IN 408
Analyse d'algorithmes

Polycopié sans trous

Chapitre 1

Introduction

1.1 prérequis mathématiques

1.1.1 outils à connaitre

- $2^k \approx \mathbf{10^{3\frac{k}{10}}}$
- $\Sigma_{i=1}^{n} i = \frac{\mathbf{n(n+1)}}{\mathbf{2}}$
- $log(a^b) = \mathbf{b} \times \mathbf{log(a)}$
- $2^{log(n)} = \mathbf{n}$
- $\Sigma_{i=1}^{n} \frac{1}{i} \approx \mathbf{log(n)}$ au sens des équivalents (formule d'Euler)

1.1.2 fonctions numériques

le cadre général

de $\mathbb{N}$ *dans* $\mathbb{R}$, *de* $\mathbb{N}$ *dans* $\mathbb{N}$; *partie entière*

des fonctions particulières

les fonctions exponentielle, log, factorielle et constante
log : terre lune : 385000km disons 4×10^{12} dizièmes de mm
$\mathbf{4 \times 10^{12} = 4 \times (10^3)^4 \approx 4 \times (2^{10})^4 = 4 \times 2^{40} = 2^{42}}$

classes de fonctions : exemples

les *fonctions linéaires, les fonctions quadratiques*

1.1.3 pourquoi est-il important de mesurer l'efficacité ?

soit une machine de cycle de base 10^{-9}secondes, et un programme travaillant sur un tableau de n éléments, et dont le nombre d'opérations soit une fonction Φ de n

– *temps d'exécution en fonction de la taille* **à la louche**

Φ / n	$log(n)$	n^2	n^3	2^n
10^3	*4* $\times 10^{-9}$ *seconde*	*1 mili seconde*	*1 seconde*	4×10^4 *milliards d'années*
10^5	**6 10^{-9}**	**10 sec.**	**11,5 jours**	
10^6	**8 10^{-9}**	**16,6 mn.**	**31,7 années**	

– *taille maximum d'un problème soluble en un temps donné sur la même machine*

Φ / Durée	$log(n)$	n^2	n^3	2^n
1 minute	∞	$2,4 \times 10^5$	$3,9 \times 10^3$	31
1 heure	∞	$< \mathbf{2 \times 10^6}$	$< \mathbf{15 \times 10^3}$	**37**
1 jour	∞	$< \mathbf{10^7}$	$< \mathbf{45 \times 10^4}$	**43**

– *évolution de la taille maximum quand la vitesse est multipliée*

par	$log(n)$	n^2	n^3	2^n
10^2		$\times 10^{\mathbf{30}}$	$\times \mathbf{4,64}$	$+\mathbf{6,64}$
10^3		$\times 10^{300}$	$\times \mathbf{10}$	$+\mathbf{9,97}$

1.2 Prérequis de programmation

1.2.1 Types utilisés de données

int, char, float, pointeur

1.2.2 structures utilisées de données

tableaux *l'indice du premier élément sera noté* **1**.
en TP C++, c'est **0**

liste chaînée *liste chaînée avec pointeur sur le dernier élément, liste doublement chaînée*

1.3 classes de fonctions : formalisation

1.3.1 notation $\mathcal{O}$

définition

Si f et g *sont deux fonctions de* $\mathbb{N}$ *dans* $\mathbb{N}$ *(ou* $\mathbb{R}$*),* $f \in \mathcal{O}(g)$ *si et seulement si* $\exists \mathbf{n_0} \in \mathbb{N}$ **et** $\exists \mathbf{c} \in \mathbb{N}$ **tels que** $\forall \mathbf{n} \in \mathbb{N}\ \mathbf{n} > \mathbf{n_0} \Rightarrow \mathbf{f(n)} < \mathbf{c} \times \mathbf{g(n)}$

exemples

les exemples suivants $\in \mathcal{O}(n)$? $\in \mathcal{O}(n^2)$? $\in \mathcal{O}(n^3)$? $\in \mathcal{O}(n^4)$?[1]

- $f(n) = 5n^3 + 3\ n^2 + 1 \in \mathcal{O}(\mathbf{n^3})$
- $f(n) = 1 + 10^{-100}2^n \notin \mathcal{O}(\mathbf{n^4})$
- $f(n) =$ si n est pair $10^{30} \times n^2$ sinon $10^{-30} \times n^3 \in \mathcal{O}(\mathbf{n^3})$

classes de fonctions particulières et inclusions de classes de $\mathcal{O}$

$\mathcal{O}(nlog(n))$, $\mathcal{O}(1)$, $\mathcal{O}(log(n))$, $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$

1.3.2 notation Θ

définition

$(f \in \Theta(g)) \iff (\mathbf{f} \in \mathcal{O}(\mathbf{g})) \wedge (\mathbf{g} \in \mathcal{O}(\mathbf{f}))$

intérêt et inconvénient par rapport à $\mathcal{O}$

intérêt : ω
inconvennient : plus difficile

1.4 taille des instances (données)

1.4.1 taille d'une donnée élémentaire

soit d *une donnée et* $t(d)$ *la taille de cette donnéee,* d *est une donnée élémentaire si et seulement si* $t(d) \in \Theta(\mathbf{1})$

1.4.2 taille d'une structure de n données

chaque donnée est élémentaire

si S *est une structure de* n *données élémentaires et* $t(S)$ *la taille de cette donnéee* $t(S) \in \Theta(\mathbf{n})$

chaque donnée est elle même une structure

si S *est une structure de* n *données structurées, et* $t(d)$ *la taille d'une de ces donnéee structurée,* $t(S) \in \Theta(\mathbf{n} \times \mathbf{t(d)})$

