

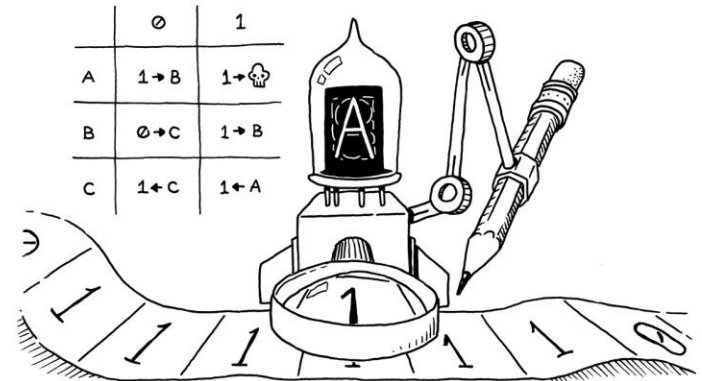
CS353 – Algorithmique et structure de données

03/02/2022 première séance

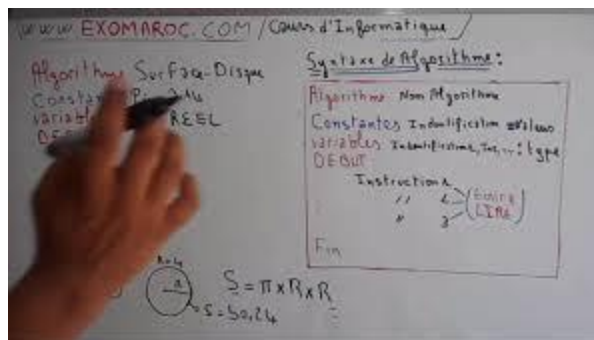
Introduction du cours

Algorithmes

- Introduction
- Définitions
- Exemples
- Pseudo-code



De quoi s'agit-il ?



CS353 – ALGORITHMIQUE ET STRUCTURE DE DONNÉES

Enseignants : E. Brun et Y. Guido (CM et TD), Adrien Rochedy- Vacataire (TP)

Enseignements :

- Cours Magistraux : 18 heures (12 séances de 1h30 ; 7xYG-5xEB)
- Travaux dirigés : 18 heures (12 séances de 1h30 ; 5xYG-7xEB)
- Travaux Pratiques : 12 heures (8 séances de 1,5h)

Crédits : 5

Evaluation :

- Contrôle continu (30% de la note) : DS en présentiel, travail à la maison
- Travaux Pratiques (20% de la note) : Exercices en binômes, rédaction de rapports.
- Examen (50% de la note) : Epreuve écrite individuelle portant sur l'ensemble des notions abordées dans l'année.

Travail :

- Standard : 5 ECTS = $5 \times 25 = 125$ heures : 48 heures encadrées (CM-TD-TP) + **77 heures de travail perso**
- Plus réaliste : 5 ECTS = $5 \times 20 = 100$ heures : 48 heures encadrées (CM-TD-TP) + 52 heures de travail perso

=> 1 heure de cours/TD/TP = 1 heure de travail à la maison.

Attention : tâches de programmation chronophages !

Attendus :

- utiliser des techniques de résolution de problèmes à l'aide d'algorithmes ;
- décrire des algorithmes et des structures algorithmiques : structures de contrôle et structures de données ;
- justifier la qualité des algorithmes ;
- et comparer les algorithmes à l'aide de l'étude de leur complexité.

Compétence majeure (académique) pour :

- *Concevoir des systèmes complexes sous contraintes à partir d'une spécification*
- *Concevoir un bloc fonctionnel unitaire au sein d'un système*
- *(Prérequis cursus cybersécurité- SecNumEdu)*

Compétence discriminante (métier) :

- Profil ingénieur en informatique
- VS utilisateur (même « avancé ») informatique
- Demandé/Vérifié par RH des organisations (Google, MS, et toute spécialisée)

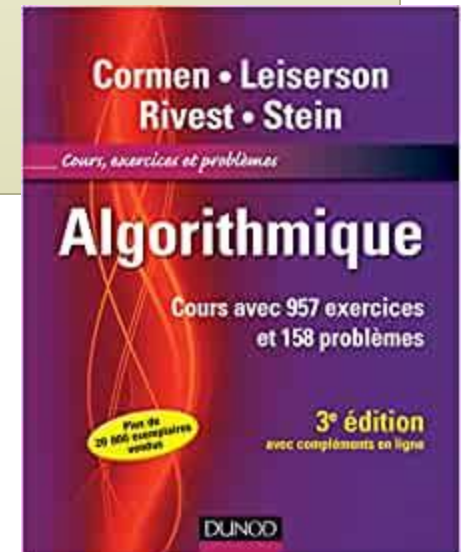
Méthodes de travail :

1. Méthode « traditionnelle » : cours/TD/TP

- Support : voir Chamilo (en cours de mise à jour)
 - Biblio : « Cormen » **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (third ed.)
 - Cours en ligne (vintage) : MIT

2. Spécifique à la matière : Travaux pratiques perso : **il faut programmer !**

- Les exercices qu'on voit pendant le cours ou les TD
- Langages C, Python, ou autre (java par ex)



Sommaire (partie YG)

1. Algorithmes

- Introduction
- Définitions
- Exemples
- Pseudo-code

2. Analyse des algorithmes

- Introduction
- Analyse du tri par insertion
- Analyse du pire des cas et du cas moyen
- Ordres de grandeur

3. Conception des algorithmes

- L'approche « diviser pour régner » (divide & conquer)
- Analyse des algorithmes basés sur « diviser pour régner »

4. Le pattern matching

- Les principaux algorithmes
- Les applications

5. Le parcours de graphes

(partie EB)

- Les fonctions de hash – application à la sécurité : tables arc-en-ciel
- La programmation dynamique
- Le algorithmes gloutons

Algorithmique :

1. Origine : 9^{ème} siècle (mathématicien *Al-Khwârizmî*)
2. Algorithmique contemporaine (depuis 20^{ème} siècle)
3. Maintenant et futur (?) : *informatique quantique*

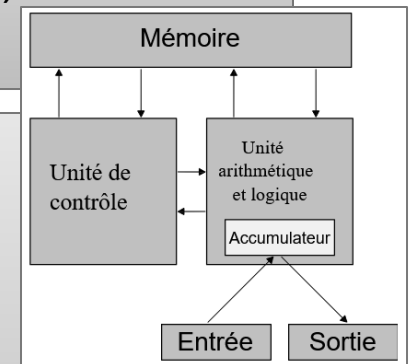
Algorithmique contemporaine :

1. Modèle théorique : **machine de Turing (universelle)**
2. Modèle physique : **architecture de Von Neumann (1945)**



Conséquences :

1. Limites théoriques : **calculabilité, décidabilité**¹
2. **Espaces discrets finis** (machine physique)
3. **Exécution séquentielle** (modèle Turing et modèle physique)



¹Problème de l'arrêt, analyse statique

Machine de Turing (*rappels ?*) :

- **Un ruban *infini*** avec des cases qui contiennent des symboles issus d'un alphabet fini donné. On peut faire avancer ou reculer le ruban dans la machine (analogie : mémoire)
- **Une tête de lecture écriture** pour lire et écrire sur le ruban
- **Un registre *fini d'états*** (similaire à un automate d'états) qui contient une table qui associe des états à des actions
 - Chaque ligne de cette table contient un « état », une « instruction » que sait faire la machine (par ex. avancer le ruban ou écrire dessus) et un « nouvel état »
 - Au début la machine commence à la première ligne. Par ex : état_init / avancer le ruban / passer à l'état 2/
 - La machine regarde dans le registre ce quelle doit faire dans l'état 2. Par ex : état 2/écrire « A » sur le ruban /passer à l'état x
 - etc,... jusqu'au moment où il n'y a plus d'action associé à l'état courant.

→ *C'est très similaire à ce qui se passe sur un ordi !*

Machine de Turing « **Universelle** » :

Principe :

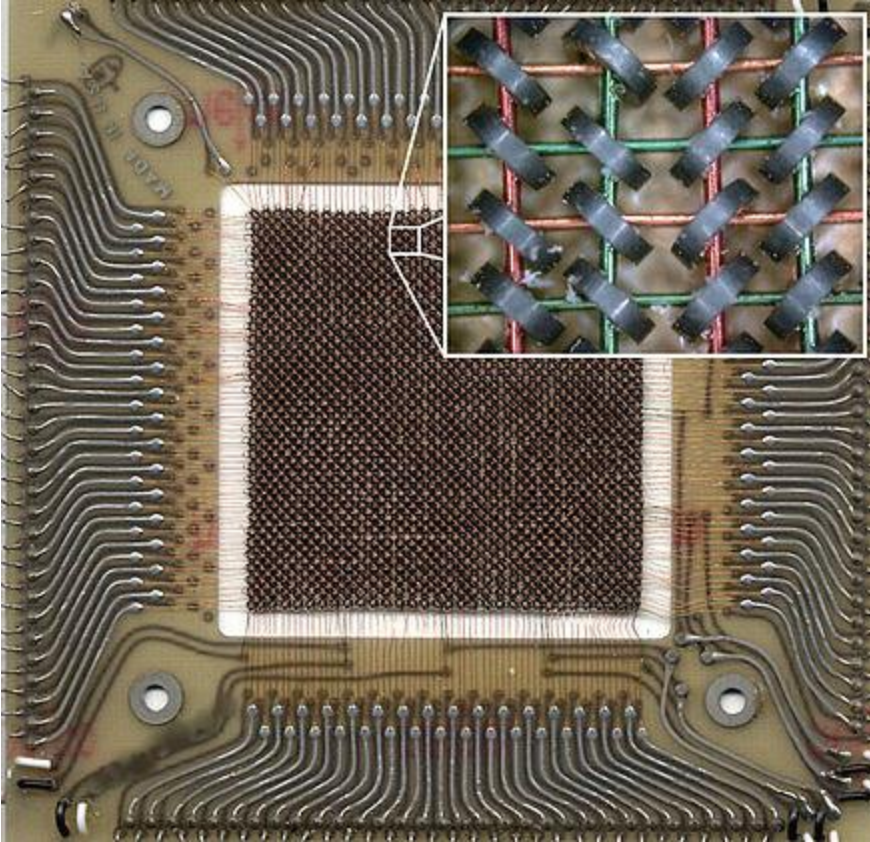
La machine « universelle » exécute un « programme » qui

- Prend en entrée n'importe quel autre « programme »
- Et l'exécute sur la machine de Turing

Sympa, non ?

A quoi cela fait-il penser ?

ARCHITECTURE DE VON NEUMANN- LA MÉMOIRE



Mémoire à tores (1980)



Processeur Intel 386 (1990)



Processeur SnapDragon (2021)



DDR-RAM 8Go (2020)

Architecture de Von Neumann (1945)

RAM (ou ROM) :

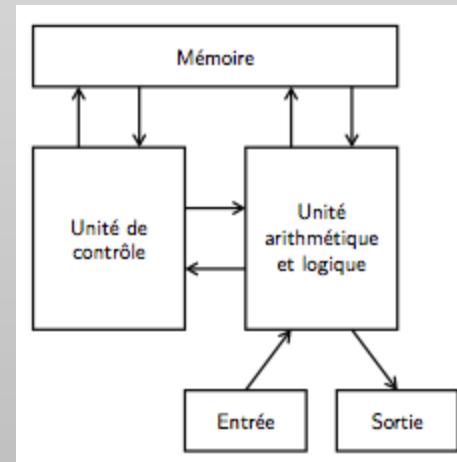
- suite de chiffres binaires, organisés en « octets »
 - Organisée en « mots » : processeur « 32 bits », « 64 bits »
 - « mot » : unité que le processeur peut traiter en une seule fois
- **Propriétés :**
 1. Pas de « sens » : un « mot » = une instruction ou une données (un nombre, un pixel,...)
 2. Inertie : pas de calcul
 3. Accès direct : tout « mot » possède une **adresse** à partir de laquelle on peut lire ou écrire dans la mémoire (en une seule opération)

-> vs machine de turing : accès à la mémoire en une seule opération élémentaire (impact sur le temps d'exécution)

Processeur :

2 parties :

- L'Unité de Contrôle (**UC**)
- L'Unité Arithmétique et Logique (**UAL**)



Architecture de Von Neumann – Exécution d'un programme

Il existe 2 registres spéciaux :

- **PC** : Program Counter
- **IR** : Registre d'instruction

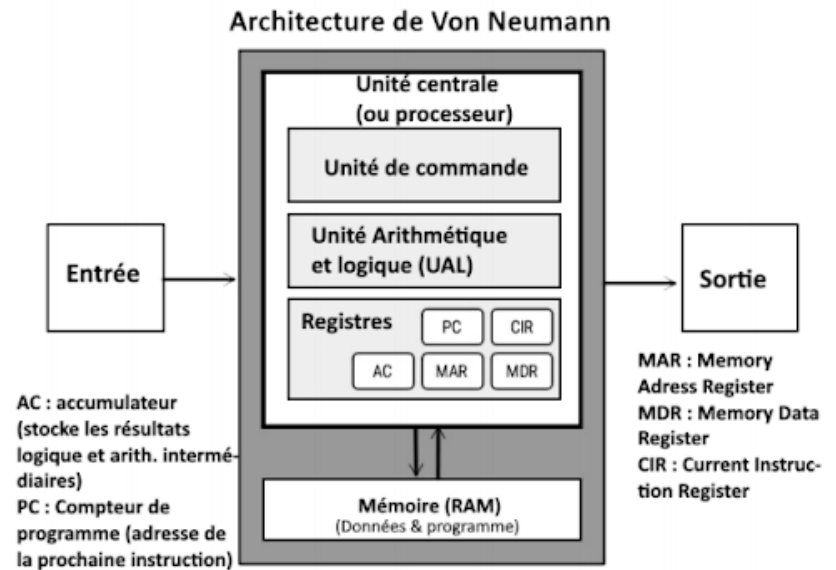
En boucle :

