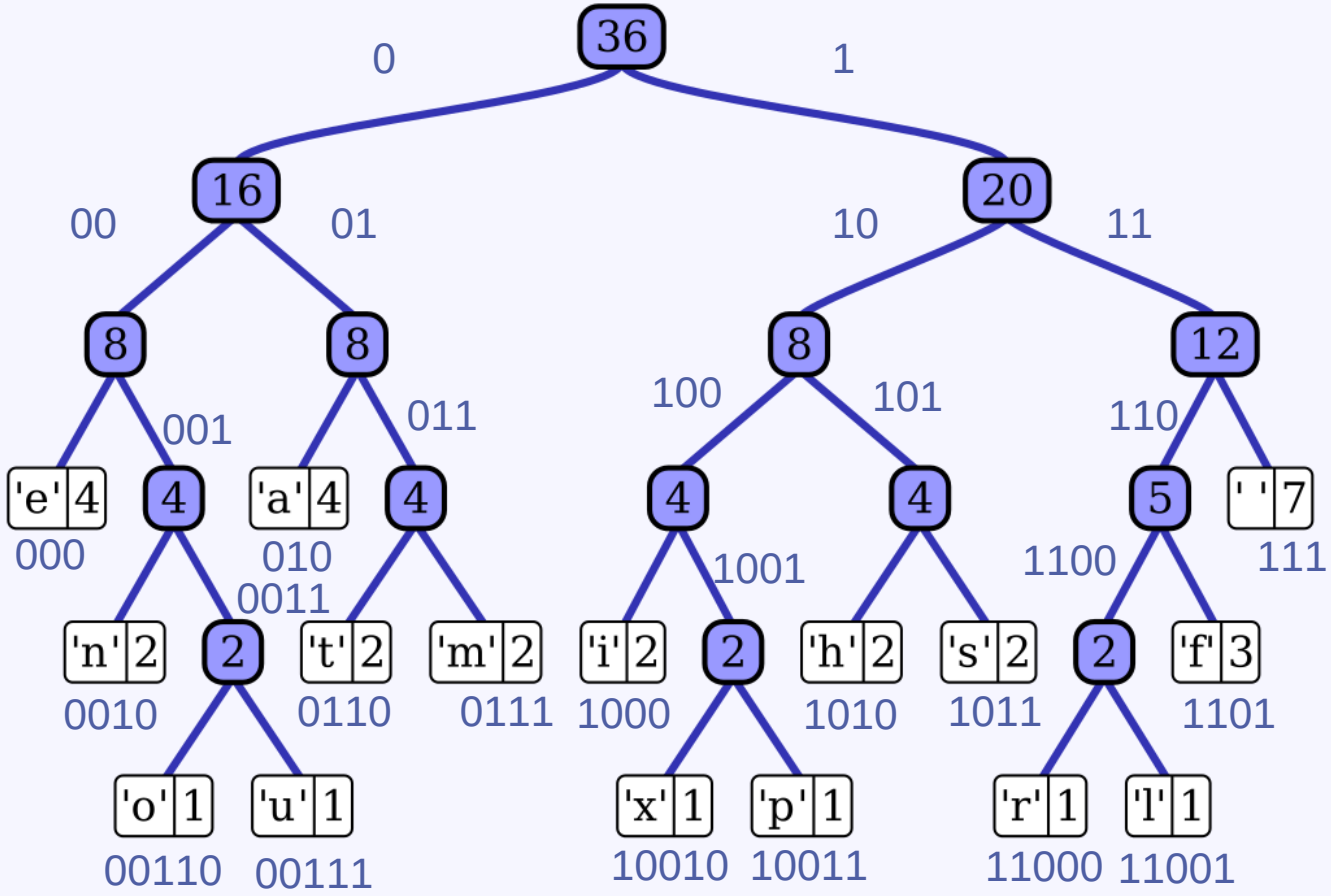
Algorithme d’Huffman

#12

- Décodage du message

Binaire
(00100011101…)
+
{‘e’ = ‘’000’’, ‘n’ = ‘’0010’’, …}

Texte



Formats de fichier

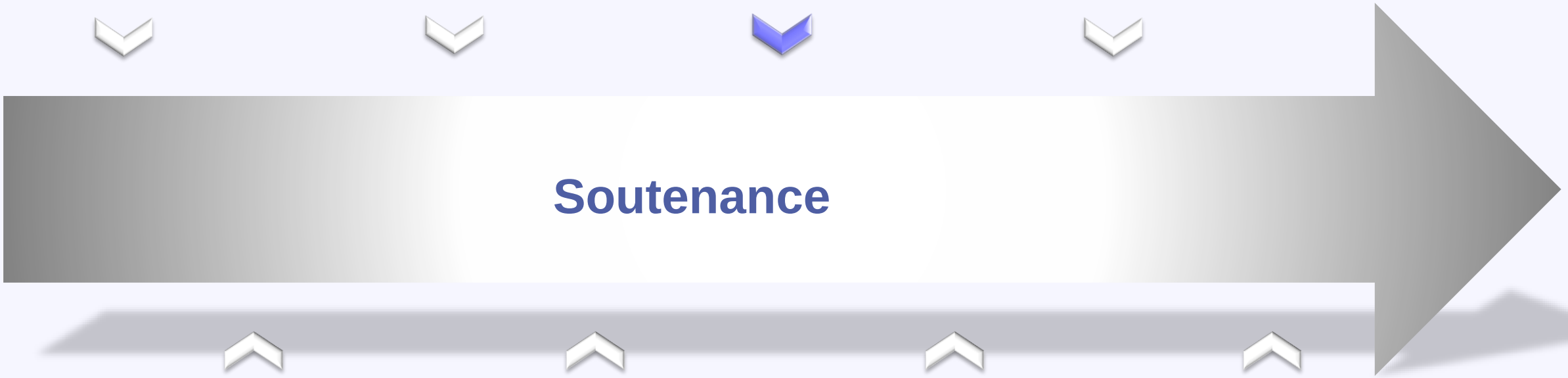