1. *noter l'abus de notation qui consiste à noter* $nlog(n)$ *la fonction qui à* n *fait correspondre* $nlog(n)$

1.4.3 exemples importants

à quelle classe Θ appartient la taille

1. *d'un tableau de n int ?* $\in \mathbf{\Theta(n)}$
2. *d'un tableau de n booléens ?* $\in \mathbf{\Theta(n)}$
3. *d'un tableau de $n \times n$ booléens ?* $\in \mathbf{\Theta(n^2)}$
4. *une liste chaînée de n int ?* $\in \mathbf{\Theta(n)}$
5. *d'un tableau de n listes chaînées d'entiers $L_1,\ L_2, \ldots, L_n$, sachant que la liste chaînée L_i est de taille d_i et que $\Sigma_{i=1}^{n} d_i = m$?* $\in \mathbf{\Theta(m+n)}$

1.5 Efficacité des programmes

1.5.1 introduction

- mesure

mémoire	*temps*
bit	**seconde**

- abstraction

  programme $\rightarrow$ **algo**

  mesure du temps

  en secondes pour une donnée particulière $\rightarrow$
  en nombre d'*opérations élémentaires* en fonction de la taille des données

- *intêret : évolution dans le temps*

1.5.2 opération élémentaire

qu'est-ce qu'une opération élémentaire ?

temps d'exécution dans $\mathcal{O}(\mathbf{1})$ **(ne dépend pas de la taille de la donnée)**

exemples d'opération élémentaire et d'opération non élémentaire

opération	*élémentaire ou non*
affectation, lecture ou écriture d'une donnée élémentaire	**oui**
opération arithmétique : addition, soustraction, multiplication, division	**oui**
opération arithmétique : puissance	**non**
comparaison de deux données élémentaires	**oui**
accès à un élément (indiqué par son indice) d'un tableau	**oui**
recherche dans un tableau de l'indice d'un élément donné par sa valeur	**non**
rajouter dans une liste chaînée un élément derrière un élément donné par son adresse	**oui**
supprimer dans une liste chaînée le premier élément d'une valeur donnée	**non**
comparer deux listes chaînées	**non**
comparer deux tableaux	**non**

1.5.3 évaluation d'un algorithme

Pour étudier l'efficacité d'un algo, on étudie à quelle classe appartient la fonction qui mesure, en fonction de la taille de la donnée (l'instance), le nombre d'opérations élémentaires de cette algorithme.

pourquoi s'interesser à la classe au sens Θ ?
- *données de grande taille*
- *constantes multiplicatives ignorées*

pourquoi s'interesser à la classe au sens $\mathcal{O}$?

exemple *ROT : rotation d'une case vers la droite d'un bloc d'un tableau*

Données: *un tableau T de dimension d , deux indices valides de T, k et n tels que $k \leq n$*
Résultat: *le sous tableau des cases de T entre l'indice k et l'indice n a effectué une rotation de une case (vers la droite).*
exemple : $n = 7,\ d = 8\ ,\ k = 4$

avant	*1*	*3*	*5*	*7*	*2*	*4*	*6*	*8*
après	**1**	**3**	**5**	**6**	**7**	**2**	**4**	**8**

temp $\leftarrow \mathbf{T[n]}$;
pour i décroissant de $n-1$ à k **faire** $\mathbf{T[i+1]} \leftarrow \mathbf{T[i]}$;
$T[i] \leftarrow$ **temp**;

- *taille de la donnée :* $\mathbf{d} \in \mathbf{O(d)}$
- *opération élémentaire :* **l'affectation**
  - **d'une case du tableau**
  - **de l'indice**
- *nombre d'opérations élémentaires* : $\mathbf{n-k+2}$ **pour les affectations de case,** $\mathbf{n-k}$ **pour les indices.Les deux sont dans** $\mathcal{O}(\mathbf{d})$
  Attention k et n sont des données, mais n'affectent pas la **taille de la donnée.**
  C'est pour ça qu'on ne parle pas de $\mathcal{O}(\mathbf{n-k})$

1.5.4 analyse dans le pire des cas et en moyenne : appartenance d'un élément à un tableau

recherche linéaire

recherche dichotomique itérative dans un tableau trié

Données: un entier e, un tableau trié T de n cases
Résultat: un booléen indiquant si $e \in T$
$deb \leftarrow 1$; $fin \leftarrow n$;
tant que deb $\leq$ **fin faire**
  $mil \leftarrow$ (**deb** + **fin**) **div 2**;
  si $T[mil]$= **e alors retourner *vrai*** ;
  si $T[mil]$< **e alors**
    $deb \leftarrow$ **1** + **mil**;
  sinon
    $fin \leftarrow$ **mil** − **1**;
  fin
fin
retourner *faux*

Pourquoi ne peut-on pas faire une telle recherche dichotomique itérative dans une liste chaînée triée ?

recherche d'un élément v dans un tableau T : analyse en moyenne

*Les éléments de T de taille n sont des nombres entiers distribués de façon **équiprobable** entre 1 et k (une constante).*

Considérons l'algorithme de recherche ci-dessous qui cherche une valeur v dans T.

pour $i \leftarrow 0$ **to** n **faire**
  si $T[i] =$ **v alors retourner**
fin
retourner Faux

- *complexité dans le pire des cas qui est* $\mathbf{v} \not\in \mathbf{T} \quad : \quad \in \mathbf{\Theta(n)}$
- complexité en moyenne
  - *nombre de tableaux différents :* $\mathbf{k^n}$
  - *nombre de tableau n'ayant pas la valeur v :* $\mathbf{(k-1)^n}$
    nombre de passages dans la répétitive nécessaires pour chacun $\mathbf{n}$
  - *nombre de tableaux dont la première occurence de v est dans la case i :* $(k-1)^{i-1} \times k^{n-i}$
    : chacun de ces tableaux nécessite **i** *passages dans la répétitive*