Etape 1 : lire instruction (aller à l'adresse stockée en **PC**, lire la **RAM** (le mot), charger le registre **IR**)

Etape 2 : incrémenter **PC** (lui rajouter 1, i.e. aller à l'instruction suivante)

Etape 3 : décoder instruction contenue dans **IR**

- Opération arithmétique ou logique
- Ou accès à la **RAM**
- Ou branchement pour modifier le flot d'instruction
 - Saut (conditionnel ou pas)
 - Qui modifie le registre **PC**



Faiblesse du modèle :

1. Exécution séquentielle

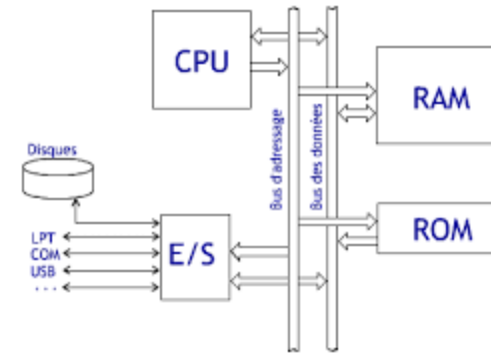
Parades : « multiprogrammation » ; *algorithms parallèles* ; *optimisations d'archi (canaux directs d'accès à la mémoire,...)* ; *informatique quantique*

1. Goulet d'étranglement (bus)

2. Robustesse : programme et données mélangés (sécurité sûreté)

Architecture de Von Neumann :

- **Modèle toujours valable depuis 1945**
- **Améliorations « modernes » :**
 - processeurs spécialisés pour les périphériques (exemples : cartes réseaux, processeurs graphiques)
 - Accès à la mémoire partagé ; caches
 - Plusieurs processeurs en parallèle (« core ») pour l'exécution des programmes
- **Modèle d'avenir ? Ordi quantique ?**



Exercice : dessins d'applications simples d'architectures de Von Neumann :

- Un PC
- Un Smartphone, un « serveur »
- Un système « embarqué »
- Un « objet » connecté

CS353 – Algorithmique et structure de données

03/02/2021 1^{ère} séance partie 2

Algorithmes

- Introduction
- Définitions
- Exemples
- Pseudo-code



*The Art of Computer Programming (TAOCP)
By Donald Knuth*



Sabotage d'une installation Orange à Crest (DL 16/02/2021)

1ère partie

- Définitions
- Exemples du tri par insertion et du décryptage de secrets par brute-force
- Pseudo-code
- **Temps d'exécution** : temps en fonction de la **taille** du problème

Définitions

Algorithme :

- Procédure de **calcul**¹ bien définie
 - Entrée : valeur ou un ensemble de valeurs
 - Sortie : valeur ou un ensemble de valeurs
- **Séquence**¹ finie d'étapes de **calcul** permettant de passer de la valeur d'entrée à la valeur de sortie

¹Turing

Exemple du **problème du tri** :

Entrée :

- Une suite de n éléments
 $\langle a_1, a_2, \dots, a_n \rangle$
- Une relation d'ordre notée « \leq »

attention ce n'est pas forcément une relation d'ordre sur des nombres : exemple jeu de cartes

Sortie :

- Une permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ de $\langle a_1, a_2, \dots, a_n \rangle$ telle que :
$$a'_1 \leq a'_2 \leq \dots \leq a'_n$$

Exemple : tri de nombres entiers (relation d'ordre sur les entiers naturels)

*Entrée : une « **instance** » du problème $\langle 31, 41, 59, 26, 41, 58 \rangle$*

Sortie : la séquence $\langle 26, 31, 41, 41, 58, 59 \rangle$

Définitions

Algorithme correct

- Pour chaque entrée il se termine avec la sortie correcte
- Il résout le problème posé

Algorithme incorrect

- Il s'arrête sur une réponse non souhaitée
- Il ne se termine pas

Difficultés :

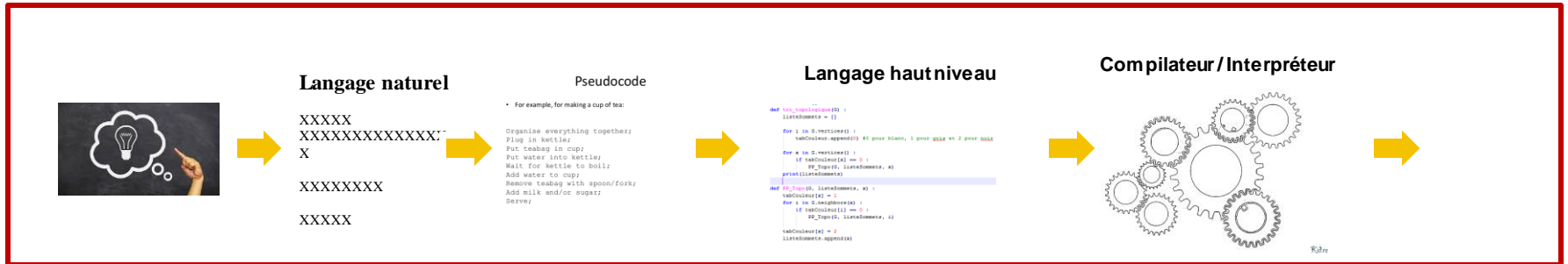
- Enoncé du problème (« spécifications »)
- « Preuves » que l'algorithme fonctionne

Question centrale pour les applications
qui nécessitent de la sécurité/sûreté !

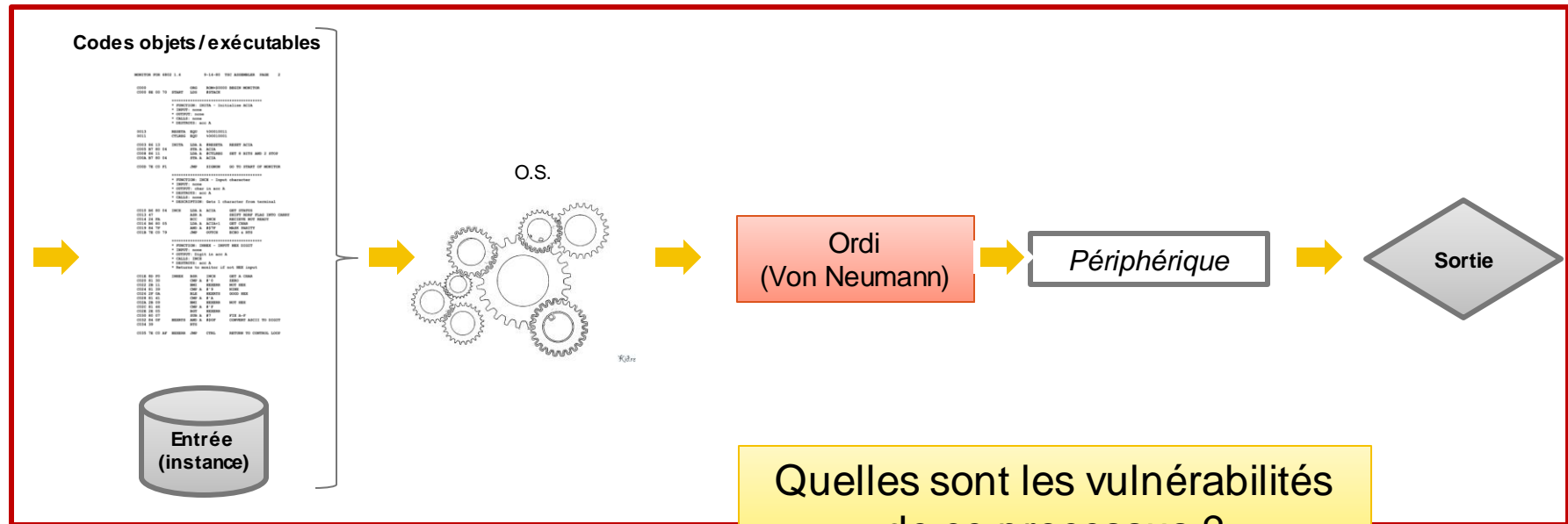
1ÈRE PARTIE

Schéma de la **conception** et de l'exécution d'un algorithme

1-Conception



2-Exécution



CS353 – Algorithmique et structure de données
2ème séance

Algorithmes

- Analyse d'un algorithme
- Exemple détaillé du tri par insertion
- Conception : méthode diviser pour régner
 - Exemple : Tri par fusion
 - Analyse du tri par fusion
- Base mathématiques – grandeur des fonctions



Why great care and consideration should be taken when selecting the proper password



2.1 Insertion sort



Figure 2.1 Sorting a hand of cards using insertion sort.

Exemple du tri d'une poignée de cartes à jouer

(au passage : relation d'ordre = ordre d'un jeu de carte standard)

Tri par insertion :

1. On part : d'une main vide et d'un talon de cartes
2. On prend une carte dans le tas
3. On l'insère « à la bonne place » dans la main en cours
4. <on itère jusqu'à épuisement du talon>

- Remarques, Questions ?
- Pourquoi cet algorithme « fonctionne » ?

Trace de l'algorithme de tri par insertion sur une instance donnée <6 entiers, relation d'ordre sur les entiers>

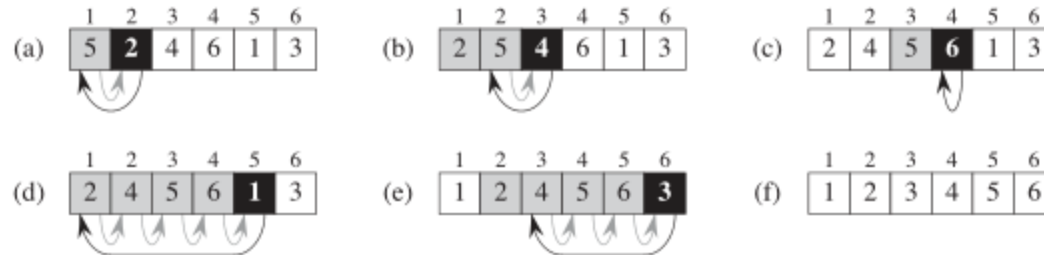


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. (a)–(e) The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. (f) The final sorted array.

- Remarques, Questions ?

Attention : Il faut savoir faire la trace d'un algorithme

Pseudo-code de l'algorithme de tri par insertion

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2     $key = A[j]$ 
3    // Insert  $A[j]$  into the sorted sequence  $A[1..j-1]$ .
4     $i = j - 1$ 
5    while  $i > 0$  and  $A[i] > key$ 
6       $A[i + 1] = A[i]$ 
7       $i = i - 1$ 
8     $A[i + 1] = key$ 
```

Remarques, Questions ?

Exercice

« Preuve » du programme :

(Idée)

- j est le n° de la carte en cours d'observation
- Quel est « l'invariant » du programme (propriété toujours vraie) ?
-

Preuve

- Montrer que l'invariant est vrai au début du programme ;
- Qu'il reste vrai pendant l'exécution (ici, juste une boucle while) ;
- Qu'il est vrai à la fin
- Conclusion ?

Conventions sur le Pseudo-code

- Indentation (et/ou marques de début/fin de blocs)
- Commentaires
- Structures de contrôle :
 - If then else ; while ; for
- Affectation : symbole \leftarrow ; différent du symbole = (opérateur booléen)
- Variables locales par défaut, sinon décrire explicitement la portée de la variable (mot-clé global ou autre)
- Tableaux :
 - indices commencent à 1 ;
 - description du tableau : T[1..N]
 - accès aux variables du tableau : T[i]
- Fonctions : par défaut, passage par valeur ; instruction return(v) pour renvoyer une (des) valeur(s)
- Booléens :
 - Opérateurs and / or / not()
 - Constantes Vrai Faux

Pseudo-code : destiné à des humains

⇒ Communication à privilégier : formes libres
Schémas, explications, couleurs, dessins,...

⇒ Evaluations (examens, exercices,...) :
Le pseudo-code sans explications (détaillées- pas de paraphrases) **ne sera pas évalué.**

Analyse d'un algorithme

- Estimer les ressources nécessaires à l'exécution :
 - Mémoire utilisée (Complexité « spatiale »)
 - Temps de calcul (Complexité « temporelle »)

Hypothèses : Machine RAM + processeur unique, exécution séquentielle

- Identifier l'algorithme « le plus efficace »
 - Question de l'existence d'une solution
 - ... et de son optimalité

Exemple : problème du tri

1. *Il existe des solutions : par ex. tri par insertion en $O(n^2)$ (quelle est sa complexité spatiale ?)*
2. *Est-elle la meilleure ?*
 - ↳ *Non : il existe d'autres algorithmes en $O(n \log(n))$, tri par fusion par exemple*
3. *Ce dernier est-il optimal ?*
 - ↳ *Oui : c'est démontré*

Conclusion

1. *En théorie, il est démontré que le problème du tri peut être résolu de façon optimale en $O(n \log(n))$ opérations.*
2. *En pratique :*
 - *Doit-on systématiquement utiliser un tri par fusion ?*
 - *Pourquoi ?*
 - *Limite de la théorie : ?*

Analyse d'un algorithme

« **Taille** » du problème :

↳ Dépend du problème ! Par exemple :

- Tri : nombre d'éléments de l'ensemble à trier
- Parcours de graphes : nombre de sommets, d'arêtes
- Recherche de motifs : taille de la chaîne recherchée, de la chaîne où se fait la recherche,....

« **Temps d'exécution** » de l'algorithme :

- Est fonction de la taille : on le note par défaut $T(n)$, n étant la taille du problème
- C'est le nombre d'opérations « élémentaires » exécutées (au sens de la machine) : ce que « sait faire » la machine dans un temps constant¹. Par exemple, une opération arithmétique ou logique, un saut d'adresse d'exécution

¹Ce temps est constant pour une machine et un environnement d'exécution donné.

« **Espace** » utilisé par l'algorithme (**complexité spatiale**):

- C'est la quantité de mémoire utilisée par l'algorithme en fonction de la **taille** du problème.
- Par exemple le tri par insertion n'utilise que l'espace nécessaire au tableau de données. Sa complexité spatiale est linéaire. (on parle d'un tri « en place »)

Analyse de l'algorithme de tri par insertion

INSERTION-SORT(A)