#13

Cahier des charges

Architecture

Formats de fichier

Études des résultats



Organisation

Algorithme d’Huffman

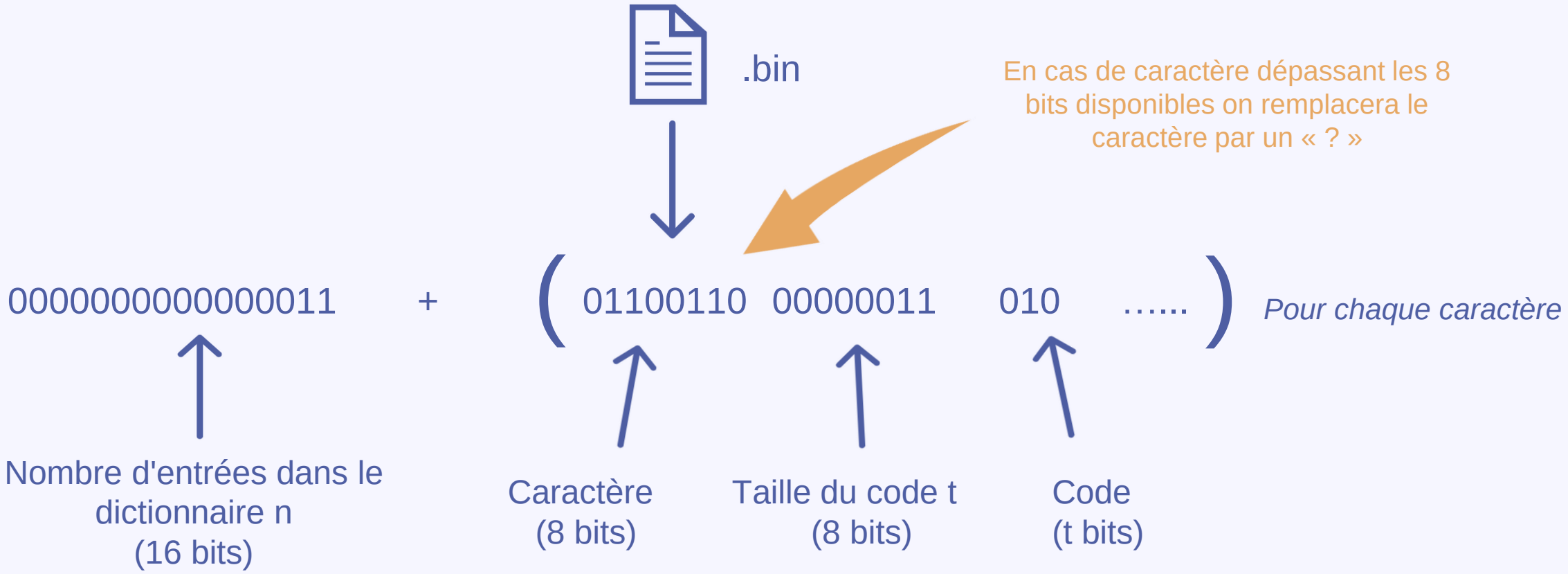
Stratégies utilisées

Conclusion