  - *nombre moyen de passages dans la répétitive* $= \frac{\textit{nombre total de passages dans la répétitive si on traite tous les tableaux}}{\textbf{nombre total de tableaux}}$

$$= \frac{\mathbf{n}\times\mathbf{(k-1)^n} + \mathbf{\Sigma_{i=1}^{i=n} i}\times\mathbf{(k-1)^{i-1}}\times\mathbf{k^{n-i}}}{\mathbf{k^n}}$$

  - *après des gros calculs* $= k \times (1 - (1 - \frac{1}{k})^n)$

1.6 Limites de la théorie

des limites pratiques

- *plus mauvais cas très rare*
- *coefficient multiplicatif*
- *prix occupation mémoire*
- *prix maintenance (simplicité code)*
- *utilité du programme*

des limites théoriques *Comparons les deux algo qui reçoivent tous deux en entrée un entier n et renvoient tous deux un entier Res*

Gauss
$Res \leftarrow 0$;
pour k *de* 1 *à* n **faire** $Res \leftarrow Res + k$;

Fibonacci
$i \leftarrow 1$; $Res \leftarrow 0$;
pour k *de* 1 *à* n **faire** $Res \leftarrow Res + i$; $i \leftarrow Res - i$;

les additions sont en temps constant pour des entiers $< \mathbf{2^{32}}$ *(pour des mots de 32 bits). C'est raisonnable pour Gauss, ça ne l'est plus pour Fibo dés que* $\mathbf{n = 47}$ *pour considerer comme élémentaire l'addition, pour* $n = 2^{32}$*, il faudrait des mots de* ***45496*** *bits !*

grain de sel $n^{log\ log\ n} \leq n^{10}$ *tant que* $n \leq 10^{300}$ *or* $\mathbf{\Theta(n^{10} \subset \Theta(n^{log(log(n))}}$

Chapitre 2

Outils de calcul de la classe de complexité d'un algorithme

2.1 prérequis mathématiques : classes de fonction et opération sur les fonctions

2.1.1 les résultats

1. $\forall \lambda$ constant $\Theta(\lambda f) = \mathbf{\Theta(f)}$
2. $O(f) \subseteq O(g) \Rightarrow \Theta(f+g) = \mathbf{\Theta(g)}$
3. $\Theta(max(f,g)) = \mathbf{\Theta(f+g)}$
4. $\{\Theta(f') = \Theta(f)\} \wedge \{\Theta(g') = \Theta(g)\} \Rightarrow \{\Theta(f' \times g') = \mathbf{\Theta(f \times g)}\}$

2.1.2 utilisation

$f(n) = 3^n + n^4,\ g(n) = n^2 + n + 1 \Rightarrow f(n) \times g(n) \in \Theta(\mathbf{n^2 3^n})$

2.2 calcul de la complexité d'un algorithme

2.2.1 problème indécidable

f(n)= si n pair alors $\frac{n}{2}$ *sinon si* $n = 1$ *alors 1 sinon 3n+1*
on croit *que* $\forall n\ f(n) = 1$ *mais* **on n'est pas sûr que dans certains cas ça ne boucle pas !**

2.2.2 itération

exemples

1. *nombre d'occurences d'un entier donné v dans un tableau donné T de dimension d ?* **avec une boucle pour** $\in \mathbf{\Theta(d)}$

   Res $\leftarrow$ **0**;
   pour i ***de 1 à*** **n faire si v** $=$ **T[i] alors Res** $\leftarrow$ **Res** $+$ **1**;
   retourner Res

2. *un tableau contient-il deux éléments identiques ?* **avec un return dans la boucle pour interne** $\in \mathbf{\Theta(d^2)}$

   pour ***i de 1 à n*** **faire pour** ***j de i+1 à n*** **faire si** ***T[i]=T[j]*** **alors retourner** ***vrai***; **retourner** ***faux***

2.2.3 séquence

exemple donnée : 3 tableaux $T_1,\ T_2,\ T_3$ de même taille n
rèsultat : un tableau T_R somme des 3 donnés
avec une seule boucle et à chaque fois une addition ternaire, avec une seule boucle et à chaque fois deux additions, avec deux boucles et à chaque fois une addition

pour ***i de 1 à n*** **faire** $\mathbf{T_R[i] = T_1[i] + T_2[i] + T_3[i]}$

pour ***i de 1 à n*** **faire** $\mathbf{T_R[i] = T_1[i] + T_2[i]; T_R[i] = T_R[i] + T_3[i]}$

pour ***i de 1 à n*** **faire** $\mathbf{T_R[i] = T_1[i] + T_2[i]}$;
pour ***i de 1 à n*** **faire** $\mathbf{T_R[i] = T_R[i] + T_3[i]}$;

2.2.4 exercices

1. *imprimer une chaine de caractères qui indique si un nombre donné appartient à un tableau donné*
2. *données : un tableau de nombres pairs P de taille n_P, un tableau de nombres impairs I de taille n_I, et un nombre n*
   résultat : savoir si n est dans l'un des deux tableaux (en utilisant le nombre d'occurences) sans test (nboccurences dans P plus nboccurences dans I) ou avec test (n pair ou impair)
3. *donnée : 3 tableaux $T_1,\ T_2,\ T_3$ de tailles $n_1,\ n_2,\ n_3$ un entier d*
   Resultat : si $d \in T_1$ son nombre d'occurences dans T_2 sinon son nombre d'occurences dans T_3