```
1  for  $j = 2$  to  $A.length$ 
2     $key = A[j]$ 
3    // Insert  $A[j]$  into the sorted
      sequence  $A[1..j-1]$ .
4     $i = j - 1$ 
5    while  $i > 0$  and  $A[i] > key$ 
6       $A[i+1] = A[i]$ 
7       $i = i - 1$ 
8     $A[i+1] = key$ 
```

Démarche :

- Évaluer le coût par ligne de code
- Faire la somme des lignes

Question :

- « coût de l'algo en «nombre de comparaisons »
- « **taille** » du problème : nombre d'éléments = taille du tableau = $A.length$)
- Ce que l'on compte, ce sont les opérations de comparaisons entre éléments suivant la relation d'ordre fournie en entrée. Pas les comparaisons entre entiers (boucle for par ex.)

Hypothèse :

- On considère que les opérations « élémentaires » (Turing) : affectation d'une variable, opérations arithmétiques, booléennes, ... ont un coût constant (sur une machine donnée)

Analyse détaillé de l'algorithme de tri par insertion

INSERTION-SORT(<i>A</i>)	<i>cost</i>	<i>times</i>
1 for <i>j</i> = 2 to <i>A.length</i>	c_1	n
2 $key = A[j]$	c_2	$n - 1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n - 1$
4 $i = j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i = i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i+1] = key$	c_8	$n - 1$

t_j : nombre de tests effectué pour tout j

Temps d'exécution


$$T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8 (n - 1) .$$


Cas le plus favorable :
(lequel est-ce ?)

Cas le plus favorable : Le tableau est déjà trié.

Le temps d'insertion $t_j = 1$ à chaque itération

$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\ + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n-1).$$


$$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8).$$



Cas le plus favorable :
 $T(n) = an + b$
Fonction linéaire avec a et b constantes

1ÈRE PARTIE

Analyse de l'algorithme de tri par insertion

Cas le pire: Le tableau est déjà trié mais dans l'ordre décroissant.

à chaque itération, on doit comparer l'élément en cours à tous ceux déjà triés (j)

Le temps d'insertion $t_j = j$ à chaque itération

If the array is in reverse sorted order—that is, in decreasing order—the worst case results. We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1 \dots j-1]$, and so $t_j = j$ for $j = 2, 3, \dots, n$. Noting that

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

(somme des entiers d'une suite arithmétique) $1+2+\dots+n$

Cas le pire :

$$T(n) = an^2 + bn + c$$

Fonction quadratique avec a, b et c constantes

Cas moyen:

Les valeurs des élément du tableau suivent une loi uniforme.

Combien faut-il de temps pour insérer $A[j]$ dans le sous-tableau $A[1, j-1]$?

- La moitié des éléments de $A[1, j-1]$ sont inférieurs (resp. supérieurs) à j .
- En moyenne : $j/2$ comparaisons

$$\Rightarrow T(j) = j/2$$

Mêmes calculs que dans le cas précédent (pire des cas)



Cas moyen :

$$T(n) = an^2 + bn + c$$

Fonction

quadratique avec a ,
 b et c constantes

Analyse de l'algorithme de tri par insertion

Conclusion de l'analyse :

L'algorithme de tri par insertion d'un ensemble de n éléments a un temps d'exécution $T(n)$ qui vaut :

- $O(n^2)$ dans le pire des cas et le cas moyen
- $O(n)$ dans le meilleur des cas
- Complexité spatiale : $\Theta(n)$

Remarques / Questions :

- Les notations $O()$ et Θ -qui seront vue en détail plus loin dans le cours, donnent «seulement » une notion asymptotique (borne sup.) du temps d'exécution ;
- Selon la répartition des données, le pire des cas peut se présenter « souvent » ;
- Avec un tri par insertion, le cas moyen (données tirées au hasard par exemple) est aussi mauvais que le pire des cas (asymptotiquement) = $O(n^2)$

Existe-t-il de meilleurs algorithmes de tri ? Si oui, quels sont leurs temps d'exécution ?

Lectures et exercices

A faire :

Relire le cours (livre)- Faire les exercices du livre (1,5/2h)

Pratique (4h) :

- Programmer un tri par insertion (sans copier sur internet)
- Faire des essais avec des jeux de données de taille et de contenus différents et mesurer le temps réel d'exécution sur votre machine.
- Présenter les résultats (graphiques c'est mieux avec analyse et commentaires)

Rendu (pris en compte dans le CC) :

- Code source+résultats+commentaires/explications ;
- Dans un seul fichier zip, nom_prénom dans le nom du fichier ;
- Sur chamilo, cours CS353, dossier travaux/Tri_Insertion
- Avant le 27/02 minuit
- *Le respect de ces consignes est évalué*
- *La copie, le plagiat sont sanctionnés*

Exercises

2.2-1

Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ -notation.

2.2-2

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A , and exchange it with $A[2]$. Continue in this manner for the first $n - 1$ elements of A . Write pseudocode for this algorithm, which is known as **selection sort**. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n - 1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ -notation.

2.2-3

Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ -notation? Justify your answers.

2.2-4

How can we modify almost any algorithm to have a good best-case running time?

CS353 – Algorithmique et structure de données
3ème séance

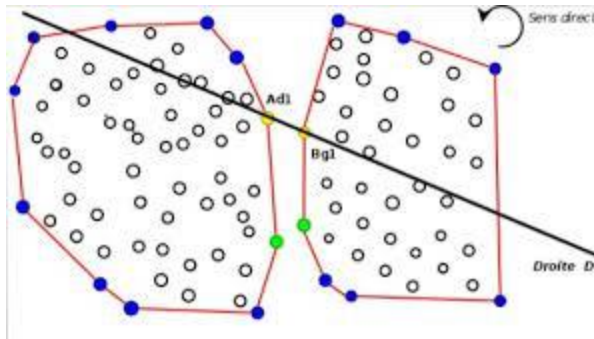
Algorithmes

- Conception : méthode diviser pour régner
 - Exemple : Tri par fusion
- Base mathématiques – grandeur des fonctions
- Analyse
 - du tri par fusion
 - Des algorithmes D&R (méthode générale)



Lorsque vous dites « le droit à la vie privée ne me préoccupe pas, parce que je n'ai rien à cacher », cela ne fait aucune différence avec le fait de dire « Je me moque du droit à la liberté d'expression parce que je n'ai rien à dire », ou « de la liberté de la presse parce que je n'ai rien à écrire ».

E. Snowden, Mémoires vives



Enveloppes Convexes. Voronoï - Delaunay

CONCEPTION DES ALGORITHMES

Approche « **Diviser pour régner** » (Divide & Conquer)

- C'est une méthode pour concevoir des algorithmes
- Elle ne fonctionne pas avec tous les problèmes
- Il y a d'autres méthodes (programmation dynamique, méthode gloutonne, algos génétiques,...)

Principe :

3 étapes :

1. **Diviser (D)** : le problème en sous-problèmes (2 ou plus)
2. **Résoudre (R)** : les sous-problèmes récursivement (jusqu'à une taille réduite où le problème a une solution triviale)
3. **Combiner (C)** : les solutions des sous-problèmes (issus de l'étape précédente)

Remarques :

- La conception n'est pas toujours facile
- Les performances de ces algorithmes sont plutôt bonnes (voir plus loin dans le cours)
- De nombreuses applications utilisent ces algorithmes (lorsqu'ils existent !)

Exemples d'algorithmes D&R :

- *Recherche dichotomique, Tri par fusion, Multiplication de Karatsuba, FFT, Enveloppe convexe,...*

CONCEPTION DES ALGORITHMES- DIVISER POUR RÉGNER

Exemple du **tri par fusion** :

Entrée : séquence de n éléments

Étapes :

1. (D) **Diviser** la séquence à trier en **deux** sous-séquences de $n/2$ éléments
2. (R) **Trier** les deux sous-séquences récurivement à l'aide du **tri par fusion**
3. (C) **Fusionner** les deux sous-séquences triées en une seule séquence triée

Sortie : séquence de n éléments triée

Les indices p et r représente la tranche du tableau à trier ; au premier appel, $p=1$ et $r=n$

Corps de l'algorithme

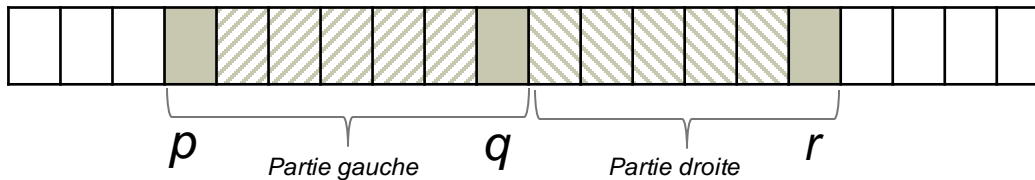
```
TRI-FUSION( $A, p, r$ )
1  si  $p < r$ 
2    alors  $q \leftarrow \lfloor (p + r) / 2 \rfloor$ 
3          TRI-FUSION( $A, p, q$ )
4          TRI-FUSION( $A, q + 1, r$ )
5          FUSION( $A, p, q, r$ )
```

Cas trivial (arrêt de la récursion, il n'y a plus rien à trier)

On coupe le tableau en 2 (la tranche en cours)

On trie récursivement la partie gauche puis la partie droite

On fusionne les deux sous-tableaux triés en un seul trié également



CONCEPTION DES ALGORITHMES- DIVISER POUR RÉGNER

TRI-FUSION(A, p, r)