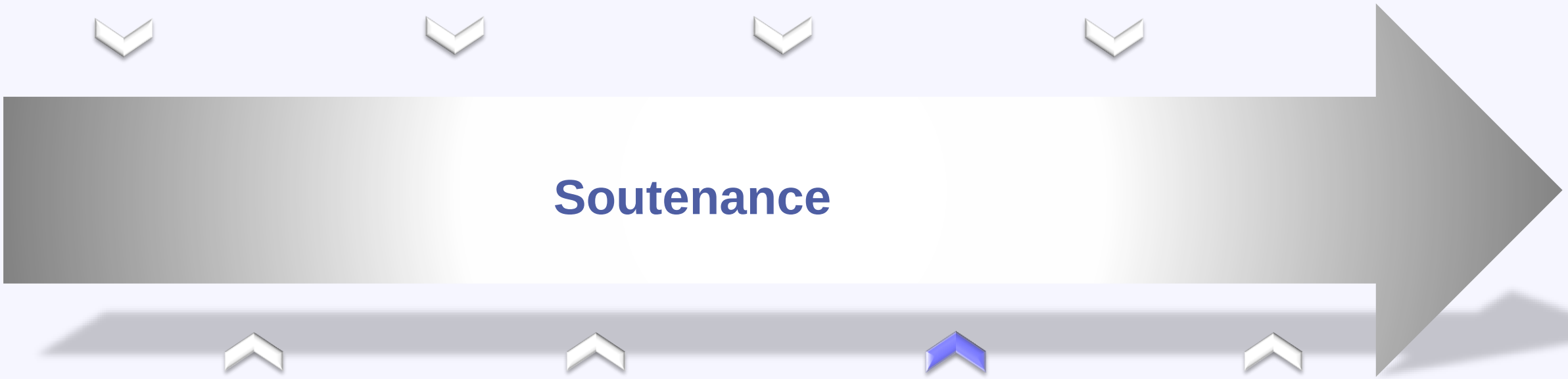
Les formats de fichier

- Les fichiers .html .css et .js sont compressés en fichier .bin

Mais comment incorporer le dictionnaire de décodage dans le fichier binaire ?