2.2.5 appels de fonction

exemples

1. *fonctions appelées*

   Stupide
   Données: *un entier n*
   Résultat: *un entier Res*
   $Res \leftarrow 0$;
   pour i de 1 à n **faire** $Res \leftarrow Res + i$;
   retourner $2 \times Res\ - n$

   renvoie $\mathbf{n^2}$ **(demo par récurence sur n)** $\in \mathbf{\Theta(n)}$

Intelligent
Données: *un entier* n
Résultat: *un entier*
retourner $n \times n$
$\in \mathbf{\Theta(1)}$

2. *appels*

PlusStupide
Données: *un entier* n
Résultat: *un tableau* $TRes$ *des* n *premiers carrés*
$TRes \leftarrow$ *un tableau vide*;
pour i de 1 à n **faire** $TRes[i] \leftarrow$**Stupide**$[i]$;
retourner $Tres$

PlusIntelligent
Données: *un entier* n
Résultat: *un tableau* $TRes$ *des* n *premiers carrés*
$TRes \leftarrow$ *un tableau vide*;
pour i *de 1 à* n **faire** $TRes[i] \leftarrow i \times i$;
retourner $Tres$

$\in \mathbf{\Theta(n)}$

Et ça ?
Données: *un entier* n
Résultat: *un tableau* $TRes$ *des* n **premiers carrés**
$TRes \leftarrow$ *un tableau vide ;* $TRes[1] \leftarrow 1$;
pour i *de 2 à* n **faire** $TRes[i] \leftarrow TRes[i-1] \ -1 \ +2 \times i$;
retourner $Tres$
$\in \mathbf{\Theta(n)}$
l'accès à une case d'un tableau est une opération **élémentaire**.

2.3 appels recursifs

2.3.1 quelques exemples sur les listes

1. **ajouteEnFin**
Données: un élément e une liste L
Résultat: une liste identique à L, sauf que e a été ajouté en fin
si *estVide ?[L]* **alors**
$[e]$
sinon
cons(first(L),ajouteEnFin(e, succ(L)));
fin

Démonstration correction par ... **récurrence**.

- *opérations élémentaires :* **cons, first, succ et appel de fonction (et fabrication d'une liste à un élément)**
- *équation de récurrence :*
  si f(n) est la complexité du calcul de ajouteEnFin(e,L) quand L a n éléments :

  - *f(0)=***1**
  - *f(n+1)=***f(n) + a (une constante indépendante de n)**
- *solution*
  - *on va voir que* $f(n) \in \Theta(\mathbf{n})$
  - *si l'équation de récurrence était f(n+1)=2+f(n) avec f(0)=1 on vérifierait* $f(n) = \mathbf{2n+1} \in \mathbf{\Theta}(\mathbf{n})$
  - *si l'équation de récurrence est f(n+1)=a+f(n) avec f(0)=b on vérifie* $f(n) = \mathbf{an+b} \in \mathbf{\Theta}(\mathbf{n})$

2. **dernier** $\in \Theta(\mathbf{n})$
   si *L a un seul élément* alors retourner *le seul élément de L*;
   retourner *dernier(cdr(L))*;

3. **concatener** $\in \Theta(\mathbf{n_1})$
   si *L1 est vide* alors retourner L2;
   retourner *cons(first(L1), concatener(succ(L1),L2))*;

4. **concatenerStupide**
   Données: $L_1,\ L_2$
   Résultat: $L = L_1 L_2$
   si EstVide(L_2) **alors**
   L_1
   sinon
   ajouteEnFin(first(L_2*),concatenerStupide(*L_1*, succ(*L_2*))*;
   fin

renvoie **la concaténation de L1 et L2** *équation de récurrence :*
f(n_2)= $\lambda(n_1 + n_2 - 1) + \mathbf{f(n_2 - 1)}$
$=\lambda n_2 + f(n_2 - 1) + \mu$ *avec* $\mu = \lambda(\mathbf{n_1 - 1})$
$f(0) = \mathbf{1}$
on va voir que $f(n_2) \in \Theta(\mathbf{n_2^2})$

2.3.2 solution d'équations récursives

preuve par récurrence pour l'appartenance à $\mathcal{O}$
il faut montrer, pour la réciproque (appartenance à $\mathbf{\Theta}(\phi)$ **), que** ϕ **vérifie l'équation de récurrence.**

1. $f(n) = f(n-1) + a \Rightarrow f(n) = \mathbf{f(0) + an} \in \mathbf{\Theta}(\mathbf{n})$
   Démonstration par récurrence sur n
   - $\mathcal{P}_\mathbf{n}$ $\mathbf{f(n) = f(0) + an}$

   - **initialisation** : $n = \mathbf{0}$ **trivial**
   - **récurrence : supposons** $\mathcal{P}_\mathbf{n}$ **et prouvons** $\mathcal{P}_{\mathbf{n+1}}$
     $\mathbf{f(n+1) = f(n) + a}$ **par l'équation récursive**
     $\mathbf{= f(0) + an + a}$ **par hypothese recurrence** $\mathbf{= f(O) + a(n+1)}$ **CQFD**