```
1  si  $p < r$ 
2    alors  $q \leftarrow \lfloor (p+r)/2 \rfloor$ 
3          TRI-FUSION( $A, p, q$ )
4          TRI-FUSION( $A, q+1, r$ )
5          FUSION( $A, p, q, r$ )
```

Source : Cormen
Traces de l'algorithme d'une
exécution du tri par fusion

Pour trier toute la séquence $A = \langle A[1], A[2], \dots, A[n] \rangle$, on fait l'appel initial TRI-FUSION($A, 1, \text{longueur}[A]$) (ici aussi, $\text{longueur}[A] = n$). La figure 2.4 illustre le fonctionnement de la procédure, du bas vers le haut, quand n est une puissance de 2. L'algorithme consiste à fusionner des paires de séquences à 1 élément pour former des séquences triées de longueur 2, à fusionner des paires de séquences de longueur 2 pour former des séquences triées de longueur 4, etc. jusqu'à qu'il y ait fusion de deux séquences de longueur $n/2$ pour former la séquence triée définitive de longueur n .

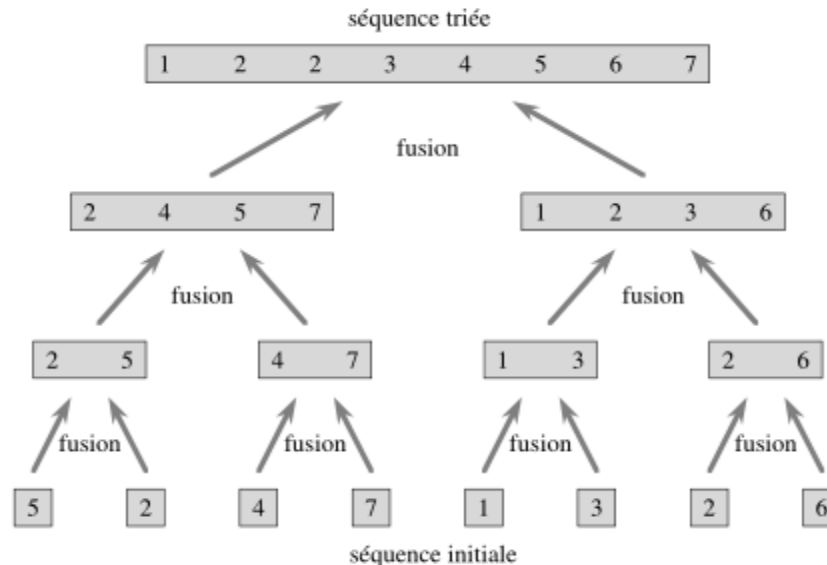


Figure 2.4 Le fonctionnement du tri par fusion sur le tableau $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. Les longueurs des séquences triées en cours de fusion augmentent à mesure que l'algorithme remonte du bas vers le haut.

Analyse des algorithmes D&R

Temps d'exécution – cas du tri par fusion

- $a=2$; $b=2$ (on divise l'ensemble en 2 parties et on résous 2 sous-problèmes)

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2T(n/2) + \Theta(n) & \text{si } n > 1. \end{cases}$$

$$\Rightarrow T(n) = \Theta(n \lg n),$$

D'après la « méthode générale » (voir plus bas)

- Pour des entrées assez grandes, le tri par fusion est nettement plus efficace que le tri par insertion
($n \lg(n) \ll n^2$)

CONCEPTION DES ALGORITHMES- DIVISER POUR RÉGNER

Exercice à la maison : Fonction fusion

1. *Ecrire l'algorithme de la procédure de fusion (pseudo-code) en minimisant le nombre d'emplacement mémoire nécessaire.*

Le temps d'exécution devra être en $\Theta(n)$, où $n = r - p + 1$

1. *Complexité de la fonction de fusion ?*

- Donner la trace complète de l'algorithme en indiquant le n° de l'étape concernée préfixé de la lettre (D/R/C) . d_4 correspond à la 4ème étape de l'algorithme qui est une division*

Indication :

- Fusion(A, p, q, r) où A est un tableau et p,q,r sont les indices tels que $p \leq q \leq r$*
- $A[p,..q]$ et $A[q+1,..r]$ sont préalablement triés*
- Fusion en un seul tableau qui remplace le sous-tableau $A[p,..r]$*

Comparaison d'efficacité

Tri d'un tableau d'un million de nombres

Tri par insertion

- Super calculateur
- Vitesse : 100 millions d'opérations par seconde
- Programme écrit en langage machine par le meilleur programmeur du monde
- Coût : $2n^2$ pour trier n nombres

$2(10^6)^2$ instructions / 10^8 ips = 20.000 secondes \approx 5,56 heures

CONCEPTION DES ALGORITHMES- DIVISER POUR RÉGNER

Comparaison d'efficacité

Tri d'un tableau d'un million de nombres

Tri par fusion

- PC moyen
- Vitesse : 1 million d'opérations par seconde
- Programme écrit en langage de haut niveau par un programmeur moyen
- Coût : $50 \cdot n \cdot \lg(n)$ pour trier n nombres

$50 \cdot 10^6 \cdot \lg(10^6)$ instructions / 10^6 ips = 1.000 secondes \approx 16,67 minutes

L'algorithme de tri par fusion reste 20 fois plus rapide, même dans des conditions de test défavorables

Bases mathématiques

- Grandeur des fonctions
- Théorème (méthode) général(e)

Bases mathématiques - Grandeur des fonctions

Chapter 3 overview

- A way to describe behavior of functions *in the limit*. We're studying *asymptotic* efficiency.
- Describe *growth* of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare "sizes" of functions:

$$O \approx \leq$$

$$\Omega \approx \geq$$

$$\Theta \approx =$$

$$o \approx <$$

$$\omega \approx >$$

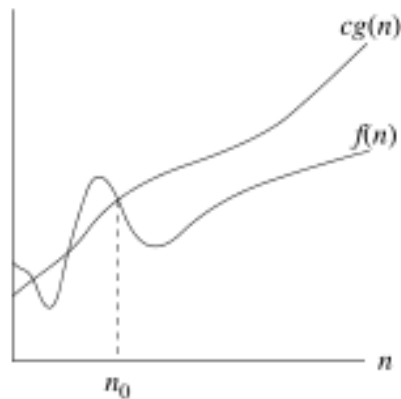
Bases mathématiques - Grandeur des fonctions

Notation O (« grand O »)- Définition

Asymptotic notation

O -notation

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$



$g(n)$ is an *asymptotic upper bound* for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$ (will precisely explain this soon).

Bases mathématiques - Grandeur des fonctions

Notation O (« grand O »)- Exemples

Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

$$n^2$$

$$n^2 + n$$

$$n^2 + 1000n$$

$$1000n^2 + 1000n$$

Also,

$$n$$

$$n/1000$$

$$n^{1.99999}$$

$$n^2 / \lg \lg \lg n$$

Par convention, on note
 $T(N) = O(g(N))$,
Au lieu de $T(N) \in O(N)$

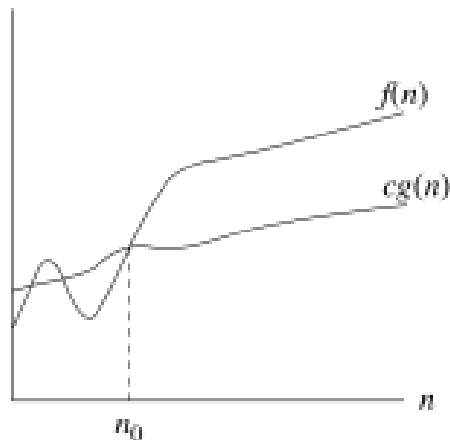
Trouvez les constantes !

Bases mathématiques - Grandeur des fonctions

Notation Ω (« grand Oméga »)- Définition

Ω -notation

$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}.$



$g(n)$ is an *asymptotic lower bound* for $f(n)$.

Bases mathématiques - Grandeur des fonctions

Notation Ω (« grand Oméga »)- Exemples

Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

$$n^2$$

$$n^2 + n$$

$$n^2 - n$$

$$1000n^2 + 1000n$$

$$1000n^2 - 1000n$$

Also,

$$n^3$$

$$n^{2.00001}$$

$$n^2 \lg \lg \lg n$$

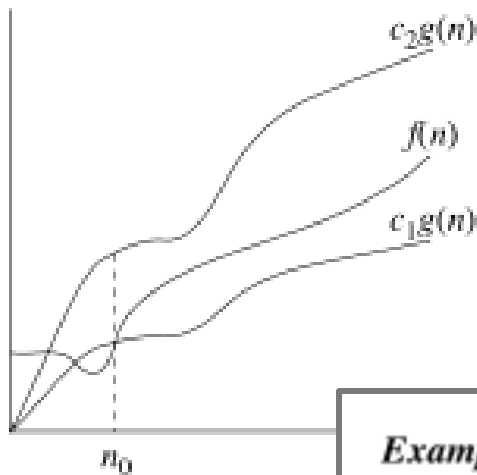
$$2^{2^n}$$

Trouvez les constantes !

Notation Θ (« Théta »)- Définition

Θ -notation

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$.



$g(n)$ is an *asymptote*

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Theorem

$f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$.

Bases mathématiques - Grandeur des fonctions

Lectures et exercices
Cormen

Analyse des algorithmes D&R – CAS GENERAL

Temps d'exécution – **cas général**

- $T(n)$: temps d'exécution global pour un problème de taille n
- Division en a sous-problèmes de tailles b
- $D(n)$: temps pour diviser le problème en sous-problèmes
- $C(n)$: temps pour combiner les solutions en une solution finale

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{sinon.} \end{cases} \quad \leftarrow \text{Cas trivial}$$

CONCEPTION DES ALGORITHMES

Analyse des algorithmes D/R

Exemple du tri par fusion- rappels

TRI-FUSION(A, p, r)

```
1  si  $p < r$ 
2    alors  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3          TRI-FUSION( $A, p, q$ )
4          TRI-FUSION( $A, q + 1, r$ )
5          FUSION( $A, p, q, r$ )
```

Pour trier toute la séquence $A = \langle A[1], A[2], \dots, A[n] \rangle$, on fait l'appel initial TRI-FUSION($A, 1, \text{longueur}[A]$) (ici aussi, $\text{longueur}[A] = n$). La figure 2.4 illustre le fonctionnement de la procédure, du bas vers le haut, quand n est une puissance de 2. L'algorithme consiste à fusionner des paires de séquences à 1 élément pour former des séquences triées de longueur 2, à fusionner des paires de séquences de longueur 2 pour former des séquences triées de longueur 4, etc. jusqu'à qu'il y ait fusion de deux séquences de longueur $n/2$ pour former la séquence triée définitive de longueur n .

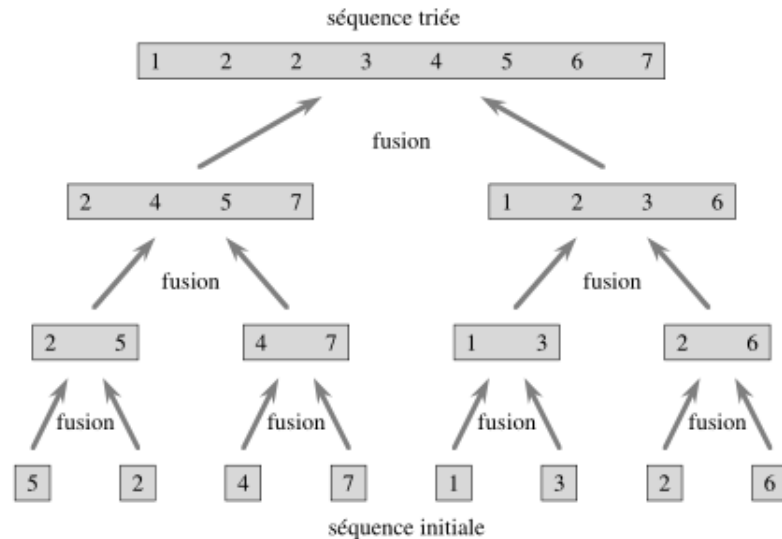


Figure 2.4 Le fonctionnement du tri par fusion sur le tableau $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. Les longueurs des séquences triées en cours de fusion augmentent à mesure que l'algorithme remonte du bas vers le haut.

CONCEPTION DES ALGORITHMES

Analyse des algorithmes D/R

Exemple du tri par fusion

TRI-FUSION(A, p, r)

