Complexité et Algorithmes
Partie I : Complexité

G. Fertin
guillaume.fertin@univ-nantes.fr

Université de Nantes, LS2N
Bât 34 – Bureau 301

M1 Informatique – 2019-2020

Sommaire

Module Complexité et Algorithmes

Volume horaire

- CM: 12h (9 séances)
- TD: 9h20 (7 séances)
- pas de TP (!)
- Distanciel: 2h40

Intervenant CM/TD: G. Fertin

Résultats d'apprentissage 1/2

- Savoir manipuler les notations standards des ordres de grandeurs et des complexité d'algorithmes: $O()$, $\Omega()$, $\Theta()$
- Savoir évaluer la complexité (taille mémoire et temps de calcul) d'un algorithme donné
- Savoir comparer les performances des structures de données standards sur des algorithmes de recherche, d'insertion et de suppression d'éléments dans de grands volumes de données
- Savoir comparer la complexité de plusieurs algorithmes résolvant le même problème, et argumenter le choix d'un algorithme par rapport à sa complexité
- Connaître et savoir interpréter les principales classes de complexité d'un problème, notamment les classes P, NP, NP-dur, NP-complet
- Comprendre et savoir manipuler la notion de réduction polynomiale
- Être capable de réaliser des réductions polynomiales pour montrer qu'un problème est NP-dur

Résultats d'apprentissage 2/2

- Être capable, étant donné un problème nouveau, de déterminer la classe de complexité à laquelle il appartient
- Être capable, étant donné un problème NP-dur, d'identifier des sous-classes d'instance polynomiales
- Connaître les classes de complexité avancées d'un problème, notamment les classes FPTAS, PTAS, APX, APX-dur, APX-complet, FPT, W[1]-dur
- Comprendre et savoir manipuler la notion d'inapproximation d'un problème
- Comprendre la notion de réduction polynomiale préservant l'approximation
- Être capable démontrer qu'un problème est approximable à ratio constant
- Connaître les techniques classiques permettant de prouver qu'un problème est FPT (Fixed- Parameter Tractable)
- Être capable de démontrer qu'un problème est FPT

Évaluation

1 CC, 1 examen, autres

- 1 CC: 40%
- 1 examen: 50%
- Distanciel: 10%

Contenu du Cours

Complexité et Algorithmes

$\Rightarrow$ Étudier des problèmes

- Conception et/ou analyse d'algorithmes qui les résolvent:
  - Conception $\rightarrow$ écriture (en pseudo-code)
  - Analyse $\rightarrow$ *correctness*, complexité
- Complexité des problèmes: meilleur algo, P vs NP-dur
- Problèmes NP-durs: stratégies de résolution

Contenu du Cours

Pré-requis (ordre arbitraire)

- logique booléenne
- structures de données simples (tableaux, matrices, listes chaînées) + graphes
- récursivité
- fonctions mathématiques (ex: $f(n) = n^2$, $f(n) = \log n$, $f(n) = \log \log n$)

Contenu du Cours

Déroulement: 3 grandes parties

1. Complexité (pour l'analyse/évaluation des algorithmes) $\Rightarrow$ les transparents actuels
2. Complexité des problèmes
3. Problèmes NP-durs: stratégies de résolution

Explications/exemples au tableau $\rightarrow$ prendre des notes !!!

Sommaire

Introduction Générale

But de ce cours

- Étudier des problèmes
- Concevoir, écrire, analyser, évaluer des algorithmes
- En particulier, évaluer le coût d'un algorithme
- Coût d'un algo = ressources nécessaires à son exécution, càd:
  - ressources en temps
  - ressources en mémoire

Remarque: dans la suite, et pour les algorithmes, complexité=coût

Algorithme

Qu'est-ce qu'un algorithme ?