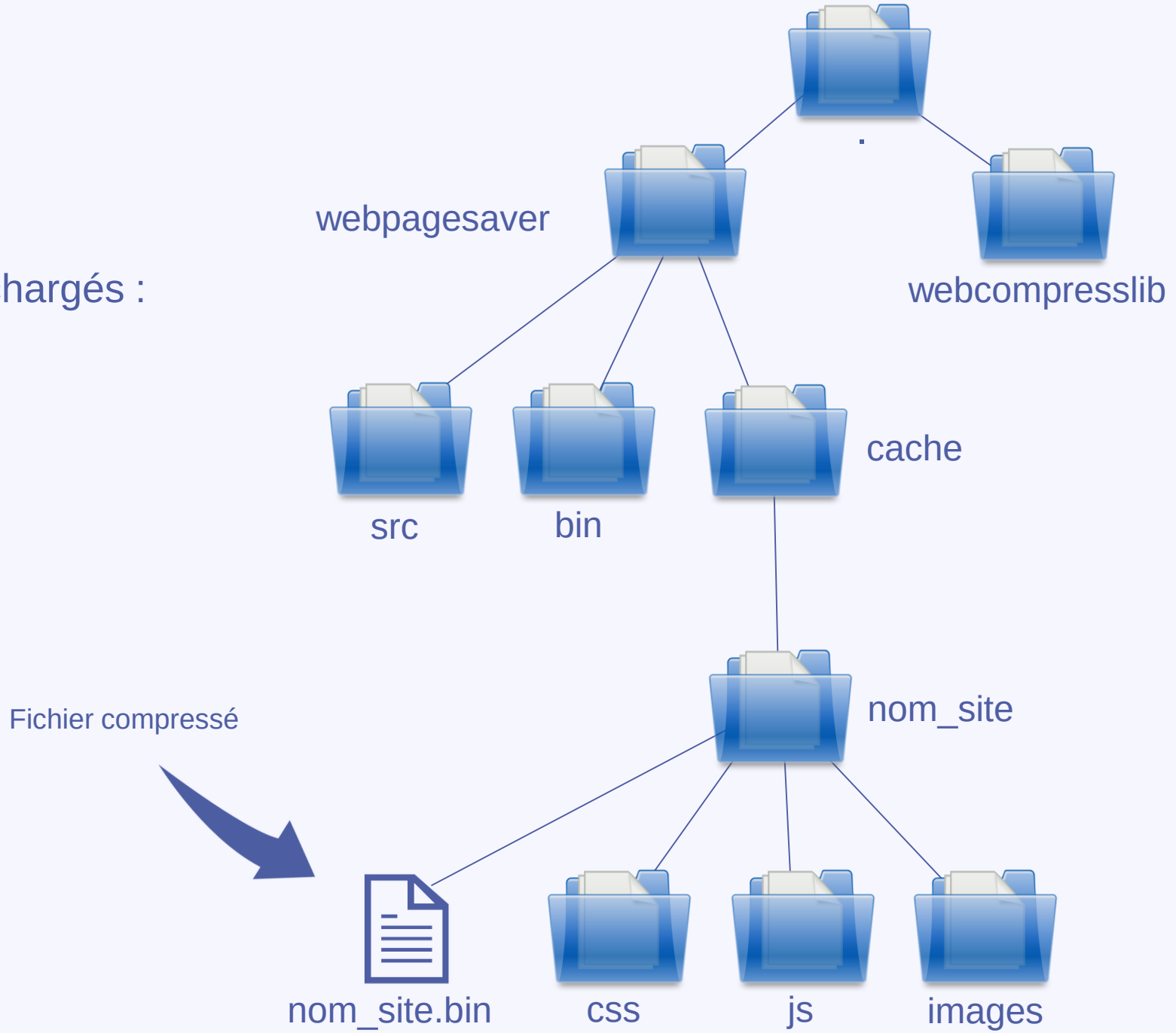


Stratégies utilisées



Stratégies utilisées

- Arborescence des site téléchargés :



Stratégies utilisées

- Amélioration maison de list :

Attendu :