```
1  si  $p < r$ 
2    alors  $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
3          TRI-FUSION( $A, p, q$ )
4          TRI-FUSION( $A, q + 1, r$ )
5          FUSION( $A, p, q, r$ )
```

Diviser : L'étape diviser se contente de calculer le milieu du sous-tableau, ce qui consomme un temps constant. Donc $D(n) = \Theta(1)$.

Régner : On résout récursivement deux sous-problèmes, chacun ayant la taille $n/2$, ce qui contribue pour $2T(n/2)$ au temps d'exécution.

Combiner : Nous avons déjà noté que la procédure FUSION sur un sous-tableau à n éléments prenait un temps $\Theta(n)$, de sorte que $C(n) = \Theta(n)$.

Quand on ajoute les fonctions $D(n)$ et $C(n)$ pour l'analyse du tri par fusion, on ajoute une fonction qui est $\Theta(n)$ et une fonction qui est $\Theta(1)$. Cette somme est une fonction linéaire de n , à savoir $\Theta(n)$. L'ajouter au terme $2T(n/2)$ de l'étape régner donne la récurrence pour $T(n)$, temps d'exécution du tri par fusion dans le cas le plus défavorable :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2T(n/2) + \Theta(n) & \text{si } n > 1. \end{cases} \quad (2.1)$$

CONCEPTION DES ALGORITHMES

Analyse des algorithmes D/R

Exemple du tri par fusion

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2T(n/2) + \Theta(n) & \text{si } n > 1. \end{cases}$$

Dans ce cas :

- $a=2$
- $b=2$,
- $C(n)+D(n) = \Theta(n)$

La « **méthode générale** » nous permet d'affirmer que dans ce cas :

$$T(n) = n \lg_2(n)$$

Pour comprendre intuitivement ce calcul, posons :

$$T(n) = \begin{cases} c & \text{si } n = 1, \\ 2T(n/2) + cn & \text{si } n > 1, \end{cases} \quad (2.2)$$

la constante c représentant le temps requis pour résoudre des problèmes de taille 1, ainsi que le temps par élément de tableau des étapes diviser et combiner. ⁽⁸⁾

Analyse du tri par fusion

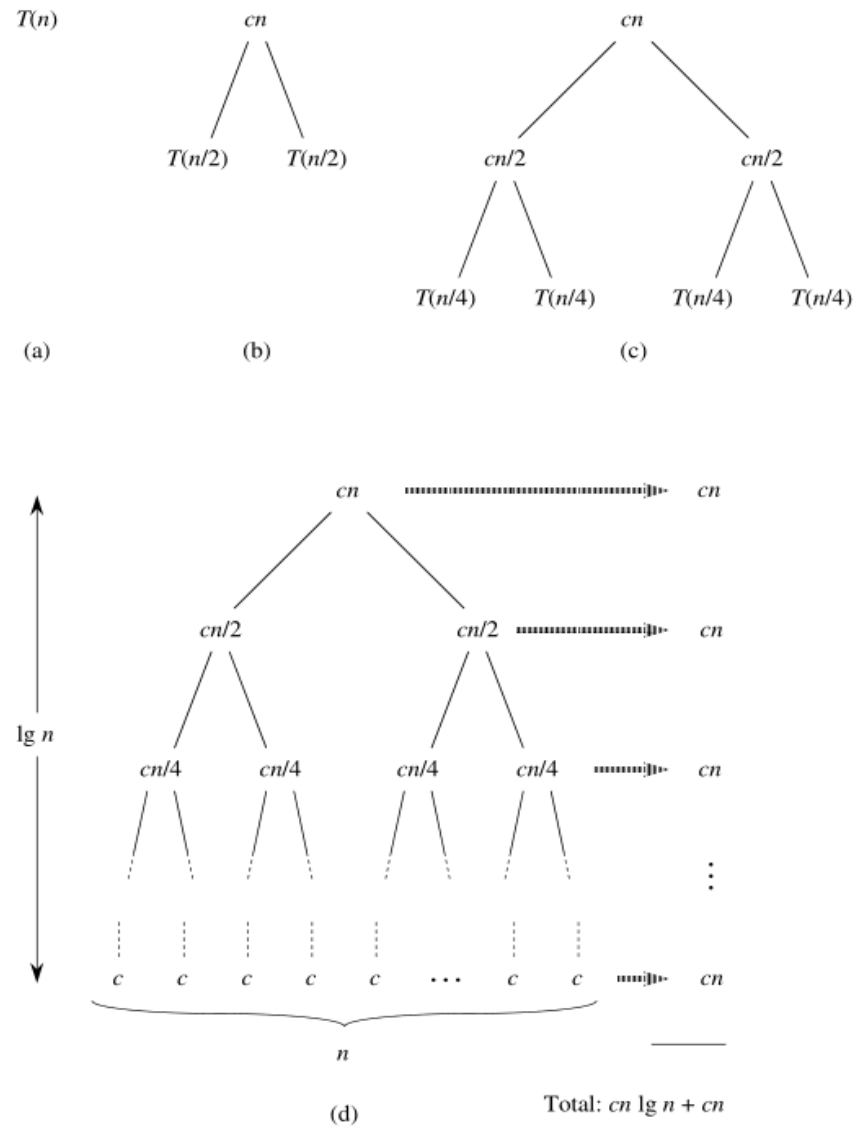


Figure 2.5 La construction d'un arbre de récursivité pour la récurrence $T(n) = 2T(n/2) + cn$. La partie (a) montre $T(n)$, progressivement développé en (b)–(d) pour former l'arbre de récursivité. L'arbre complet, en partie (d), a $\lg n + 1$ niveaux (c'est-à-dire, il a une hauteur $\lg n$ comme indiquée), sachant que chaque niveau contribue pour un coût total de cn . Le coût global est donc $cn \lg n + cn$, c'est à dire $\Theta(n \lg n)$.

Dans les exemples, pour presque tous les algorithmes $a=b=2$;

Existe-t-il des algorithmes avec des stratégies différentes ?

Oui, bien sûr :

Dans l'algorithme de **Karatsuba** (multiplication de deux nombres de longueur n) :

- $a=3$ et $b=2$ (on appelle récursivement trois fois la fonction sur un ensemble de taille $n/2$)
- Le calcul (donné par la formule magique) du temps d'exécution donne :

$O(n^{\log_2(3)}) \approx O(n^{1,585})$ au lieu de $O(n^2)$ pour la méthode naïve

À approfondir à la maison

La méthode générale donne une « recette » pour résoudre les récurrences de la forme

$$T(n) = aT(n/b) + f(n) , \quad (4.5)$$

où $a \geq 1$ et $b > 1$ sont des constantes, et $f(n)$ une fonction asymptotiquement positive. La méthode générale oblige à traiter trois cas de figure, mais permet de déterminer la solution de nombreuses récurrences assez facilement.

Théorème 4.1 (Théorème général.) *Soient $a \geq 1$ et $b > 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ définie pour les entiers non négatifs par la récurrence*

$$T(n) = aT(n/b) + f(n) ,$$

où l'on interprète n/b comme signifiant $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. $T(n)$ peut alors être bornée asymptotiquement de la façon suivante.

- 1) Si $f(n) = O(n^{\log_b a - \varepsilon})$ pour une certaine constante $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.*
- 2) Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \lg n)$.*
- 3) Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour une certaine constante $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour une certaine constante $c < 1$ et pour tout n suffisamment grand, alors $T(n) = \Theta(f(n))$.*

Cela paraît bien compliqué.

Voyons :

- quelques exemples d'utilisation
- et une explication (intuitive) de ce résultat

CONCEPTION DES ALGORITHMES

Analyse des algorithmes D/R

Généralisation

Exemples d'utilisation de la méthode générale

Comme premier exemple, considérons

$$T(n) = 9T(n/3) + n.$$

Pour cette récurrence, on a $a = 9$, $b = 3$ et $f(n) = n$; on a donc $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Puisque $f(n) = O(n^{\log_3 9 - \varepsilon})$, où $\varepsilon = 1$, on peut appliquer le cas 1 du théorème général et dire que la solution est $T(n) = \Theta(n^2)$.

Considérons à présent

$$T(n) = T(2n/3) + 1,$$

où $a = 1$, $b = 3/2$, $f(n) = 1$ et $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. On est dans le cas 2 puisque $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, et donc la solution de la récurrence est $T(n) = \Theta(\lg n)$.

Pour la récurrence

$$T(n) = 3T(n/4) + n \lg n,$$

on a $a = 3$, $b = 4$, $f(n) = n \lg n$ et $n^{\log_b a} = n^{\log_4 3} = O(n^{0,793})$. Puisque $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$, où $\varepsilon \approx 0,2$, on est dans le cas 3 si l'on peut montrer que la condition de régularité est vraie pour $f(n)$. Pour n suffisamment grand,

$$af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n),$$

avec $c = 3/4$. Par conséquent, d'après le cas 3, la solution de la récurrence est $T(n) = \Theta(n \lg n)$.

Faire les exercices du chap. 4

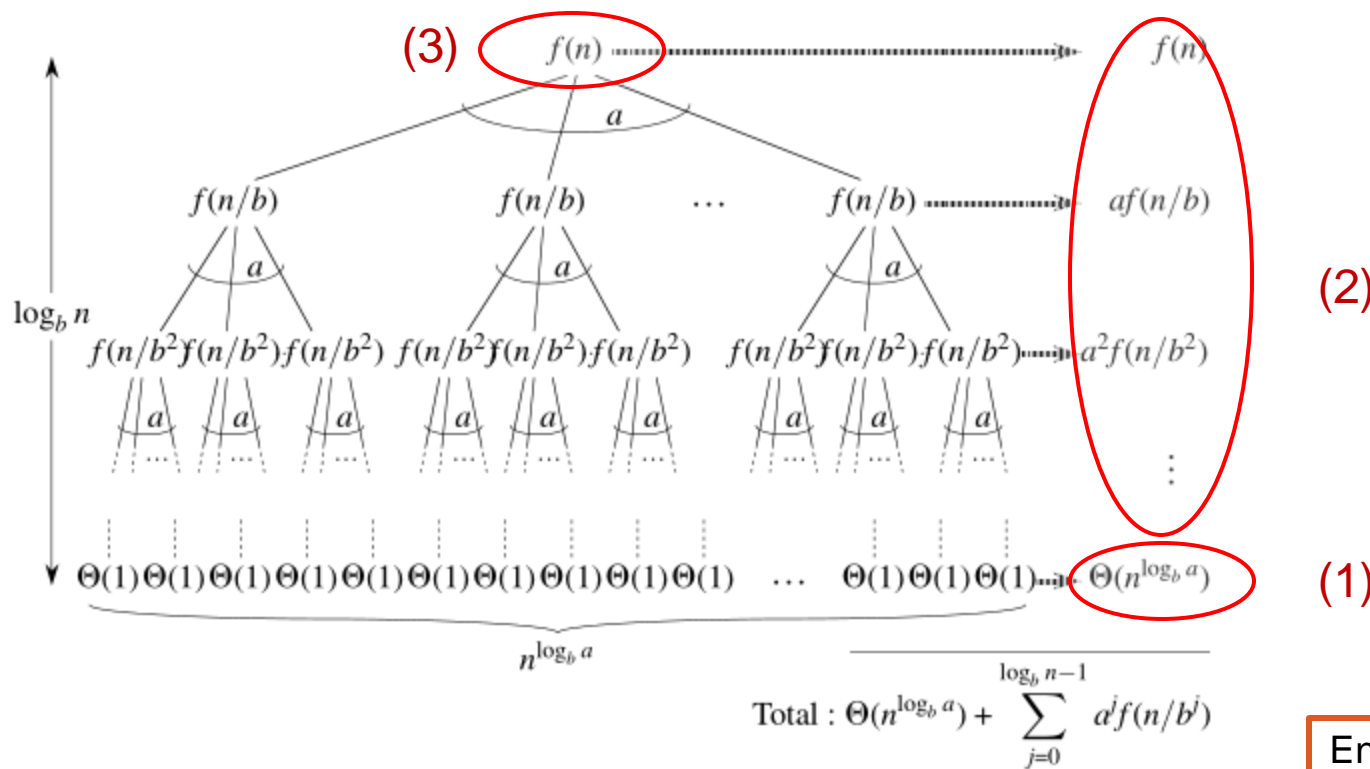


Figure 4.3 L'arbre récursif généré par $T(n) = aT(n/b) + f(n)$. L'arbre est un arbre complet d'arité a ayant $n^{\log_b a}$ feuilles et une hauteur $\log_b n$. Le coût de chaque niveau est montré à droite, et leur somme est donnée dans l'équation (4.6).

En pratique, pour trouver le bon cas du théorème, on compare $f(n)$ avec $n^{\log_b a}$.

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j). \quad (4.6)$$

démonstration formelle et « finesses » sur Cormen chap. 4

En termes d'arbre récursif, les trois cas du théorème général correspondent aux cas où le coût total de l'arbre est (1) dominé par les coûts des feuilles, (2) uniformément distribué à travers les niveaux de l'arbre ou (3) dominé par le coût de la racine.

CONCEPTION DES ALGORITHMES

Analyse des algorithmes D/R- Synthèse

Diviser pour régner :

- Une méthode de conception d'algorithmes
- Qui peut résoudre des problèmes variés
- Parfois
 - Pas évidente à mettre en œuvre
 - Pas forcément optimale
- Une recette de cuisine (« **méthode générale** ») pour déterminer la complexité des algorithmes D/R

Pour approfondir :

- Tri « **Quick-Sort** »
 - Intérêt pédagogique : approfondissement des techniques de tri, de l'analyse et de l'optimalité
- Multiplication de **Karatsuba**
 - Cas original d'un algorithme D/R
 - Sujet majeur (math) et applications (info)
- **Enveloppe convexe (convex hull)**
 - Autre cas original d'un algorithme D/R
 - Découverte de l'algorithmique graphique

CONCEPTION DES ALGORITHMES

Synthèse derniers cours

Théorie :

- Notations O , Ω , Θ ; majorations minorations de fonctions (éventuellement définies par récurrence)
- « Méthode générale » pour les algorithmes conçus en D/R
- Algorithmes de **tris** : par insertion, par fusion. Conception et analyse.

Extensions et aspects pratiques :

- Tris :
 - Quicksort
 - Heapsort,...

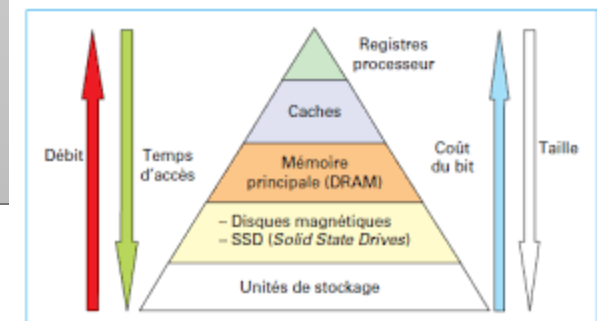
Implémentations, performances :

- **Timsort** (Tim Peters)

Algorithme hybride insertion/fusion, avec de multiples optimisations

(parmi les nombreuses choses à regarder, la question de la « hiérarchie mémoire »)

- Exemples de problèmes « bien » résolus » par D/R:
 - Court exemple : **Enveloppe convexe** (au tableau)



Lectures et exercices

A faire : (exercice1)

Pratique (4h) :

- Programmer un tri par insertion (sans copier sur internet)
- Faire des essais avec des jeux de données de taille et de contenus différents et mesurer le temps réel d'exécution sur votre machine.
- Présenter les résultats (graphiques c'est mieux), avec analyse et commentaires

Rendu (pris en compte dans le CC) :

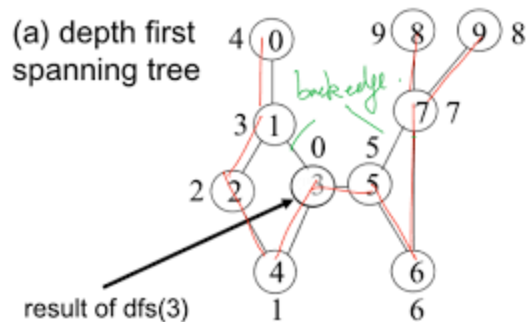
- Code source+résultats+commentaires/explications ;
- Dans un seul fichier zip, nom_prénom dans le nom du fichier ;
- Sur chamilo, cours CS353, dossier travaux/Tri_Insertion
- Avant le 27/02 minuit
- Le respect de ces consignes est évalué
- La copie, le plagiat sont sanctionnés

A faire : extension

- Programmer un tri par fusion (sans copier sur internet)
- Faire des essais avec des jeux de données de taille et de contenus différents et mesurer le temps réel d'exécution sur votre machine.
- Comparer avec les résultats obtenus avec un tri par insertion (graphiques c'est mieux), avec analyse et commentaires)
- Facultatif :
 - Idem avec un tri Timsort (issu de la librairie standard)
 - Idem sur des plateformes différentes (par ex. Linux / Windows)
- Délai supplémentaire : Avant le 06/03 minuit

CS353 – Algorithmique et structure de données Graphes

- Mini-cours sur les parcours de graphes
- Application : Le **tri topologique**



Exploration d'un labyrinthe
(algorithme de Trémaux)

Algorithmes de parcours de graphes

Terminologie & usages :

- Graphe orienté : nœuds, arcs, successeur, prédécesseur, arborescence, source, puit,...
- Graphe non orienté : sommets, arêtes, voisins, arbre, racine,...
- Temps d'exécution : en général taille fonction du nombre de sommets (N ou S) et du nombre d'arêtes (M ou A)

Modèles / Implémentations :

- Listes d'adjacences : Tableau (fixe) des sommets (nœuds) ; chaque élément du tableau pointe vers la liste de ses voisins (successeurs)

- Matrice d'incidence: Tableau statique 2D ; $M(x,y) = 0$ si pas d'arc, 1 s'il y a un arc entre x et y

Figure 11. Exemple de graphe orienté valué et de sa liste d'adjacence.

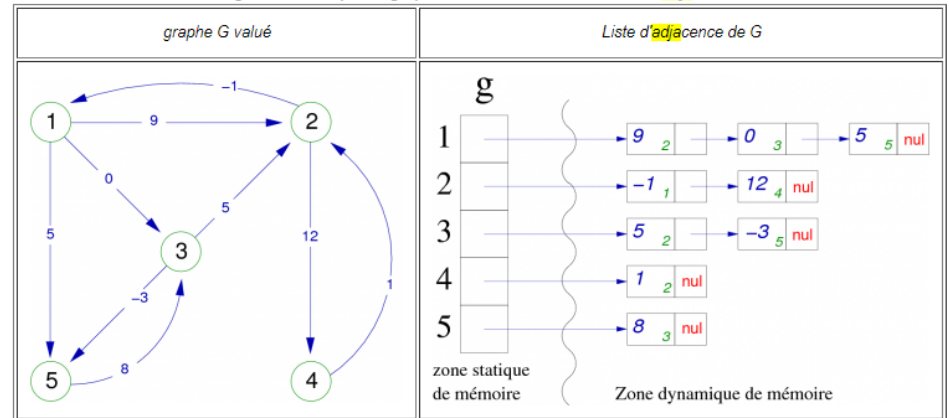
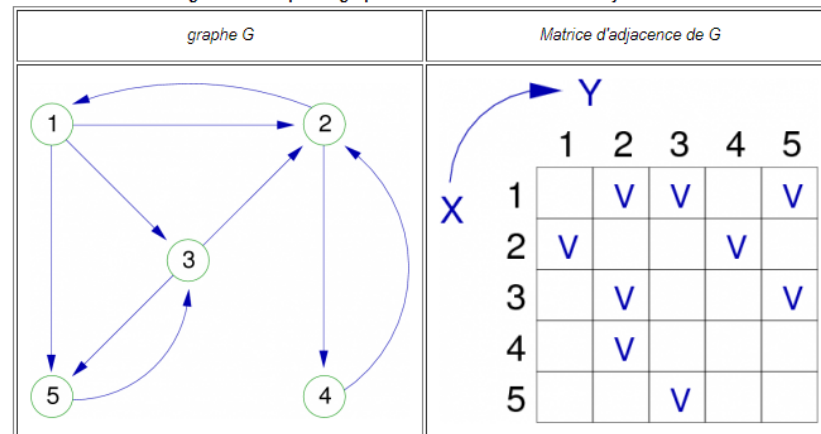


Figure 9. Exemple de graphe orienté et de sa matrice d'adjacence.



Algorithmes de parcours de graphes

Modèles / Implémentations :

- Liste d'adjacences :
 - Complexité spatiale $\Theta(S+A)$
 - Liste des voisins d'un sommet x : $\Theta(\text{degré}(x))$
 - Existence d'une arête (x,y) : $O(\text{degré}(x))$
- Matrice d'incidence :
 - Complexité spatiale $\Theta(S^2)$
 - Liste des voisins d'un sommet x : $\Theta(S)$
 - Existence d'une arête (x,y) : $\Theta(1)$

Attention : l'implémentation peut changer le temps d'exécution !

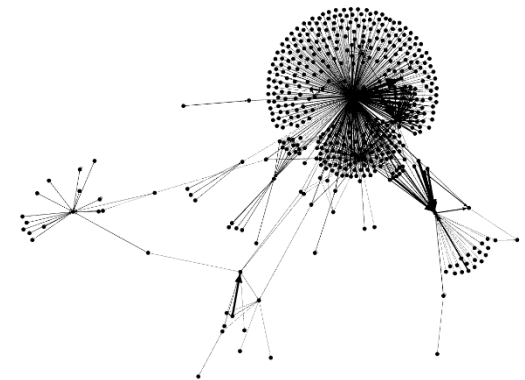
Ex : « tant qu'il y a des arêtes »...

Algorithmes généraux de parcours :

- DFS : **Deep**-First-Search
- BFS : **Breadth**-First-Search

Permettent d'explorer tout un graphe (sommets et arêtes)

Applications : très nombreuses



Algorithmes de parcours de graphes - DFS

DFS Deep-First-Search

(Cormen)

La stratégie suivie par un parcours en profondeur est, comme son nom l'indique, de descendre plus « profondément » dans le graphe chaque fois que c'est possible. Lors d'un parcours en profondeur, les arcs sont explorés à partir du sommet v découvert le plus récemment et dont on n'a pas encore exploré tous les arcs qui en partent. Lorsque tous les arcs de v ont été explorés, l'algorithme « revient en arrière » pour explorer les arcs qui partent du sommet à partir duquel v a été découvert. Ce processus se répète jusqu'à ce que tous les sommets accessibles à partir du sommet origine initial aient été découverts. S'il reste des sommets non découverts, on en choisit un qui servira de nouvelle origine, et le parcours reprend à partir de cette origine. Le processus complet est répété jusqu'à ce que tous les sommets aient été découverts.

Principe : les sommets sont coloriés pendant le parcours pour indiquer leur état. Chaque sommet est initialement **blanc**, puis **gris** quand il est découvert pendant le parcours, et enfin **noirci** en fin de traitement, c'est-à-dire quand sa liste d'adjacence a été complètement examinée.

Fonction PP(G) : Parcours en Profondeur du graphe G

qui utilise une fonction récursive :

VISITER-PP(u) qui explore tous les voisins d'un sommet u

Algorithmes de parcours de graphes - DFS

(BFS : à la maison)

DFS Deep-First-Search

PP(G)

pour chaque sommet $u \in S[G]$
faire $\text{couleur}[u] \leftarrow \text{BLANC}$

pour chaque sommet $u \in S[G]$
faire si $\text{couleur}[u] = \text{BLANC}$
alors VISITER-PP(u)

VISITER-PP(u)
 $\text{couleur}[u] \leftarrow \text{GRIS} \triangleright$ sommet blanc u vient d'être découvert.

pour chaque $v \in \text{Adj}[u] \triangleright$ Exploration de l'arc (u, v) .
faire si $\text{couleur}[v] = \text{BLANC}$
alors
 VISITER-PP(v)
 $\text{couleur}[u] \leftarrow \text{NOIR} \triangleright$ noircir u , car on en a fini avec lui.

Analyse :

- Initialisation : $O(S)$
- On parcourt tous les arcs une seule fois, total : $\Theta(A)$
 (voir justifications [Cormen])
- Complexité finale : $O(S+A)$

Algorithmes de parcours de graphes - BFS

Deuxième algorithme simple de parcours d'un graphe

- A la base de nombreux algorithmes importants sur les graphes :
 - algorithme de **Dijkstra** pour calculer les plus courts chemins à origine unique
 - algorithme de **Prim** pour trouver l'arbre couvrant minimum
- L'algorithme fonctionne aussi bien sur les graphes orientés que sur les graphes non orientés.

Principe :

Étant donné un graphe $G = (S, A)$ et un sommet origine s ,

- Exploration systématique des arcs de G pour « découvrir » tous les sommets accessibles depuis s .
- Au « passage » :
 - calcul de la distance (plus petit nombre d'arcs) entre s et tous les sommets accessibles.
 - construction d'une « arborescence de parcours en largeur » de racine s , qui contient tous les sommets accessibles depuis s . Pour tout sommet v accessible depuis s , le chemin reliant s à v dans l'arborescence de parcours en largeur correspond à un « plus court chemin » de s vers v dans G , autrement dit un chemin contenant le plus petit nombre d'arcs.

Algorithmes de parcours de graphes - BFS

Breadth-First-Search

Code de l'algorithme BFS :

- On commence par explorer tous les voisins de la source (ceux qui sont à une distance = 1)
- Puis tous ceux qui sont à une distance = 2, c'est-à-dire les voisins de tous les voisins de la source, etc. jusqu'à ce que tous les sommets soient explorés
- Dans le cas de l'algorithme **DFS**, la récursivité utilisait une **Pile**
- Ici, nécessité de gérer une **File** (qui à un instant t va contenir tous les sommets à une distance x de la source)

