Algorithmique (en C)

Julien Alexandre dit Sandretto

Department U2IS
ENSTA ParisTech
Version 2019-2020

ENSTA

IP PARIS

L'algorithmique (rappels)

« *Algorithmie ou algorithmique ?* »

Algorithmie est un néologisme dérivé de algorithmique !

Alors pourquoi ?

Parfois plus clair, algorithmique est à la fois un adjectif et un nom. Pour un domaine d'étude, la terminaison "ie" est souvent privilégiée.

En conclusion

Les deux sont acceptés.

« Définition »

L'algorithmique est l'étude et la production de règles et techniques qui sont impliquées dans la définition et la conception d'algorithmes, c'est-à-dire de processus systématiques de résolution d'un problème permettant de décrire précisément des étapes pour résoudre un problème algorithmique.

« Un algorithme »

Un algorithme est une procédure de calcul qui prend en entrée un ensemble de valeurs et qui retourne en sortie un ensemble de valeurs.

Exemple :

- Problème : trier une suite de nombres entiers dans l'ordre croissant
- Entrée : une suite de n nombres entiers $(a_1, .., a_n)$
- Sortie : une permutation de la suite donnée en entrée $(a'_1, .., a'_n)$ telle que $a'_1 \leq a'_2 \leq .. \leq a'_n$

« Un algorithme »

Un ensemble particulier de valeurs données en entrée est une **instance** du problème.
Exemple : (2,5,1,3) est une instance du problème de tri.

« *Un algorithme* »

Un algorithme est **correct**, si pour toute instance du problème :
il se termine **et** produit une sortie correcte.

« *Un algorithme* »

Un algorithme a besoin que les données d'entrées soient organisées pour pouvoir travailler.
Une **structure de données** est utilisée pour stocker de manière organisée les données. Dans ce cours, nous verrons :

- les structures
- les tableaux
- les piles
- les listes chainées
- les arbres
- les graphes

« *Un algorithme* »

Un algorithme est plus ou moins efficace. Pour mesurer cette efficacité, nous utilisons la complexité :

- en temps : le nombre d'opération élémentaires pour traiter une instance de taille n
- en mémoire : l'espace mémoire nécessaire pour traiter une donnée de taille n

« *Un algorithme* »

Un algorithme est construit au moyen de structures de controle, les trois principales sont :

- le bloc d'instructions (réalisées l'une après l'autre)
- l'alternative (on réalise soit l'une soit l'autre suivant une certaine condition)
- la répétition (on répète la même instruction (ou bloc) un certain nombre de fois ou jusqu'à une certain condition)

La structure

« La structure »

La première structure de données est la séquence.
Une **séquence** est une suite d'éléments $(e_1, .., e_n)$ de l'ensemble E ($e_i \in E$).
Exemples : (2,5,8,1) ou (a,f,e,t) sont des séquences (d'entiers ou de caractères)

Un vecteur peut être défini par une séquence.

« La structure »

La **structure** est une généralisation de la séquence :
$(e_1, .., e_n)$ est une structure de l'ensemble $E_1 \times .. \times E_n$ $(e_i \in E_i)$.
Exemples : un individu peut être défini par une structure :

- Nom : séquence de caractères
- Age : entier
- Sexe : booléen
- Taille : réel
- Profession : séquence de caractères

Remarque : les 4 types prédéfinis ont été utilisés dans cette structure !

« La structure »

Une structure peut être définie en fonction d'autres structures.
Exemples : une entreprise peut être définie par une structure :

- Nom : séquence de caractères
- Personnel : séquence d'individus

Ceci permet de créer des structures très complexes !

Recherche en table

« Pourquoi cette question ? »

- Problème : retrouver une information à partir d'une **clef** (info **discriminante**) qui lui est associée.
- Exemples :
  - Dictionnaire : mot $\mapsto$ définition.
  - Annuaire : (nom + prénom) $\mapsto$ (adresse + téléphone).
  - Courbe temporelle : date $\mapsto$ mesure.
- Opérations souhaitées :
  - **Insertion**.
  - **Recherche**.
  - Suppression.

Choix de la structure de données à utiliser ?

Paires « *clef-valeur* »

- But : gérer un ensemble d'éléments (`clef`, `valeur`).
- La `clef` donne accès à **l'information** `valeur`.
- Les clefs représentent un ensemble : m éléments indexés de 0 à $m-1$.
  - Clefs entières sur 16 bits : $2^{16} = 65536$, d'où clefs de 0 à 65535.
  - Téléphone, 10 chiffres décimaux : $10^{10}, \approx 2^{33}$ clefs.
  - Noms sur 8 lettres : $26^8 \approx 2^{37}$ clefs.
- Les clefs doivent être idéalement **discriminantes** : 1 clef $\rightarrow$ 1 valeur.
- Problème : généralement, nombre de clefs possibles >> nombre de valeurs à enregistrer.