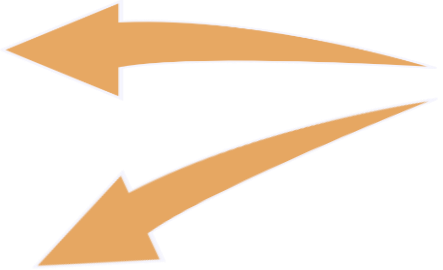
Webcompresslib

- Site 1
- Site 2
- Site 3

Version améliorée :

Webcompresslib

- Site 1
  .css
  .js
  .png
  .html
- Site 2
  .css
  .js
  .png
  .html



Cliquables si **view** déjà utilisé

Études des résultats

Études des résultats

- Temps d’exécution :

Estimation de la complexité de la compression : $O(n^3)$

Pour un site « classique » : Uniquement la compression <45s

Pour un site « classique » : Téléchargement + Compression <2mn

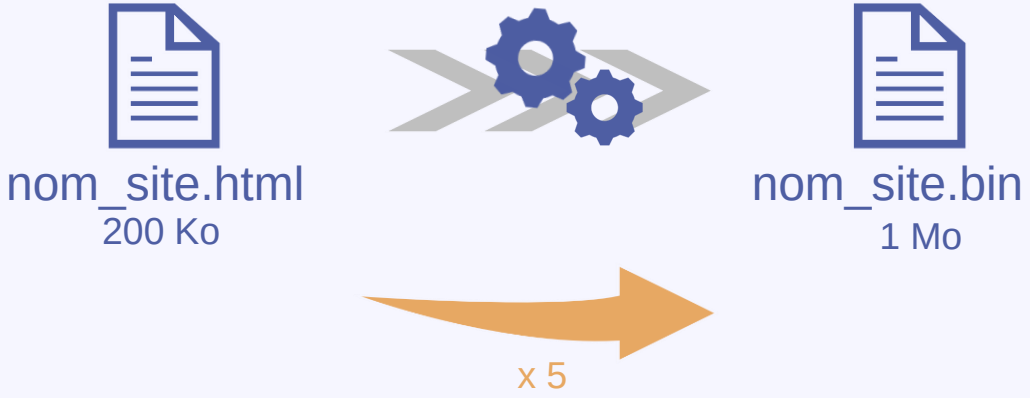
Le temps d’exécution est en pratique bien plus dépendant du nombre d’images à télécharger que de la compression

Études des résultats

- Taux de compression :

Théorie :

En pratique :



Problème d'encodage :
Encodage octets par octets à la place de bits par bits
(Sera corrigé si possible pour le rendu final)

Études des résultats

.bin
? bits

- Calcul de la taille du fichier .bin :

Taille du dictionnaire encodé =

$$16 + \sum_{k=1}^{i} (16 + t_k) \text{ bits}$$

i étant le nombre d'entrées du dictionnaire
t_k étant la taille du code du k-ième caractère

Taille du message encodé =

$$\sum_{k=1}^{i} n_k t_k \text{ bits}$$

n_k étant le nombre d'occurrences du k-ième caractère
t_k étant la taille du code du k-ième caractère

Taille du fichier binaire =

$$16 + \sum_{k=1}^{i} (16 + t_k) + \sum_{k=1}^{i} n_k t_k = 16 + \sum_{k=1}^{i} (16 + t_k) + n_k t_k \text{bits}$$

Études des résultats

- Différents cas :