- Algorithmes = idées derrière un programme informatique (parallèle avec la "recette de cuisine")
- Algorithme = la chose qui reste la même quel(le) que soit
  - le langage de programmation (C, C++, Python, Java, Ruby, VBA, assembleur MIPS, etc.)
  - la machine (ordinateur) sur laquelle il tourne
- Un algorithme doit résoudre un problème général et bien spécifié
- Un tel problème est décrit par:
  - les instances sur lesquelles il doit fonctionner
  - ce qui est demandé en sortie

Description standard d'un problème

Recherche de Motifs (= Pattern Matching)

$T = ABRACADABRIABRACADABRA$

$M = ABR$

Occurrences de M dans T aux positions 1,8,12,19:

$$T = \underline{ABR}ACAD\underline{ABR}I\underline{ABR}ACAD\underline{ABR}A$$

Convention "NOM/Instance/Sortie":

PATTERN MATCHING
Instance: un texte T de longueur n, un texte M de longueur m
Sortie: les positions de toutes les occurrences de M dans T

Texte = suite ordonnée de lettres prises dans un alphabet Σ

Remarques complémentaires

Instance = n'importe quelle entrée du problème qui respecte la définition

Exemple

Instance: ensemble S d'entiers

- $S = \{2\}$
- $S = \{42, 17, 8, 1, 9, 20, 1, 3\}$
- $S = \{-27, -27, -27, -27, -27, -27, -27, -27, -27\}$
- $S = \{100, 99, 98, \ldots 4, 3, 2, 1\}$

sont des instances autorisées

Remarques complémentaires

Problèmes

- description du problème: pas de structure de données fixée (il faut rester général)
- objectifs:
  - un algo correct
  - aux meilleurs coûts (temps et mémoire)

Algorithme

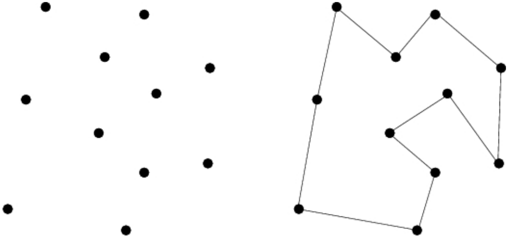
Algorithme correct – Définition

- Algo qui retourne toujours la solution désirée...
- ...pour toutes les instances d'entrée autorisées

Problème et Algorithmes: Exemple

Algorithme correct: pas toujours facile

- Robot qui doit souder des points d'une pièce en 2D...
- ...et revenir au point de départ pour démarrer la pièce suivante



Exemple du robot soudeur

Algorithme correct: pas toujours facile

- But: trouver l'ordre dans lequel souder les points...
- ...qui minimise la distance totale parcourue
- Minimiser pour:
  - Gain de temps ($\Rightarrow$ productivité max.)
  - Durée de vie du robot ($\Rightarrow$ usure min.)
- $\Rightarrow$ Etant donné une liste de points à souder, trouver un algorithme qui en détermine l'ordre optimal
- optimal $\leftrightarrow$ qui minimise la distance

Robot: un algo rapide

Plus proche voisin non soudé

- On commence par un point (n'importe lequel, disons p_1), et on soude.
- On va au point non soudé le plus proche (disons p_2), et on soude.
- ...et on recommence jusqu'à ce que tout soit soudé.

Robot: un algo rapide

(Lazy) Pseudo-code "Plus proche voisin non soudé" (PPV)

1. Choisir un point initial p_1, le souder
2. $i := 1$
3. Tant qu'il reste un point non soudé faire
    4. $i := i + 1$
    5. $p_i :=$ point le plus proche de p_{i-1} et non soudé (si plusieurs $\rightarrow$ choisir comme on veut)
    6. Souder p_i
7. Fin Tant que
8. Retourner à p_1

Robot: un algo rapide

A propos de PPV

- PPV est un algorithme glouton (en anglais: *greedy*)
- $\rightarrow$ pas de retour sur une décision prise avant (en anglais: pas de *backtrack*)
- Idée: on optimise la distance localement

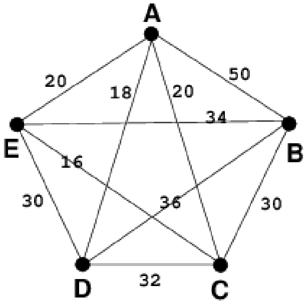
Est-ce qu'une série d'optimisations locales mène à une optimisation globale ?