```

ParcoursLargeur(Graphe G, Sommet s):
    f = CreerFile();
    f.enfiler(s);
    marquer(s);
    tant que la file est non vide
        s = f.defiler();
        afficher(s);
        pour tout voisin t de s dans G
            si t non marqué
                f.enfiler(t);
                marquer(t);

```

Version « Wikipédia »

```

PL(G, s)
1  pour chaque sommet  $u \in S[G] - \{s\}$ 
2      faire couleur[u] ← BLANC
3       $d[u] \leftarrow \infty$ 
4       $\pi[u] \leftarrow \text{NIL}$ 
5  couleur[s] ← GRIS
6  d[s] ← 0
7   $\pi[s] \leftarrow \text{NIL}$ 
8   $F \leftarrow \{s\}$ 
9  tant que  $F \neq \emptyset$ 
10     faire  $u \leftarrow \text{tête}[F]$ 
11     pour chaque  $v \in \text{Adj}[u]$ 
12         faire si couleur[v] = BLANC
13             alors couleur[v] ← GRIS
14                  $d[v] \leftarrow d[u] + 1$ 
15                  $\pi[v] \leftarrow u$ 
16                 ENFILE(F, v)
17                 DÉFILE(F)
18     couleur[u] ← NOIR

```

Version « Cormen »

Algorithmes de parcours de graphes - BFS

```

PL( $G, s$ )
1  pour chaque sommet  $u \in S[G] - \{s\}$ 
2    faire  $\text{couleur}[u] \leftarrow \text{BLANC}$ 
3     $d[u] \leftarrow \infty$ 
4     $\pi[u] \leftarrow \text{NIL}$ 
5   $\text{couleur}[s] \leftarrow \text{GRIS}$ 
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow \text{NIL}$ 
8   $F \leftarrow \{s\}$ 
9  tant que  $F \neq \emptyset$ 
10   faire  $u \leftarrow \text{tête}[F]$ 
11     pour chaque  $v \in \text{Adj}[u]$ 
12       faire si  $\text{couleur}[v] = \text{BLANC}$ 
13         alors  $\text{couleur}[v] \leftarrow \text{GRIS}$ 
14            $d[v] \leftarrow d[u] + 1$ 
15            $\pi[v] \leftarrow u$ 
16            $\text{ENFILE}(F, v)$ 
17            $\text{DÉFILE}(F)$ 
18    $\text{couleur}[u] \leftarrow \text{NOIR}$ 

```

Version « Cormen »

- Utilisation de listes d'adjacences
- Utilisation de « couleurs », comme dans DFS
- Blanc : pas exploré, Gris : en cours d'exploration, Noir : exploré
- Mise à jour de deux tableaux (de taille N) :
 - d : distance de chaque sommet depuis la source
 - π : prédécesseur de chaque sommet dans l'arborescence de parcours

Faire une trace de l'algorithme sur des exemples

Analyse :

- Initialisation : $O(S)$
- Puis, chaque sommet n'est enfilé une seule fois (ligne 13 qui le garantit)
- Les opérations sur la file et les affectations des tableaux sont en $O(1)$
- On parcourt tous les arcs une seule fois, total : $\Theta(A)$
- **Complexité finale : $O(S+A)$**

Algorithmes de parcours de graphes - BFS