2. $f(0) = 0$ *et* $\forall n > 0\ \exists n_1,\ n_2 \in \{0 \dots n-1\}$ *non nuls tels que* $n = n_1 + n_2 + 1$ *et* $f(n) = k + f(n_1) + f(n_2) \Rightarrow f(n) = \mathbf{kn} \in \mathbf{\Theta(n)}$
   Démonstration par récurrence sur n
   - $\mathcal{P}_\mathbf{n}$: $\forall \mathbf{n'} \in \{\mathbf{0} \dots \mathbf{n}\}\ \mathbf{f(n')} = \mathbf{kn'}$
   - **initialisation** : $n = \mathbf{0}$
   - **récurrence : supposons $\mathcal{P}_\mathbf{n}$ et $\mathbf{n+1} = \mathbf{n'1} + \mathbf{n'_2} + \mathbf{1}$ tels que**
     $\mathbf{f(n+1)} = \mathbf{f(n'_1)} + \mathbf{f(n'_2)} + \mathbf{k}$
     $= \mathbf{kn'_1} + \mathbf{kn'_2} + \mathbf{k}$ **par hypothèse de récurrence**
     $= \mathbf{k(n'_1} + \mathbf{n'_2} + \mathbf{1)} = \mathbf{k(n+1)}$ **CQFD**
3. $f(n) = f(n-1) + a(n-1) + b \Rightarrow f(n) = a\frac{n(n-1)}{2} + nb + f(0) \in \Theta(n^2)$
   Démonstration par récurrence sur n
   - $\mathcal{P}_\mathbf{n}$: $\mathbf{f(n)} = \mathbf{a}\frac{\mathbf{n(n-1)}}{\mathbf{2}} + \mathbf{nb} + \mathbf{f(0)}$
   - **initialisation** : $\mathbf{n = 0}$
   - **récurrence : supposons $\mathcal{P}_\mathbf{n}$**
     $\mathbf{fn+1)} = \mathbf{f(n)} + \mathbf{an} + \mathbf{b}$ **par définition**
     $= \mathbf{a}\frac{\mathbf{n(n-1)}}{\mathbf{2}} + \mathbf{nb} + \mathbf{f(0)} + \mathbf{an} + \mathbf{b}$ **par Hypothèse récurrence**
     $= a\frac{(n+1)n}{2} + (n+1)b + f(0)$ **CQFD**
4. *pour* $n \geq 1$: $f(n) = f(n \text{ div}^1 2) + a \Rightarrow f(n) = alog_2(n) + f(1) \in \Theta(log_2(n))$
   Démonstration par récurrence sur k tel que $\mathbf{n < 2^k}$
   - $\mathcal{P}_\mathbf{k}$: $\forall \mathbf{n}$ **tel que** $\mathbf{1 \leq n < 2^k}$: $\mathbf{f(n)} = \mathbf{alog_2(n)} + \mathbf{f(1)}$
   - **initialisation** : $k = \mathbf{1}$
     on a alors $\mathbf{n = 1}$ **et** $\mathbf{log_2(1) = 0}$
   - **récurrence : supposons $\mathcal{P}_\mathbf{k}$**
     on a $\forall \mathbf{n} \in \{\mathbf{2^k} \dots \mathbf{2^{k+1} - 1}\}\ \mathbf{f(n)} = \mathbf{a} + \mathbf{f(ndiv2)}$ **or comme n div 2 plus petitque** $\mathbf{2^k}$
     $\forall \mathbf{n} \in \{\mathbf{2^k} \dots \mathbf{2^{k+1} - 1}\}\ \mathbf{f(n)} = \mathbf{a} + \mathbf{alog_2(ndiv2)} + \mathbf{f(1)} = \mathbf{alog_2(n)} + \mathbf{f(1)}$ **CQFD**
5. $f(n) = af(n-1) \Rightarrow f(n) = a^n f(0)$
   Démonstration par récurrence sur n

2.3.3 exemple d'utilisation : recherche de la valeur la plus forte d'une branche maximale (au sens de l'inclusion) dans un arbre binaire : BVPF

Données: *un arbre binaire* A
Résultat: *un entier*
si Vide(A) **alors retourner** ***0***;
retourner racine(A) + max (**BVPF(SAG(**A**)), BVPF(SAD(**A**))**);

- *complexité en fonction de la hauteur de l'arbre* $\mathbf{\Theta(h)}$
- *complexité en fonction du nombre de sommets de l'arbre* $\mathbf{\Theta(n)}$
- *rapport entre la hauteur et le nombre de sommets de l'arbe ; arbre équilibré*

1. division entière

Chapitre 3

graphes et arborescences binaires

3.1 Rappels sur les définitions

Graphe orienté $G = (X, E)$
- X *est un ensemble donné de* **sommets**
- $E \subseteq \mathbf{X} \times \mathbf{X}$

arborescences binaires $A = (X, r, E)$ *c'est un graphe tel que*
- r *n'est* **fils d'aucun sommet**
- *le degré rentrant de tout sommet* $x \in X - \{r\}$, $d^-(x) = \mathbf{1}$
- *le degré sortant de tout sommet* $x \in X$, $d^+(x) \leq \mathbf{2}$

3.2 représentations en machine

3.2.1 d'un graphe G de n sommets

1. *par une matrice de* $\mathbf{n} \times \mathbf{n}$ **booleens**
   la taile de $G \in \Theta(\mathbf{n^2})$
2. *par un tableau de* **n listes chaînées**
   si G *a* m *arcs, la taile de* $G \in \Theta(\mathbf{n + m})$

3.2.2 d'une arborescence A de n sommets

```
struct Sommet;

typedef Sommet* AB;

struct Sommet {
  Valeur racine;
  AB Pere,SAG, SAD;
```

la taile de $A \in \Theta(\mathbf{n})$

3.3 Algorithmes

3.3.1 addition d'un arc dans un graphe

représenté par

1. *une matrice :* $\in \Theta(\mathbf{1})$
2. *un tableau de listes chaînées* : $\in \Theta(\mathbf{1})$ **ou** $\in \mathbf{\Theta}(\mathbf{n})$ **suivant que l'on teste ou non si l'atc y est déjà**

3.3.2 suppression d'un arc dans un graphe

représenté par

1. *une matrice :* $\in \Theta(1)$
2. *un tableau de listes chaînées :* $\in \Theta(\mathbf{n})$ **où** $\mathbf{d \leq n}$ **est le degré du soomet dont est issus l'arc qu'on supprime**