Robot: un algo rapide

PPV est rapide... mais pas correct

- Réponse: parfois OUI, souvent NON:
  - Dijkstra (plus courts chemins): OUI
  - notre problème: NON
- Pas correct $\Rightarrow$ au moins une instance pour laquelle PPV ne renvoie pas l'optimal

PPV n'est pas correct



PPV: $A \xrightarrow{18} D \xrightarrow{30} E \xrightarrow{16} C \xrightarrow{30} B \xrightarrow{50} A$
Distance totale $= 144$

J'ai mieux! $A \xrightarrow{20} E \xrightarrow{16} C \xrightarrow{30} B \xrightarrow{36} D \xrightarrow{18} A$
Distance totale $= 120$

PPV: le bilan

- PPV est un algorithme rapide
- PPV n'est pas un algorithme correct

Remarque

Pour montrer qu'un algorithme

- est correct: $\rightarrow$ fonctionne sur toutes les instances autorisées
- n'est pas correct: $\rightarrow$ une seule instance qui ne fonctionne pas (contre-exemple)

Retour sur le coût d'un algorithme

Rappels

On s'intéresse:

- au coût en temps de l'algorithme $\rightarrow$ rapidité/lenteur
- au coût en mémoire de l'algorithme (= coût en espace)

A retenir !

Le coût (en temps/en mémoire) est une **fonction de la taille des données** d'entrée du problème

Taille des données

Exemple - Coût en temps

Quel que soit l'algorithme utilisé:

- Problème du robot: 10 points vs 1.000 points
- Problème PATTERN MATCHING: texte de taille $n = 10$ vs $n = 1.000.000.000$

Taille des données

- $\sim$ taille de l'instance d'entrée
- $\sim$ ce qu'il faut augmenter si on veut "compliquer" le problème (on y reviendra)

Coût en temps d'un algorithme

Comment mesurer le coût en temps ?

- Le coût en temps dépend de l'algorithme (bien sûr)
- Mais on ne veut pas qu'il dépende:
  - du langage de programmation
  - de la machine
- Raisons:
  - trop compliqué à calculer
  - c'est suffisant (on veut un ordre d'idée, une tendance)
- $\Rightarrow$ coût d'un algo à partir du pseudo-code
- en comptant les opérations élémentaires qui composent l'algo

Comment mesurer le coût en temps ?

Opérations élémentaires

- Affectation, comparaison, opération $(+, -, *, /)$: coût=1
- Accès à la mémoire: coût=1
- Attention: boucles et appels à des fonctions ne sont pas des opérations élémentaires !

$\Rightarrow$ intuitivement, opération élémentaire $=$ opération "de base" dans l'algorithme

Comment mesurer le coût en temps ?

- Coût en temps d'un algo = **nombre total** d'opérations élémentaires utilisées pour son exécution
- C'est une fonction de la **taille des données** d'entrée

Coût en temps d'un algorithme

Remarque

- C'est une simplification
  - Par ex., addition $\neq$ accès à un tableau $\neq$ division
  - Comment fonctionne précisément le processeur ? (assembleur)
  - $\rightarrow$ il y aura toujours une différence entre ce calcul et la réalité (chronomètre)
- Mais cette simplification permet de compter:
  - facilement (toute opération de base a le même coût)
  - indépendamment du langage de programmation
  - indépendamment de la machine
- et cela reflète bien la réalité ! (tendance générale)

Importance de la complexité en temps

- n=taille des données
- $f(n)$=nb d'opérations=coût en temps
- une opération prend 1 micro-seconde (μs)= 10^{-6}s
- 6 algos, chacun de complexité différente

n/f(n)	n	n^2	n^3	2^n	3^n	$n!$
10	$10\mu s$	$0.1ms$	$1ms$	$1ms$	$59ms$	$3.63s$
20	$20\mu s$	$0.4ms$	$8ms$	$1s$	58 min	77094 ans
40	$40\mu s$	$1.6ms$	$64ms$	12.73 j	385253 ans	$2.58 \cdot 10^{34}$ ans
60	$60\mu s$	$3.6ms$	$216ms$	36533 ans	$1.34 \cdot 10^{15}$ ans	$2.63 \cdot 10^{68}$ ans

Coût en espace d'un algorithme

Comment mesurer le coût en espace ?