```

PL( $G, s$ )
1  pour chaque sommet  $u \in S[G] - \{s\}$ 
2      faire  $\text{couleur}[u] \leftarrow \text{BLANC}$ 
3           $d[u] \leftarrow \infty$ 
4           $\pi[u] \leftarrow \text{NIL}$ 
5   $\text{couleur}[s] \leftarrow \text{GRIS}$ 
6   $d[s] \leftarrow 0$ 
7   $\pi[s] \leftarrow \text{NIL}$ 
8   $F \leftarrow \{s\}$ 
9  tant que  $F \neq \emptyset$ 
10     faire  $u \leftarrow \text{tête}[F]$ 
11         pour chaque  $v \in \text{Adj}[u]$ 
12             faire si  $\text{couleur}[v] = \text{BLANC}$ 
13                 alors  $\text{couleur}[v] \leftarrow \text{GRIS}$ 
14                      $d[v] \leftarrow d[u] + 1$ 
15                      $\pi[v] \leftarrow u$ 
16                      $\text{ENFILE}(F, v)$ 
17                      $\text{DÉFILE}(F)$ 
18      $\text{couleur}[u] \leftarrow \text{NOIR}$ 

```

Version « Cormen »

Utilisations de BFS :

1. (plus court) chemin d'un sommet s_1 à un autre s_2 :

- Faire un **BFS** depuis s_1
- La distance entre s_1 et s_2 est donnée par $d[s_2]$
- Le chemin est construit grâce à π : $\pi[s_2], \pi[\pi[s_2]], \dots$ jusqu'à s_1
- (plus court) On peut utiliser des poids sur les arcs (distance entre deux sommets)- exercice : modifier l'algorithme

2. Arbre recouvrant :

- A construire en remontant de chaque sommet vers la source, à l'aide du tableau π des prédécesseurs.

voir justifications sur [Cormen]

CS353 – Algorithmique et structure de données
12/05/2021

Algorithmes

Problème de la **recherche de motifs** (Pattern matching)

- Introduction
- Brute force
- Rabin-Karp (detail)
- Knuth-Morris-Pratt (KMP) (details + exercice)
- Boyer-Moore (aperçu)
- Aho-Corasick (aperçu)
- Synthèse

Annexe : **filtres de bloom**



CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

Problème :

Trouver un « motif » dans un « texte » (voir définitions plus loin)

Exemples :

- un mot dans un texte en langue naturelle
- un motif dans une séquence musicale (une partition)
- une signature binaire de virus dans un flux réseau
- ...

En entrée : un texte de longueur N ; un motif de longueur M, composés de symboles issus d'un même alphabet

En sortie : un booléen (vrai si le motif apparait dans le texte, faux sinon) ; le cas échéant la position du motif dans le texte

Variante du problème :

Trouver un ensemble de « motifs » dans un « texte »

Exemples :

- Détecter la présence d'une signature de virus dans un flux réseau qui appartient à un ensemble de signatures
- Gérer des listes noires d'URL

En entrée : un texte de longueur N ; un ensemble de motifs

En sortie : des booléens (vrai si un motif apparait dans le texte, faux sinon)

CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

- ▶ un des plus vieux problèmes de l'informatique
- ▶ nombreuses applications
 - ▶ éditeurs de texte
grep en Unix – CTRL s sous Emacs – CTRL f sous Word
 - ▶ moteurs de recherche
 - ▶ analyse de séquence génétiques



- ▶ recherche de motifs musicaux



Aussi :

- Sécurité : inspection de flux réseaux (**Snort** : algo hybride **BM+AC**)
- Réseau/routage (recherche de préfixe dans table)



CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

Quelques repères historiques

- ▶ Extrait de wikipedia, article Chronologie de l'informatique
 - ▶ L'algorithme de Dijkstra par Edsger Dijkstra 1959
 - ▶ L'algorithme de Floyd par Robert Floyd 1959
 - ▶ L'algorithme Quicksort par Tony Hoare 1961
 - ▶ Invention de l'algorithme de Knuth-Morris-Pratt 1975
 - ▶ Invention de l'algorithme de Boyer-Moore 1977
- + Aho-Corasick 1975
+ Rabin-Karp 1987

▶ Prix Turing

- ▶ Donald Knuth 1974
- ▶ Michael Rabin, 1976
- ▶ Richard Karp, 1985

n taille de la chaîne, m taille du motif

Théorie :

- Brute force : $O(m.n)$, moyenne $O(n)$
- KMP : $O(m+n)$
- BM : $O(n.m)$ pire des cas, mais $O(n/m)$ moyenne
- AC : $O(n+m)$, recherches multiples
- RK : $O(m.n)$, recherches multiples

Pratique : optimisations de sous-problèmes

- AC, BM, hybrides

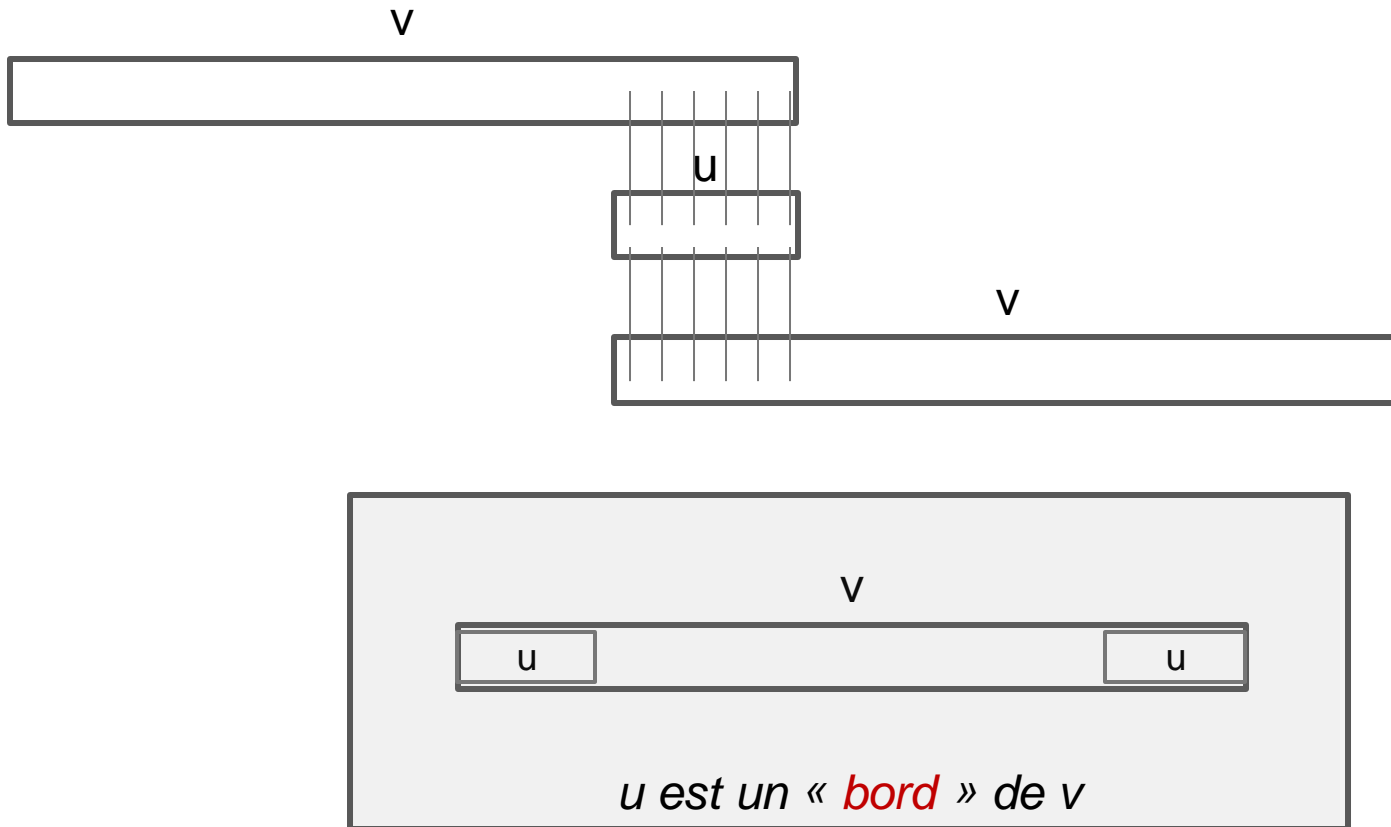
Définitions

- ▶ **Alphabet** : Σ , ensemble fini de lettres (caractères, symboles)
- ▶ **Mot** : suite finie d'éléments de Σ
- ▶ **Concaténation** : la concaténation de deux mots u et v est le mot composé des lettres de u , suivi des lettres de v . Elle est notée $u \circ v$.
- ▶ **Facteur** : un mot u est un facteur du mot v , si, et seulement s'il existe deux mots w et z tels que $v = w \circ u \circ z$
- ▶ **Occurrence** : Si u est un facteur de v , on dit que u apparaît dans v , ou que v contient une occurrence de u
- ▶ **Préfixe** : un mot u est un préfixe du mot v , si, et seulement s'il existe un mot t tel que $v = u \circ t$
- ▶ **Suffixe** : un mot u est un suffixe du mot v , si, et seulement s'il existe un mot t tel que $v = t \circ u$
- ▶ **Bord** : un mot u est un bord du mot v , si et seulement si u est à la fois un préfixe propre et un suffixe propre de v .

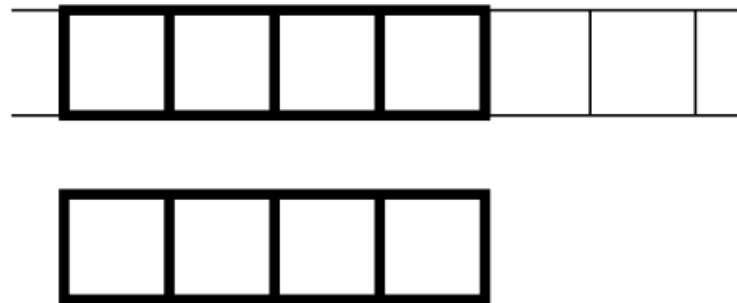
CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

Un « **bord** »



Algorithme naif (par force brute)



Texte T

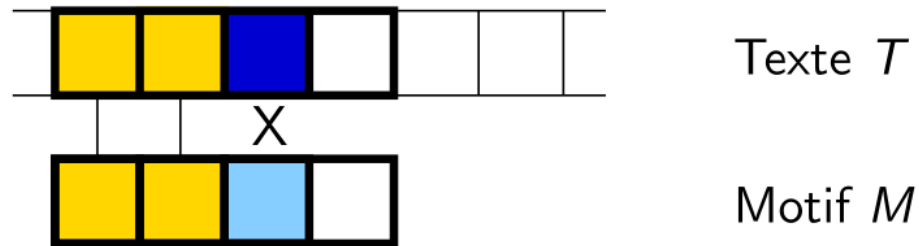
Motif M

- sens de lecture du texte : de gauche à droite

CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

Algorithme naïf (par force brute)

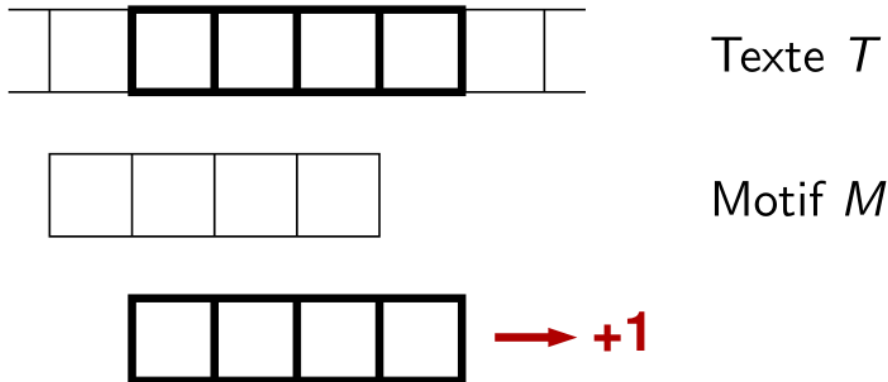


- ▶ sens de lecture du texte : de gauche à droite
- ▶ tentative : comparaison du motif contre le texte, caractère par caractère, de gauche à droite

CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

Algorithme naif (par force brute)



- ▶ sens de lecture du texte : de gauche à droite
- ▶ tentative : comparaison du motif contre le texte, caractère par caractère, de gauche à droite
- ▶ décalage du motif : $+1$ position, vers la droite

CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

Brute-force exact pattern match

Check for pattern starting at each text position.

[illegible]

```
public static int search(String pattern, String text)
{
    int M = pattern.length();
    int N = text.length();

    for (int i = 0; i < N - M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
            if (text.charAt(i+j) != pattern.charAt(j))
                break;
        if (j == M) return i; ← pattern start index in text
    }
    return -1; ← not found
}
```

$O(M.N)$ dans le pire des cas
(lequel ?)

Amélioration de l'approche par force brute

- ▶ sens de lecture du texte : de gauche à droite
- ▶ tentative
 - ▶ sens de lecture du motif : droite, gauche, mélangé
 - ▶ comparaison caractère par caractère / comparaison globale
- ▶ décalage
 - ▶ de plus de 1 position
 - ▶ sans manquer d'occurrences

Algorithme de Karp-Rabin (1987)

- ▶ décalage : +1 position, à droite
- ▶ tentative : comparaison grâce à une fonction de hachage h
- ▶ arithmétique modulo et décalage
- ▶ un mot est codé par un entier en base d

$$h(u) = (u_0 d^{m-1} + u_1 d^{m-2} \dots + u_{m-1}) \text{ modulo } q$$

- ▶ d : taille de l'alphabet
- ▶ q : plus grand entier. Le calcul du modulo se fait implicitement.

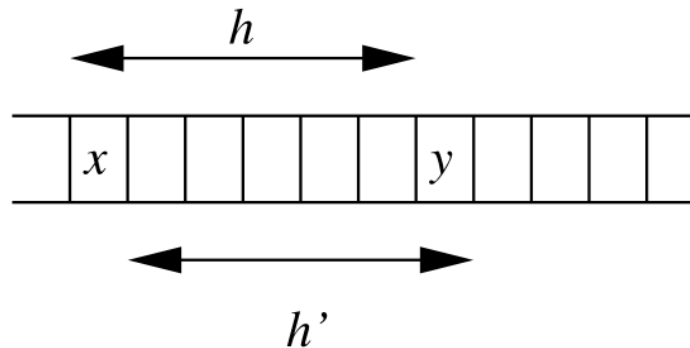
(premier, en fonction des capacités de la machine)

CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

Karp-Rabin : déroulement de l'algorithme

1. calcul de $h(M)$
2. calcul de $h(T(0..m-1))$
3. déplacement d'une fenêtre de longueur m le long du texte, de 1 en 1, avec réactualisation de la valeur de h



« empreinte » de Rabin

$$h' = (d(h - xd^{m-1}) + y) \text{ modulo } q \text{ (temps constant)}$$

4. quand on trouve une fenêtre f telle que $h(f) = h(u)$, on vérifie en **comparant f et u caractère par caractère**

$O(1)$! et pas $O(m)$

CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

Karp-Rabin : Exemple de pseudo-code