3.3.3 rajout de sous arborescence dans une arborescence

```
void GrefferSAG(AB g); void GrefferSAD(AB d);
```

$\in \Theta(\mathbf{1})$

3.3.4 suppression de sous arborescence dans une arborescence

```
void SupprimerSAG();  void SupprimerSAD();
```

$\in \Theta(\mathbf{1})$

3.3.5 Traversées en profondeur d'arborescence binaire

transparent cours2 traversée arborescence binaire
Traversee(A)

Données: *une arborescence binaire A*
si A ***n'est pas vide*** **alors**
    visite prefixe de A;
    Traversee(**SAG(A)**);
    visite infixe de A;
    Traversee(**SAD(A)**);
    visite suffixe de A;
fin

Chapitre 4

Hashing

4.1 Introduction

4.1.1 Le problème

combien d'images différentes 1000×1000 ? : $\mathbf{10^3 * 10^3 * 64}$

4.1.2 Les fonctions de hachage

de quoi s'agit-il ?

un exemple avec comme fonction $h(x) = x \ mod \ 13$

rôle
exemple : Cours7Hashing1Mod

hashing uniforme

sensibilité : Cours7Hashing2AutreFonctionHashing

le problème des collisions

et l'anniversaire

de mauvaises fonction de hachage

Cours7Hashing3Mauvaise fonctionhashing rapidité

4.2 Les différents hashings

Cours7Hashing4HashingCoalescent clustering cookie monster effect
Cours7Hashing5Chainage
Cours7Hashing6DoubleHashing
Cours7Hashing7HashingContinu
dans le hachage continu dans un tableau de taille m *on dispose de* m *fonction de hachage indépendantes*

4.3 le problème de la suppression

4.4 Complexité

en moyenne
λ facteur de charge
distinguer le nombre de tentatives en cas de recherche infructueuse $e(\lambda)$ (le même à un près qu'en cas d'insertion) et en cas de recherche couronnée de succès $s(\lambda)$
indépendance des insetions
équiprobabilité des clés donc des indices (uniforme)

4.4.1 Les résultats

1. hachage chaîné
   longueur de la chaîne moyenne : λ
   $e(\lambda) = 2 + \lambda \qquad s(\lambda) = 2 + \frac{\lambda}{2}$
   quand la clé recherchée a été insérée, tout se passe comme si elle avait été insérée au milieu de la séquence d'insertion
   attention le nombre moyen de recherche n'est pas (en cas de succès) la moitié de la longueur de la chaîne moyenne
2. hachage coalescent
   $e(\lambda) = \frac{1+\frac{1}{(1-\lambda}^2}{2} \qquad s(\lambda) = \frac{1+\frac{1}{1-\lambda}}{2}$
3. double hachage quand $\lambda < 0.319$
   $e(\lambda) = \frac{1}{1-\lambda} \qquad s(\lambda) = \frac{ln(\frac{1}{1-\lambda})}{\lambda}$
4. hachage uniforme $e(\lambda) = \frac{1}{1-\lambda} \qquad s(\lambda) = \frac{ln(\frac{1}{1-\lambda})}{\lambda}$

Chapitre 5

Les Tris

Problème classique (par exemple pour faire des recherches dichotomiques qui sont optimales). On supposera toujours que les éléments sont disjoints pour effectuer l'analyse. Comme on utilise des algorithmes basés sur les comparaisons, on peut supposer que le tableau ne comporte que des éléments de 1 à n (la taille du tableau).

5.1 Tri par insertion

ROT
rotation d'une case vers la droite d'un bloc d'un tableau

Données: **un tableau T de dimension d , deux indices valides de T, k et n tels que k $\leq$ n**
Résultat: **le sous tableau des cases de T entre l'indice k et l'indice n a effectué une rotation de une case (vers la droite).**
temp $\leftarrow$ **T**[**n**]
pour *i décroissant de* **n − 1** *à* **k** **faire** $T[i+1] \leftarrow$ **T**[**i**];
$T[i] \leftarrow$ **temp**;

INSERT
Données: **un entier n, un tableau T de d > n cases dont les n premières cases sont triées**
Résultat: **le tableau T est trié sur les n + 1 premières cases, les autres cases n'ont pas changé**
$k \leftarrow$ **n + 1**;
tant que T[**k − 1**] > **T**[**n + 1**] *et* **k > 1 faire** $k \leftarrow$ **k − 1**;
les cases de T d'indices dans [k,n] contiennent des éléments supérieurs à **toutes les cases entre 1 et n qui sont triées**;
ROT(**T**, **k**,**n + 1**);

TRI-INSERTION

Données: **un tableau non trié T de dimension d**
Résultat: *T est* **trié**
pour n *de 2 à* d faire
    T est trié sur les $n-1$ *premières cases*;
    INSERT$(n-1,\ T)$;
fin

complexité $\mathbf{\Theta(n^2)}$

5.2 Tri par selection

pour *i de* 1 *à n* **faire** $j \leftarrow indicePlusPetiteValeur(T, i)$; **Echanger**(T, j, i) ;

complexité $\in \Theta(n^2)$

5.3 Tri fusion

FUSION-ORDONNEE

Données: *un tableau T et trois indices valides d, m et f de ce tableau avec* $d \leq m < f$
T est trié
– *d'une part entre les indices (inclus)* **d et m**
– *d'autre part entre les indices inclus* $\mathbf{m+1}$ **et f**
Résultat: *T est trié entre* **d et f (inclus)**
T_1 *est alloué comme un tableau de dimensions* $(m-d+1)+\mathbf{1}$ et T_2 *comme un tableau de dimensions* $(f-(m+1)+1)+\mathbf{1}$;