- Coût en espace d'un algorithme = taille prise en mémoire par tout ce qui est stocké quand on exécute l'algorithme:
  - données d'entrée du problème
  - toute donnée supplémentaire stockée au cours de l'exécution
  - les données de sortie (si elles sont stockées)

Coût en espace d'un algorithme

Comment mesurer le coût en espace ?

- Coût en espace:
  - plus simple à calculer que le coût en temps
  - moins crucial de nos jours

    Apollo11 en 1969: 72Ko de mémoire morte + 4Ko de mémoire vive
  - attention quand même, il peut y avoir des pièges (on voit ça dans 2 transparents)

Retour sur la taille des données

Taille des données (d'entrée d'un algorithme)

- Formellement:
  Taille des données = Taille d'un meilleur codage pour ces données
- Meilleur codage: codage informatique qui prend le moins de place en mémoire
  - $\Rightarrow$ en info, penser binaire!
- Intuitivement: ce qui va faire augmenter le coût en temps de l'algorithme

Exercice

Taille des données

Quelle est la taille correspondant aux données d'entrée suivantes ?

1. un entier p (ex: problème de primalité d'un nombre)
2. deux entiers p et q (ex: problème du PGCD de deux nombres)
3. un ensemble de n entiers, dont le maximum est M
4. un texte de longueur n, exprimé sur un alphabet Σ de taille σ
5. un graphe G à n sommets et m arêtes

Analyse exacte d'algorithme

Calculer le coût exact: pas facile, pas utile

- Rappel: complexité = nb opérations élémentaires
- complexité = fonction f de la taille des données
- f peut être difficile à calculer
- But: dégager une tendance générale
- on va (encore) simplifier

Notations de Landau - le grand O

Idée: $f(n)$ est le coût précis de l'algorithme, $g(n)$ est une fonction plus simple

$O()$: définition intuitive

- $f(n) = O(g(n))$ veut dire que $c \cdot g(n)$ est une borne supérieure pour $f(n)$
  - $c > 0$ est une constante (indépendante de n)
  - on s'intéresse à des grandes valeurs de n (tendance quand n est grand)

$\Rightarrow$ on dit que $f()$ est dominée par $g()$ à une constante multiplicative $c > 0$ près

Notation de Landau - le grand O

Définition formelle

$f(n) = O(g(n))$ s'il existe des constantes strictement positives n_0 et c telles que pour tout $n \geq n_0$, $f(n) \leq c \cdot g(n)$

Astuce

Notations de Landau:

- partant de la fonction $f(n)$
- on ne garde que le terme dominant de $f(n)$
- on ignore la constante qui multiplie ce terme
- $\Rightarrow$ on obtient la fonction $g(n)$

$f(n) = 7n^2 + 3n \log n + 12\sqrt{n} - 7 \Rightarrow$

$$f(n) = O(n^2)$$

Robot soudeur: le retour

On essaye tout!

- Analyse exhaustive:
  - On génère toutes les suites de points possibles (permutations)
  - Pour chaque permutation, on calcule la distance
  - On retient la permutation qui donne la distance minimum
- Type d'algorithme appelé "brute force"

(Lazy) Pseudo-code *Brute Force*

1. $d = \infty$
2. **Pour** chaque permutation P de l'ensemble des points à souder **faire**
3. Si $distance(P) < d$ alors $d = distance(P)$ et $P_{min} = P$
4. Fin Pour
5. Renvoyer P_{min}

Robot: un algorithme correct

L'algo "Brute Force" est (forcément!) correct

- On essaye tout!
- Cet algorithme est donc correct

Coût en temps de "Brute Force"

- chaque ligne coûte un nb constant d'op. élémentaires $\rightarrow O(1)$
- mais lignes 2. et 3. exécutées de nombreuses fois
- Combien? $\rightarrow$ nb de permutations de taille n à tester, càd n!
- Complexité en temps de l'algo brute force: $O(n!)$