```
rabin_karp(T, M)
1  n ← longueur(T)
2  m ← longueur(M)
3  hn ← hach(T[1..m])
4  hm ← hach(M[1..m])
5  pour i de 0 à n-m+1 faire
6    si hn = hm
7      si T[i..i+m-1] = M[1..m]
8        résultat trouvé : i
9    hn ← hach(T[i+1..i+m])
10 résultat non trouvé
```

Temps d'exécution

- 3, 4, 7, 9 en $O(m)$
 - 3, 4 une seule fois
 - 7 uniquement quand les empreintes correspondent (cas « rare » ?)
- 6 exécutée n fois mais temps constant
- 9 ? Idée KR : $O(1)$
 - utilisation d'une fonction de hash « déroulante » : calcul des empreintes successives en $O(1)$

CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

Karp-Rabin : Exemple

- ▶ Alphabet : $\Sigma = \{a, c, g, t\}$,
- ▶ Motif *tata* $\leftrightarrow 4, 1, 4, 1$
- ▶ Hachage : $h(\text{tata}) = 4 \times 4^3 + 1 \times 4^2 + 4 \times 4^1 + 1 = 289$,
- ▶ Le texte et son hachage :

c	t	a	c	t	a	t	a	t	a	t	c
198	284	113	200	289	136	289	136	290			

$$h(\text{ctac}) = 2 \times 4^3 + 4 \times 4^2 + 1 \times 4^1 + 2 \times 4^0 = 2 \times 64 + 4 \times 16 + 4 + 2 = 128 + 64 + 6 = 198$$

Décalage une lettre à droite

$$h(\text{tact}) = 4 \times 4^3 + 1 \times 4^2 + 2 \times 4^1 + 2 \times 4^0 = 4 \times 64 + 16 + 8 + 4 = 256 + 28 = 284$$

Formule du type : $h' = d(h^i - x d^{m-1}) + y$
 $\rightarrow O(1)$

Complexité globale (pire des cas) : **$O(m.n)$**
(on calcule quand même n fois l'empreinte)

CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

Karp-Rabin : conclusion

- ▶ Points forts
 - ▶ Facile à implémenter
 - ▶ Plus efficace en pratique que l'algorithme naïf
- ▶ Points faibles
 - ▶ Complexité en $O(mn)$ dans le pire des cas
 - ▶ Pas le plus efficace en pratique

KMP, k motifs : $O(n \times k)$

KR : $O(n + k)$ en moyenne

Explications KR :

<https://www.youtube.com/watch?v=BfUejqd07yo>

CONCEPTION DES ALGORITHMES

Algorithmes de **pattern matching**

Principe : Variante de Karp-Rabin

- On calcule une empreinte (son hash)
- On vérifie son appartenance à un ensemble de motifs
 - Cette appartenance est vérifiée par un **filtre de Bloom** en $O(1)$ (mais avec quelques faux positifs)
 - En global, on obtient du $O(n + k)$ en moyenne

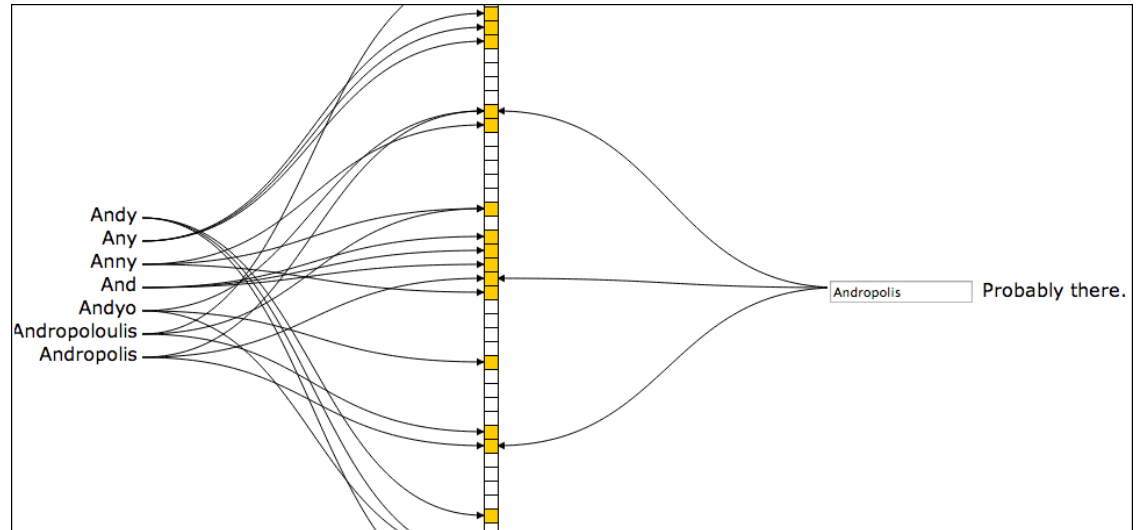
Recherche de motifs multiples :

Exemples :

- Bigtable de Goggle
- Inspection de paquets en profondeur
- Chrome : (*Safe Browser*)
 - Blacklist de 10^6 url taille : 30Mo
→ 2Mo avec filtre bloom

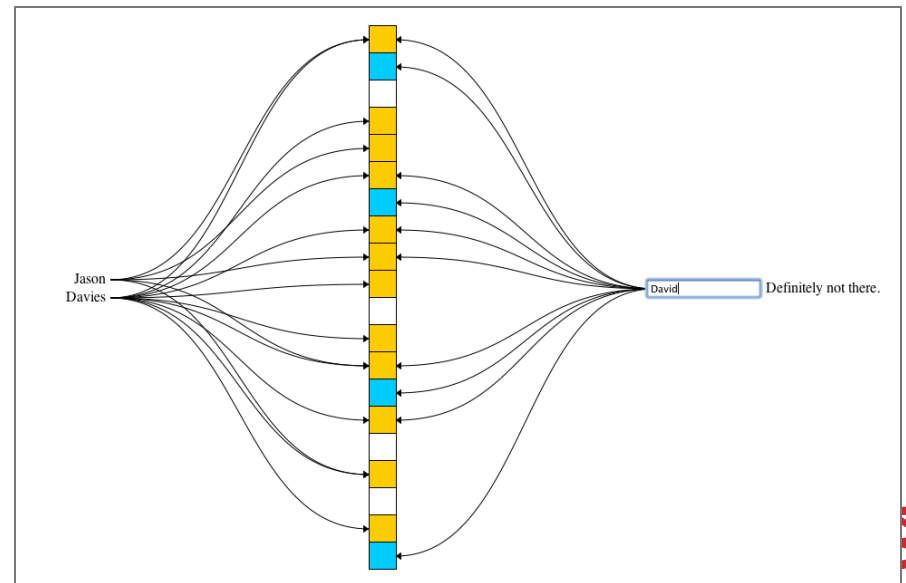
Annexe : filtres de Bloom en une diapo

1. On calcule les hashes des élément de l'ensemble avec plusieurs fonctions de hash différentes f_1 à f_k (dans l'exemple $k=3$)
2. On code le résultats dans une table de bits



Requête d'appartenance d'un mot :

1. On calcule tous les hash du mot
2. On vérifie que tous le bits correspondants sont levés



CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

L'algorithme KMP

- L'algorithme Knuth-Morris-Pratt compare le motif avec le texte **de-gauche-a-droite**, mais le décalage est choisi de façon plus intelligente que dans l'algorithme force brute.
- Quand il y a un "mismatch" quel est le "shift" maximale que l'on peut faire pour éviter comparaisons redondantes ?
- Réponse:

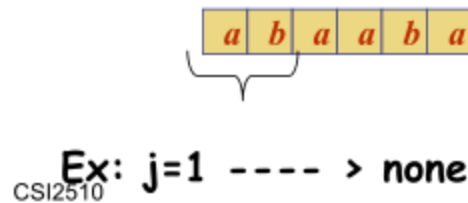
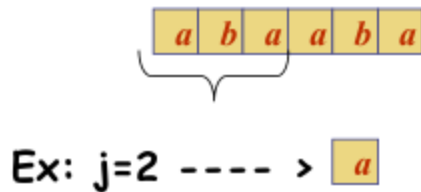
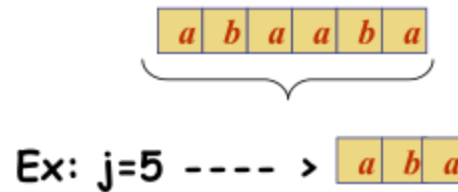
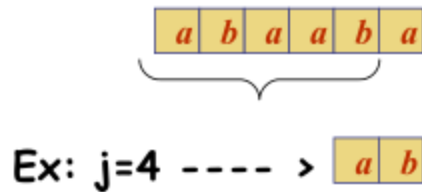
Le plus grand préfixe de $P[0..j]$ qui est un suffixe de $P[1..j]$

« Bord »

CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

Le plus grand préfixe de $P[0..j]$ qui est un suffixe de $P[1..j]$



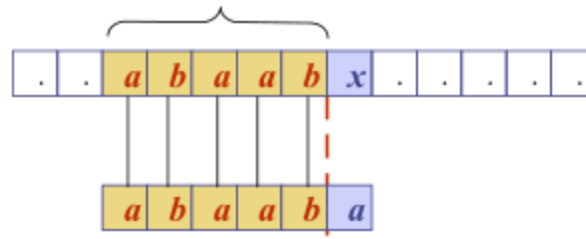
CSI2510

On calcule le **plus grand** « **bord** » de toutes les sous-chaines du motif : $P[0,1], P[0,2], \dots, P[0,m-1]$

CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

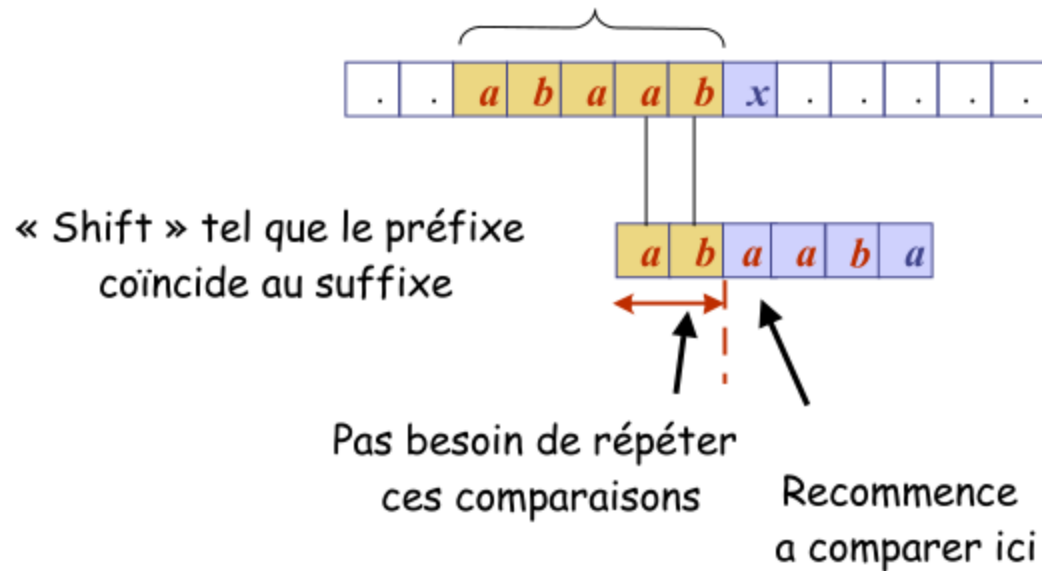
L'algorithme KMP



CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

L'algorithme KMP



CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

- L'algorithme Knuth-Morris-Pratt fait une pre-computation pour trouver préfixes du motif qui coïncident avec suffixes
- La "failure function" $F(j)$ est définie comme la taille du plus long préfixe de $P[0..j]$ qui est aussi un suffixe de $P[1..j]$

a b a a b a

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

a b a a b a

j=2 ---- > a

a b a a b a

j=3 ---- > a

a b a a b a

j=4 ---- > a b

a b a a b a

j=5 ---- > a b a

CONCEPTION DES ALGORITHMES

Algorithmes de pattern matching

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3

. . a b a a b x

- L'algorithme Knuth-Morris-Pratt modifie l'algorithme de force brute de tel façon que si le "mismatch" se vérifie a

$$P[j] \neq T[i]$$

on change: $j \leftarrow F(j - 1)$

a b a a b a

j

a b a a b a

$F(j - 1)$

L'algorithme KMP

- La "failure function" peut être représenté par un tableau qui peut être computed en temps $O(m)$
- A chaque itération de la boucle while,
 - i est incrémenté de 1, OU
 - La quantité de décalage $i - j$ est incrémenté de au moins 1 (observez que $F(j - 1) < j$)
- Donc, il n'y a pas plus que $2n$ itérations de la boucle while
- Donc, l'algorithme KMP a un temps optimal de $O(m + n)$

```
Algorithm KMPMatch( $T, P$ )  
   $F \leftarrow \text{failureFunction}(P)$   
   $i \leftarrow 0$   
   $j \leftarrow 0$   
  while  $i < n$   
    if  $T[i] = P[j]$   
      if  $j = m - 1$   
        return  $i - j$  { match }  
      else  
         $i \leftarrow i + 1$   
         $j \leftarrow j + 1$   
    else  
      if  $j > 0$   
         $j \leftarrow F[j - 1]$   
      else  
         $i \leftarrow i + 1$   
  return  $-1$  { no match }
```