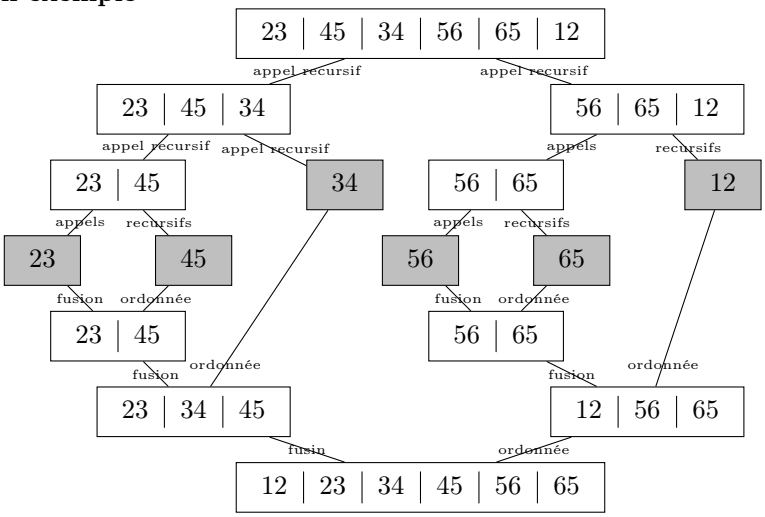
pour *i de 1 à* $m-d+1$ **faire** $T_1[i] \leftarrow \mathbf{T[d+i-1]}$;
pour *i de 1 à* $f-m+1$ **faire** $T_2[i] \leftarrow \mathbf{T[f-m+i-1]}$;
$T_1[m-d+2] \leftarrow \mathbf{1}$; $i_2 \leftarrow \mathbf{1}$;
sentinelles
$i_1 \leftarrow 1$; $i_2 \leftarrow 1$;

pour *i de d à f* **faire**
    si $\mathbf{T_1[i_1] > T_2[i_2]}$ **alors**
        $T[i] \leftarrow \mathbf{T_2[i_2]}$; $i_2 \leftarrow 1 + i_2$;
    sinon
        $T[i] \leftarrow \mathbf{T_1[i_1]}$; $i_1 \leftarrow 1 + i_1$;
    fin
fin

TRI-FUSION

Données: un tableau T et deux indices valides d et f de ce tableau avec $d \leq f$
Résultat: T est trié entre d et f (inclus)
si $\mathbf{d} \neq \mathbf{f}$ **alors**
  mil $\leftarrow \lceil \frac{\mathbf{d+f-1}}{\mathbf{2}} \rceil$;
  TriFusion$(\mathbf{T}, \mathbf{d}, \mathbf{mil})$;
  TriFusion$(\mathbf{T}, \mathbf{mil+1}, \mathbf{f})$;
  FusionOrdonnee$(\mathbf{T}, \mathbf{d}, \mathbf{mil}, \mathbf{f})$;
fin

Un exemple



complexité $\in \mathbf{\Theta(nlogn)}$

5.4 Tri rapide

algorithme :

Pour trier[1] *un ensemble S de n éléments avec $n > 1$:*

- *on choisit un élément x (appelé le pivot) de S (a priori au hasard, en pratique le premier élément du tableau)*
- *on construit l ' ensemble S_1 des éléments de S* **plus petits que x**
- *on construit l ' ensemble S_2 des éléments de S* **plus grands que x**
- tri(S) = **tri($\mathbf{S_1}$), x, tri($\mathbf{S_2}$) (la concaténation)**

Complexité en moyenne du tri rapide :

REVOIR ET EXPLICITER PIRE CAS

1. **équation de récurrence**
   - *si p est l'indice ou on place le pivot d'un tableau de n éléments, $p-1$ et $n-p$ sont les tailles des deux sous tableau à traiter, et si $f(n)$ est la complexité de traitement, alors* $f(n) = \lambda\mathbf{n} + \mu + \mathbf{f(n-p)} + \mathbf{f(p-1)}$
   - *si tous les indices peuvent être choisis de façon équiprobable pour le pivot, avec à chaque fois une probabilité de $\frac{1}{n}$, alors*

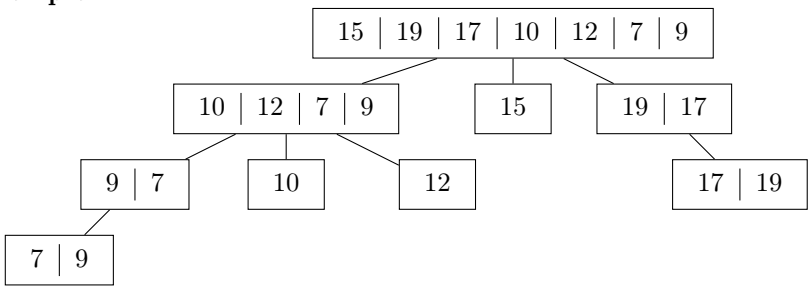
$$\mathbf{f(n)} = \lambda\mathbf{n} + \mu + \mathbf{\Sigma_{p=2}^{p=n-1}} \frac{\mathbf{f(n-p)} + \mathbf{f(p-1)}}{\mathbf{n}} + \frac{\mathbf{2f(n-1)}}{\mathbf{n}} = \lambda\mathbf{n} + \mu + \mathbf{2}\frac{\mathbf{\Sigma_{p=1}^{p=n-1}f(p)}}{\mathbf{n}}$$

2. solution : $f(n) \in \mathcal{O}(\mathbf{nlog(n)})$

Dans le pire des cas, le pivot est toujours **le plus petit élément ou le plus grand** *et la complexité devient du* $\Theta(\mathbf{n^2})$

1. cet algorithme sera approfondi en TD

exemple :



5.5 On ne peut pas faire mieux

5.5.1 Le tri par panier, un algorithme en $O(n)$

5.5.2 preuve qu'on ne peut pas faire mieux que $\Theta(nlogn)$

for comparison-based sorting algorithms we can construct a model that corresponds to the entire class of algorithms : a decision tree.