Brute force est une brute

Algo brute force impraticable (càd trop lent) sur un ordinateur standard, dès que $n \sim 15$

Test sur quelques valeurs de n

- Pour $n = 5$, combien de permutations à tester ? 120
- Pour $n = 10$, combien de permutations à tester ? 3.628.800
- Pour $n = 20$, combien de permutations à tester ? 2.43×10^{18}
- Pour $n = 60$, combien de permutations à tester ? 8.32×10^{81}
- Remarque: on estime à $\sim 10^{80}$ le nombre d'atomes dans l'univers...
- Impraticable, même avec un supercalculateur!

Coût en temps: un autre exemple

Problème PATTERN MATCHING

PATTERN MATCHING

Instance: un texte T de longueur n, un texte M de longueur m

Sortie: les positions de toutes les occurrences de M dans T

Algo à "fenêtre glissante"

- Rechercher le motif M à toutes les positions possibles dans T
- $\rightarrow$ positions possibles de M: de $T[1]$ à $T[n-m+1]$
- Dès que T et M diffèrent $\rightarrow$ décalage d'une position dans T

Coût en temps: un exemple

PATTERN MATCHING - Algorithme fenêtre glissante

```
void naive(M[1..m],T[1..n])
1. for i=1 to n-m+1 do
2.     j:=1;
3.     while M[j]=T[i+j] && j<= m do
4.        j:=j+1;
5.     endwhile
6.     if j=m+1 then
7.        printf(``Motif found at position %d\n'',i);
8.     endif
9. endfor
```

Algo à "fenêtre glissante"

- Recherche séquentielle dans T
- aussi appelé algo naïf
- Algo correct $\rightarrow$ motif cherché à chaque position possible de T
- Quel est son coût en temps ?

Coût en temps: un exemple

Rappel

Le coût en temps dépend de la taille des données

Taille des données ?

- Alphabet pour exprimer T et M: Σ, de taille σ
- T de longueur $n \rightarrow$ meilleur codage $n \log_2 \sigma$
- M de longueur $m \rightarrow$ meilleur codage $m \log_2 \sigma$
- σ considéré comme constant (ex: codage ASCII)
- $\Rightarrow$ taille des données $= n + m$

Coût en temps de PATTERN MATCHING

- chaque ligne (individuellement): nb constant d'op. élémentaires
- lignes 3. et 4. les plus coûteuses
- combien? dépend de l'instance!

On appelle cela la **forme des données**

- quelle forme des données au mieux ? quel coût associé ?
- quelle forme des données au pire ? quel coût associé ?

Analyse de l'algo naïf

Forme des données pour le coût au mieux

Coût au mieux $\rightarrow$ algo termine le plus vite

- lignes 3. et 4. exécutées le moins souvent possible
- $\rightarrow$ forme des données
  - pour tout i, $T[i] \neq M[1]$
  - (plus raisonnablement, $T[i] \neq M[1]$ ou $T[i+1] \neq M[2]$)

- coût associé $C_{mieux}(n, m)$
- lignes 3. et 4. exécutées au plus 2 fois pour chaque i coût associé $C_{mieux}(n, m) = O(n - m)$

$$\Rightarrow C_{mieux}(n, m) = O(n)$$

Analyse de l'algo naïf

Forme des données pour le coût au pire

Coût au pire $\rightarrow$ algo termine le plus lentement

- lignes 3. et 4. exécutées le plus souvent possible
- $\rightarrow$ forme des données
  - pour tout i, le motif apparaît à la position i
  - ex: $T = \underbrace{AAAA\ldots AAA}_{n}$ et $M = \underbrace{AA\ldots A}_{m}$

- coût associé $C_{pire}(n,m)$
- lignes 3. et 4. exécutées $\sim m$ fois pour chaque i coût associé $C_{pire}(n,m) = O((n-m)m)$

$$\Rightarrow C_{pire}(n,m) = O(nm)$$

Pire, Mieux, Moyenne

n = taille des données d'un algorithme A

- Coût au pire pour A = là où A exécute le maximum d'opérations élémentaires (quand la taille des données vaut n)
- Coût au mieux pour A = là où A exécute le minimum d'opérations élémentaires (quand la taille des données vaut n)
- Coût en moyenne = moyenne des opérations élémentaires pour exécuter A sur toutes les instances de taille n

Remarques:

- Coût au mieux/au pire → forme des données associée
- Erreur classique:

  "le cas au mieux, c'est quand $n = 1$" → c'est FAUX!!!