Plusieurs solutions

- Pas de solution unique.
- **Compromis** entre :
  - Complexité **spatiale** (coût mémoire du stockage).
  - Complexité **temporelle** des opérations (insertion, recherche, suppression).
- Cette notion de « table » est générale :
  - stockage en mémoire ou sur disque → ± même problème.

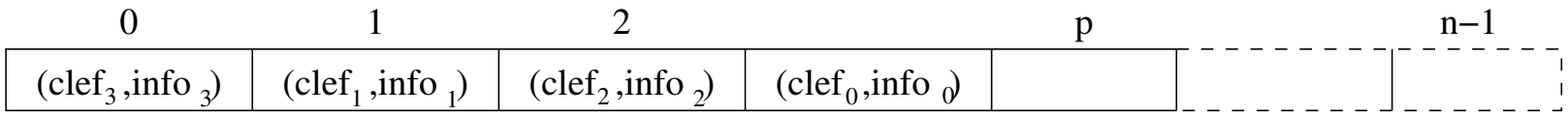
L'adressage direct

- Les résultats sportifs : **3ème** du classement → **3ème** *« case »*.

- Utilisation d'un tableau de taille m (nombre de clefs possibles).
- Info liée à la clef i à la *« case »* d'indice i : **pas de stockage de la clef**.
- ⇒ Complexité **spatiale** en $\theta(m)$, ou $\theta(m \times \text{taille info})$.
- **Mais** : si clef = mot de 10 lettres, $m = 2^{47}$! ! !
- ⇒ Même avec 1 seul bit d'info → 16 To de stockage ! ! !
- ⇒ Irréalisable en pratique dans la majorité des cas.

Recherche séquentielle (1)

- Utilisation d'un tableau de taille n (nombre d'infos).
- Infos stockées dans l'ordre *« d'arrivée »*.
- Mémorisation d'un indice p dénotant la 1ère *« case »* libre.



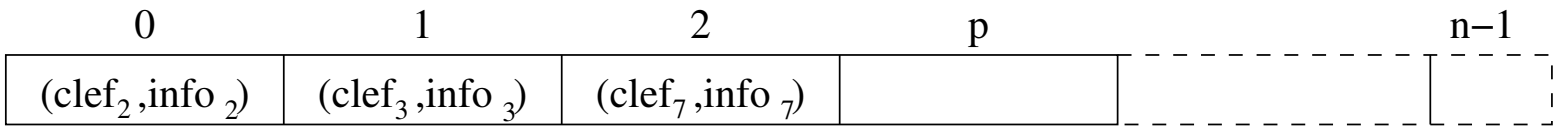
- Complexité **spatiale** en $\theta(n)$ (complexité minimale).
- **Insertion** : `t[p]` $\leftarrow$ (`clef`, `info`)
  - $\Rightarrow$ Complexité en $\theta(1)$.
  - Si recherche préalable pour éviter les doublons : ajout à la complexité.

Recherche séquentielle (2)

- Recherche trop lente.
- Utilisable si l'on insère des éléments mais qu'on les lit **rarement**.
- Exemple proche : les journaux (« *logs* »).
- Si les comparaisons sont complexes, devient excessivement coûteux :
  - Comparer des entiers : trivial, en $\theta(1)$.
  - Comparer 2 chaînes de caractères, $l = min(|s_1|, |s_2|) \Rightarrow \theta(l)$.
  - Comparer sur des clefs étant elles-mêmes des tableaux... ☹
  - Ou pire ...

Recherche dichotomique (1)

- La recherche dans le dictionnaire.
- Nécessite un tableau de taille n dont les éléments sont **triés** sur les **clefs**.
- Dans le dictionnaire : tri des mots par ordre **lexicographique**.



- Processus de recherche dichotomique :
  - Accès au milieu du tableau.
  - Si clef trouvée = clef recherchée → trouvé.
  - Si clef trouvée < clef recherchée → recherche **dichotomique** dans la partie «**droite** » du tableau.
  - Si clef trouvée > clef recherchée → recherche **dichotomique** dans la partie «**gauche** » du tableau.