Fichier lourd avec peu de caractères différents

Très bon taux de compression

Fichier léger avec peu de caractères différents

Bon taux de compression

Fichier lourd avec beaucoup de caractères différents

Mauvais taux de compression

Fichier léger avec beaucoup de caractères différents

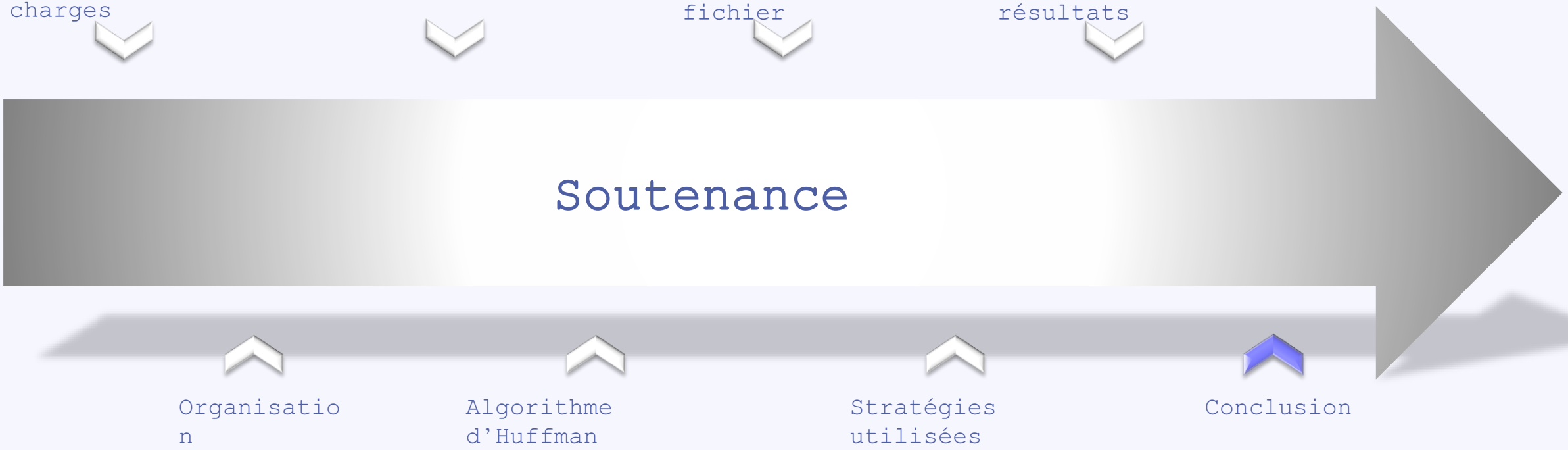
Très mauvais taux de compression

Le nombre de caractères différents est donc plus important que la taille du fichier pour le taux de compression de l'algorithme d'Huffman

Études des résultats (suivi de versions)

Tableau comparatif	V0.1	V0.2	V0.3	V1.0	V2.0
WebPagesSaver					
Téléchargement :					
HTML	✓	✓	✓	✓	✓
CSS	✗	✓	✓	✓	✓
JS	✗	✓	✓	✓	✓
IMG	✗	✗	✓	✓	✓
WebCompresslib					
Huffman :					
Arbre Binaire	✓	✓	-	✓	✓
Dictionnaire	✓	✓	-	✓	✓
Encodage :					
Sans dictionnaire	✗	✓	-	✓	✓
Avec dictionnaire	✗	✗	-	✓	✓
Fichier binaire	✗	✗	-	✓	✓
Optimisation	✗	✗	-	✗	✓
Décodage	✗	✓	-	✓	✓
Import	✗	✗	-	✓	✓

Conclusion


Cahier des charges
Architecture
Formats de fichier
Études des résultats
Soutenance
Organisation
Algorithme d'Huffman
Stratégies utilisées
Conclusion

Conclusion

- Bilan :

Réussites

Algorithme de compression fonctionnel

Toutes les fonctionnalités principales de webpagesaver sont fonctionnelles

Code robuste

Bon travail d'équipe malgré l'effectif réduit

Échecs

Serveur HTTP non implémenté

Problème du taux de compression

CSS et Javascript pas encore compressés à ce jour

À améliorer

Passer plus de temps sur l'analyse/la conception plutôt que sur le code

Prendre de l'avance pour ne pas être en rush avant un rendu intermédiaire

Créer les tests avant le code

Mieux cibler les failles potentielles de notre code avant son développement en vue de développer les tests

100011010010001111 0110100110 0110101001101100111 0110000101101110111110111100011111

(Merci de votre attention)