Exercice

Coût en espace de l'algo naïf

- Déterminer le coût en espace de l'algorithme naïf pour le problème PATTERN MATCHING
- T de longueur $n \rightarrow n \log_2 \sigma$
- M de longueur $m \rightarrow m \log_2 \sigma$
- variable $i \rightarrow \log_2 n$
- variable $j \rightarrow \log_2 m$

Retour sur Pire/Mieux/Moyenne

Le plus informatif: le coût au pire

- Coût au mieux pas très intéressant:
  - correspond à un nombre très limité de cas
  - ne donne aucune indication sur la valeur de l'algorithme
- Coût en moyenne pas facile à calculer: calcul du coût pour **chacune** des instances + faire la moyenne
- $\Rightarrow$ On préférera le coût au pire
  - Indique la robustesse de l'algo
  - Donne une garantie de performances

Faisons un point

Ce qu'il faut retenir

- La complexité en temps d'un algorithme se calcule:
  - en comptant le **nombre d'opérations** qu'il réalise
  - toujours en fonction de la **taille des données**
- Elle dépend aussi de la **forme des données** (notion de mieux, pire, moyenne)
- Le coût **au pire** sera souvent préféré: garantie de performance de l'algorithme

Sommaire

D'autres notations de Landau

- notation $f(n) = O(g(n)) \rightarrow g(n)$ est une borne supérieure (à une constante mult. près)
- besoin de deux notations supplémentaires:
  - borne inférieure (à une constante mult. près) $\rightarrow \Omega()$
  - $f(n)$ et $g(n)$ ont le même comportement (à une constante mult. près) $\rightarrow \Theta()$

$\Omega()$: définition intuitive

$\Omega()$: définition intuitive

- $f(n) = \Omega(g(n))$ veut dire que $c \cdot g(n)$ est une borne inférieure pour $f(n)$
  - $c > 0$ est une constante (indépendante de n)
  - grandes valeurs de n

$\Rightarrow$ on dit que $f()$ domine $g()$ à une constante multiplicative $c > 0$ près

$\Theta()$: définition intuitive

$\Theta()$: définition intuitive

- $f(n) = \Theta(g(n))$ veut dire que
  - $c_1 \cdot g(n)$ est une borne supérieure pour $f(n)$ et
  - $c_2 \cdot g(n)$ est une borne inférieure pour $f(n)$
  - $c_1 > 0$ et $c_2 > 0$ sont des constantes (indépendantes de n)
  - grandes valeurs de n

$\Rightarrow$ $g()$ et $f()$ ont le même comportement à des constantes multiplicatives près

Rem: $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \Omega(g(n))$ et $f(n) = O(g(n))$

Notations de Landau $\Omega()$ et $\Theta()$ - Définitions formelles

Notation Ω

$f(n) = \Omega(g(n))$ s'il existe des constantes strictement positives n_0 et c telles que pour tout $n \geq n_0$

$$f(n) \geq c \cdot g(n)$$

Notation Θ

$f(n) = \Theta(g(n))$ s'il existe des constantes strictement positives n_0, c_1 et c_2 telles que pour tout $n \geq n_0$

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Notations de Landau

Remarques

- Si $f(n) = \Omega(g(n))$, on dit que f domine g
- Si $f(n) = O(g(n))$, on dit que f est dominée par g
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$

Retour au coût des algorithmes

A quoi servent vraiment $\Omega()$ et $O()$?