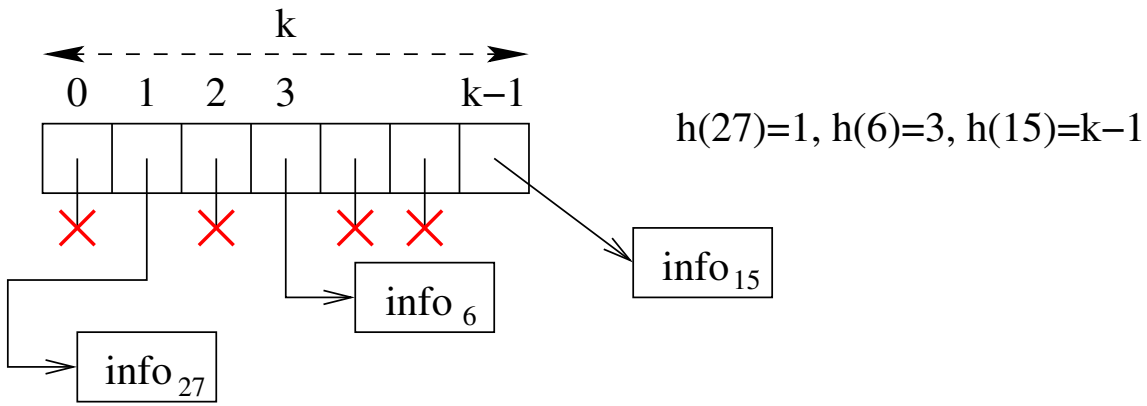
Recherche dichotomique (2)

- **Recherche** : parcours dichotomique jusqu'à trouver la clef.
  - Complexité en $\theta(log\ n)$.
- **Insertion** : recherche puis décalage.
  - Complexité en $\theta(n)$.
- **Suppression** : recherche puis décalage.
  - Complexité en $\theta(n)$.

- Bonne complexité de recherche.
- **Mais insertion** (et **suppression**) trop chère(s).
- Insertion : plus rentable d'insérer tout puis de trier tout le tableau d'un coup.

Table de hachage (« *hastable* »)

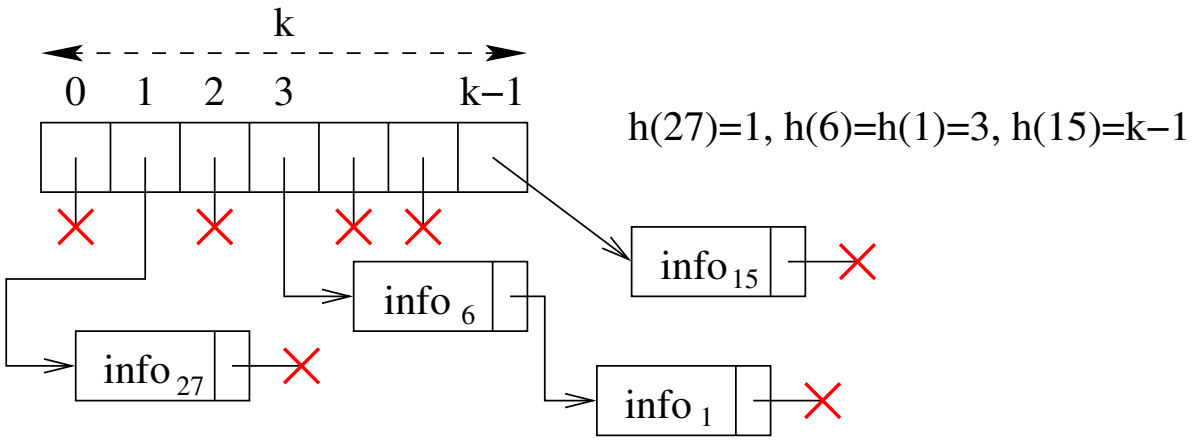
- Réduire l'espace des clefs possibles, m, à un espace plus petit, k (celui des clefs réellement utiles).
- Fonction de **hachage** $h : [0; m-1] \to [0; k-1]$ avec $k < m$.
- On utilise un tableau de taille k.
- On stocke chaque information dans le tableau à l'indice $h(clef)$.
  - Chaque case contient l'information dont le **hachage de la clef** vaut l'indice de cette case.



- Mais que se passe-t-il si 2 clefs ont le même « hash » ?
  - **Collision**.

Table de hachage et collisions

- **Collision** : plusieurs infos avec le même hachage de clef.
- Au lieu de mettre 1 info dans la table, on met une « *liste* » d'infos.
- Toutes les infos d'une liste ont le **même hachage** de leur clef.



- Autre solution : **adressage ouvert** → utiliser la 1ère case contiguë libre.

Complexité des tables de hachage

- Les complexités dépendent du **facteur de remplissage** ρ de la table.
- Complexités (en moyenne) :
  - spatiale : $\theta(k+n)$ (taille de la table + taille des données),
  - recherche : $\theta(\rho)$,
  - insertion : $\theta(1)$,
  - suppression : $\theta(\rho)$.
- Nécessité d'une bonne fonction de hachage (non biaisée sur les entrées).
- Nécessité de connaître n à l'avance pour choisir $k \approx n$.