The root of the decision tree corresponds to the state of the input of the sorting problem (e.g. an unsorted sequence of values). Each internal node of the decision tree represents such a state, as well as two possible decisions that can be made as a result of examining that state, leading to two new states. The leaf nodes are the final states of the algorithm, which in this case correspond to states where the input list is determined to be sorted. The worst-case running time of an algorithm modelled by a decision tree is the height or depth of that tree.

A sorting algorithm can be thought of as generating some permutation of its input. Since the input can be in any order, every permutation is a possible output. In order for a sorting algorithm to be correct in the general case, it must be possible for that algorithm to generate every possible output. Therefore, in the decision tree representing such an algorithm, there must be one leaf for every one of n ! permutations of n input values.

Since each comparison results in one of two responses, the decision tree is a binary tree. A binary tree with n ! leaves must have a minimum depth of log2(n !) . But log2(n !) has a lower bound of $\Theta(nlogn)$. Thus any general sorting algorithm has the same lower bound.

http ://planetmath.org/encyclopedia/LowerBoundForSorting.html

Chapitre 6

Les Tas

6.1 les arbres binaires parfaits

6.1.1 Arbres binaires complets

- *seules les feuilles ne sont pas* **de degré 2**
- *toutes les feuilles sont* **à la même profondeur**

6.1.2 arbres binaires parfaits

- *seules les feuilles* **et au plus un seul sommet interne ne sont pas de degré 2**
- *les feuilles sont uniquement sur* **les deux derniers niveaux**
- *au dernier niveau, les feuilles* **sont toutes regroupées à gauche**

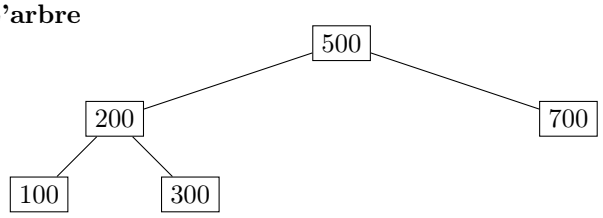
6.1.3 Représentation par un tableau et interêt algorithmique

définition formelle

Pour un tableau T de dimension $n \geq 1$, $\mathbf{T[1]}$ **représente la racine, et** $\forall \mathbf{k}$ **tel que** $\mathbf{2k \leq n}$, $\mathbf{T[2k]}$ **est le fils gauche de** $\mathbf{T[k]}$**, et** $\forall \mathbf{k}$ **tel que** $\mathbf{2k+1 \leq n}$, $\mathbf{T[2k+1]}$ **est le fils droit de** $\mathbf{T[k]}$

Un exemple

L'arbre



le tableau utile correspondant

500	**200**	**700**	**100**	**300**
1	2	3	4	5

interêt algorithmique

accès au père et aux fils, reconnaissance des feuilles en $\mathcal{O}(1)$

6.2 les tas

http ://discala.univ-tours.fr/LesChapitres.html/Cours4/TArbrechap4.6.htm

6.2.1 Définition

(ou file de ppriorité) trouve le plus petit en $\mathcal{O}(1)$ ajout et retrait pas cher (sans précision)

6.2.2 Représentation d'un tas par un arbre parfait et interêt algorithmique

descendre

Données:
- **le tableau T représentant l'arbre**
- **l'indice S représentant le sommet à faire descendre**

si $!Feuille(S)$ **alors**
  $F \leftarrow PlusPetitFils(S, T)$;
  si PlusPetit$(\mathbf{T}, \mathbf{F}, \mathbf{S})$ **alors Echanger**$(\mathbf{T}, \mathbf{F}, \mathbf{S})$; **Descendre**$(\mathbf{T}, \mathbf{F})$;
fin

monter

Données:
- le tableau T représentant l'arbre
- l'indice S représentant le sommet à faire monter

si $!Racine(S)$ **alors**
  $P \leftarrow Pere(S, T)$;
  si $PlusGrand(T, S, P)$ **alors Echanger**$(\mathbf{T}, \mathbf{P}, \mathbf{S})$; **Monter**$(\mathbf{T}, \mathbf{P})$;
fin

fabrication du tas

Données: un tableau Tab de valeurs, de dimension n
Résultat: un tas $TRes$ contenant exactement les valeurs de T

solution naïve

$TRes \leftarrow$ tas vide ;
pour *i de 1 à n* **faire inserer**$(\mathbf{Tres}, \mathbf{T[i]})$;

Complexité : au pas i, l'insertion se fait dans le pire des cas (quand $T[i]$ est supérieur à toutes les valeurs déja insérées) en $log(i)$, donc la complexité totale dans le pire des cas (quand T est trié dans le mauvais sens) est en $\mathbf{\Sigma_{i=1}^{n} log(i) \in \Theta(nlog(n))}$

bonne solution

On utilise directement T et on prend comme algorithme
pour *i* décroissant de n à 1 **faire descendre**$(\mathbf{T}, \mathbf{i})$

Complexité : en appelant h la hauteur de l'arbre, $\mathbf{\Sigma_{k=h-1}^{1} 2^k log(h-k) \in \Theta(h)}$
Il y a beaucoup de sommets qu'on descend un tout petit peu
Cela change quelque chose quand $h = \lceil log(n) \rceil$ c'est à dire quand l'arbre est équilibré.

6.2.3 Tri par tas

1. **on crée d'abord le tas à partir du tableau**
2. **puis on le parcourt en profondeur gauche droite**