- la fonction $f(n)$ varie avec la forme des données (coût au mieux/au pire)
  - coût au mieux $\Rightarrow$ borne inf. $\Rightarrow \Omega()$
  - coût au pire $\Rightarrow$ borne sup. $\Rightarrow O()$

Exemple Algo naïf PATTERN MATCHING

- $C_{mieux}(n, m) = \Omega(n)$
- $C_{pire}(n, m) = O(nm)$

Conclusion: l'algo naïf pour le problème PATTERN MATCHING est en $\Omega(n)$ et en $O(nm)$

Autre exemple

Recherche Dichotomique

- Recherche d'un élément x dans un tableau trié T de taille n
- Méthode: comparer x à $k = T[\frac{n}{2}]$
  - Si $x = k$, fin
  - Si $x < k$, recommencer sur $T_1 = T[1 \ldots \frac{n}{2} - 1]$
  - Si $x > k$, recommencer sur $T_2 = [\frac{n}{2} + 1 \ldots n]$
- $\Rightarrow$ A chaque itération, on peut ignorer la moitié du tableau...
- Et on itère le processus sur l'autre moitié

Autre exemple

- L'algorithme de dichotomie termine au bout de $\log_2 n$ itérations au pire
- Chaque itération: nombre constant d'opérations
- Recherche Dichotomique est en $O(\log n)$

- Question: coût au mieux ?
- Réponse: constant (qu'on note aussi $O(1)$)

Question

- Pourquoi la base du logarithme n’apparaît-elle pas dans la notation O?
- Réponse: elle n’a aucune importance pour les notations de Landau
- Explication:
  - la constante multiplicative n’apparaît pas dans les notations de Landau
  - passage d’une base à l’autre: en multipliant par une constante

Logarithmique, Polynomiale, Exponentielle

$n =$ taille des données

Fonctions

- Exponentielle : toute fonction en $O(c^n)$ (c=constante, $c > 1$)
- Polynomiale : toute fonction en $O(n^{c'})$ (c'=constante)
- Logarithmique : toute fonction de la forme $O(\log n)$
- (il y a aussi Polylogarithmique : exemple $O(n \log n)$)

Logarithme, Polynôme, Exponentielle

La morale de l'histoire

- Toute fonction exponentielle domine toute fonction polynomiale
- Toute fonction polynomiale domine toute fonction logarithmique
- En ce qui concerne le coût des algorithmes, on évitera tout ce qui exponentiel

Pourquoi? $\rightarrow$ voir transparent suivant

Exponentiel vs Polynomial

Pourquoi cherche-t-on à éviter l'exponentielle ?

n/f(n)	n	n^2	n^3	2^n	3^n	$n!$
10	$10\mu s$	$0.1ms$	$1ms$	$1ms$	$59ms$	$3.63s$
20	$20\mu s$	$0.4ms$	$8ms$	$1s$	58 min	77094 ans
40	$40\mu s$	$1.6ms$	$64ms$	12.73 j	385253 ans	$2.58 \cdot 10^{34}$ ans
60	$60\mu s$	$3.6ms$	$216ms$	36533 ans	$1.34 \cdot 10^{15}$ ans	$2.63 \cdot 10^{68}$ ans

Sommaire

Itératif vs Récursif

Comparaison

- Analyse d'algorithme itératif $\rightarrow$ calculs "simples"
- Analyse d'algorithme récursif $\rightarrow$ pas toujours évident
- Heureusement, il y a le **Master Theorem**

On a fait le calcul pour vous

Master Theorem

Soit $C(n)$ une fonction telle que

$$C(n) \leq a \cdot C(\frac{n}{b}) + f(n)$$

$$C(1) = c$$

où $a \geq 1$, $b \geq 2$, $c \geq 0$.

Si $f(n) = \Theta(n^d)$ avec $d \geq 0$, alors

1. $C(n) = O(n^d)$ si $a < b^d$
2. $C(n) = O(n^d \cdot \log n)$ si $a = b^d$
3. $C(n) = O(n^{\log_b a})$ si $a > b^d$

Master Theorem - Quelques exemples

Remarque:

- n est la taille des données du problème étudié
- $C(n)$ est la complexité au pire de l'algo récursif étudié

Exemple 1

Supposons $C(n) \leq C(\frac{n}{2}) + k$ où $k =$constante

$$a = 1$$

$$b = 2$$

$$d = 0$$

$\Rightarrow$ Cas 2 du Master Theorem
$\Rightarrow$ Complexité en $O(\log n)$
Exemple: Recherche Dichotomique

Master Theorem - Quelques exemples

Exemple 2

Supposons $C(n) \leq 8C(\frac{n}{2}) + n^2$

$$a = 8$$

$$b = 2$$

$$d = 2$$

$\Rightarrow$ Cas 3 du Master Theorem

$\Rightarrow$ Complexité en $O(n^3)$

Sommaire

Complexité en temps du TRI

TRI

Instance: une suite S contenant n réels

Sortie: les n réels de S triés par ordre croissant

Taille des données pour le TRI

- n valeurs, plus grand élément $= N$
- Taille des données: $n \log_2 N$
- Rem: valeur de N peu importante, c'est l'ordre relatif des n valeurs qui compte
- exemple: $S_1 = [8, 4, 7, 6, 5]$ et $S_2 = [8000, 4000, 7000, 6000, 5000]$ se trient de la même façon
- $\Rightarrow$ complexité ne dépend pas de N

Taille des données $= n$

Complexité en temps du TRI

Meilleur/s algo/s pour résoudre le problème du TRI:

$$O(n \log n)$$

Remarques

- Complexité au pire (d'où le $O()$)
- Plusieurs algos possibles:
  - Tri fusion
  - Tri par tas
  - ...et quelques autres

Complexité en temps pour 3 autres routines

Structures de Données (SD)

- Les grands classiques:
  - Tableaux non triés
  - Tableaux triés
  - Listes chaînées non triées
  - Listes chaînées triées
- Question: quelle complexité (en temps) pour des routines standard ?

Comparatif SD

Les routines de base

S = la structure considérée

$S[i]$ = l'élément stocké en i-ème position de S

S contient n éléments

1. `Rechercher(`S, x`)` : renvoie la position i de S telle que $S[i] = x$ ($i = -1$ si x n'appartient pas à S)
2. `Insérer(`S, x`)` : insère x dans S
3. `Min(`S`)` : renvoie la position i t.q. $S[i]$ est le plus petit élément de S

$\texttt{Rechercher}(S, x)$

S = la structure considérée
x = l'élément à rechercher

$\texttt{Rechercher}(S, x)$	Au mieux	Rem.	Au pire	Rem.
T non trié	constant	1er élément cherché $= x$	$O(n)$	rech. séquentielle
T trié	constant	1er élément cherché $= x$	$O(\log n)$	dichotomie
LC non triée	constant	1er élément cherché $= x$	$O(n)$	rech. séquentielle
LC triée	constant	1er élément cherché $= x$	$O(n)$	rech. séquentielle (ou dichotomie)

Rappel: constant s'écrit aussi O(1)

Insérer(S, x)

S = la structure considérée
x = l'élément à insérer

Insérer(S, x)	Au mieux	Rem.	Au pire	Rem.
T non trié	$O(1)$: insertion en fin de tableau			
T trié	$O(1)$	fin de tableau	$O(n)$	début de tableau
LC non triée	$O(1)$: insertion en début de liste			
LC triée	$O(1)$	début de liste	$O(n)$	fin de liste

$\texttt{Min}(S, x)$

S = la structure considérée

$\texttt{Min}(S)$	Au mieux	Rem.	Au pire	Rem.
T non trié	$\Theta(n)$: rech. séquentielle			
T trié	$O(1)$: 1er élément du tableau			
LC non triée	$\Theta(n)$: rech. séquentielle			
LC triée	$O(1)$: 1er élément de la liste			

Bilan

Question

Quelle est la meilleure structure ?

Réponse

Ça dépend...

- des opérations que l'on va réaliser le plus souvent
- de la valeur de n lors d'une utilisation concrète
- $\Rightarrow$ donc de l'application qui utilisera cette structure