IFT2125 - Introduction à l'algorithmique

Introduction

Pierre McKenzie

DIRO, Université de Montréal

Hiver 2018

Documentation

Livre obligatoire :

- Brassard et Bratley, *Fundamentals of algorithmics*, Prentice Hall 1996.

Autres :

- Cormen, Leiserson, Rivest, *Introduction à l'algorithmique*, 1994 ou +. Édition anglaise de 2009 comporte 4ième auteur, Stein
- Kleinberg, Tardos, *Algorithm Design*, 2006
- Transparents, notes, références ponctuelles placées sur Studium
- Web
- Bibliothèque (réserve, nombreux bouquins)

Évaluation

Devoirs :

- 4 devoirs de 4 questions
- chaque question vaut 10 indépendamment de sa difficulté
- questions pas toutes corrigées (choix aléatoire)
- équipes de 2 recommandées

Examens intra et final :

- Livre fermé
- Final cumulatif

Barême :

- Intra 30%, final 40%, devoirs 30%.
  Seuil à 40%.
- Doctorants en examen prédoctoral : intra 40%, final 60%.
  Pour les doctorants : les devoirs ne comptent pas.

Auxiliaire d'enseigmenent

Stéphanie Larocque
stephanie.larocque@umontreal.ca

Tâches :

- anime les séances de travaux pratiques
- corrige les devoirs
- répond aux questions
- disponible sur rendez-vous

BB = Livre de Brassard et Bratley

Utile d'avoir vu

Préalable IFT1065 - Mathématiques discrètes :

- Induction mathématique [BB 1.6]
- Logique, propositionnelle, des prédicats [BB 1.4.5]
- Permutations, combinaisons [BB 1.7.3]
- Arithmétique modulaire, polynômes
- Définitions de $O(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$ [BB 1.7.2, 3.2, 3.3]
- Récurrences simples et linéaires homogènes [BB 4.7.2]

Cours de prog. et concomitant IFT2015 - Structures de données :

- Recherche dichotomique
- Quelques tris
- Python (?)

Concomitant IFT1978 - Probabilités et statistique :

- Probabilités de base (BB 1.7.4)

Plan approximatif

Heures de cours	Matière
6	Introduction et exemples (en partie hors livre)
6	Compléments sur les ordres et les récurrences (chapitres 3 et 4)
6	Algorithmes voraces (chapitre 6)
6	Diviser pour régner (chapitre 7)
5	Programmation dynamique (chapitre 8)
3	Exploration de graphes (chapitre 9)
4	Algorithmes probabilistes (chapitre 10)
1	Algorithmes parallèles (chapitre 11)
TOTAL : 37	

Prendre connaissance du Code d'honneur de l'étudiant du DIRO. En particulier,

- citez toute source d'information utilisée dans vos travaux
- remettre un devoir en équipe engage la responsabilité de l'équipe.

Pour plus d'information sur les règlements de l'université, consultez Intégrité à l'Université de Montréal.

Questions ?

L'algorithmique, c'est quoi ?

- concevoir des méthodes efficaces de résolution de problèmes de calcul
- choisir la méthode appropriée pour un problème donné

Beaucoup d'intelligence au fil des ans consacrée à l'algorithmique !

Exemples

Tri d'un tableau

1. Sélection
2. Insertion
3. Merge sort - par fusion
4. Quick sort - rapide
5. Heap sort - par tas
6. Radix sort - par base
7. Bucket sort - par paquets
8. Tri en parallèle

Exemples

Déterminant d'une matrice $m \times m$

$$\begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} = x_{11}x_{22}x_{33} - x_{11}x_{32}x_{23} - x_{12}x_{21}x_{33} + \cdots$$

$$= \sum_{\sigma \in S_m} (-1)^{\text{signe de } \sigma} x_{1\sigma(1)}x_{2\sigma(2)} \cdots x_{m\sigma(m)}$$

1. Bête
   Faire la somme des $m!$ termes.
2. Gauss-Jordan
   Amener à la forme triangulaire.
   Multiplier les éléments de la diagonale.
3. Berkowitz (Samuelson)
   Réduire au calcul de puissances de matrices.

Recherche en cours : et le permanent d'une matrice ?

Exemples

Déterminer la primalité

On veut déterminer si $\underbrace{xxxxxxxxxxxx}_{12 \text{ chiffres}}$ est un nombre premier.

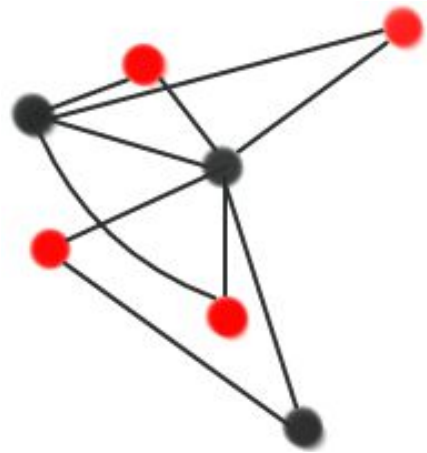
Problème addictif pour mathématiciens.
Problème fondamental pour les cryptographes.

1. Bête
   Essayer diviseurs $2, 3, 4, 5, 6, 7, 8, \ldots$ éventuellement jusqu'à 10^6
2. Crible d'Erathostènes
   Éliminer tour à tour diviseurs de la liste $2, 3, 4, 5, 6, 7, 8 \ldots, 10^6$
3. Miller-Rabin
   Rapide en acceptant une probabilité d'erreur inférieure à $2^{-1000000}$.
4. Agrawal-Keyal-Saxena (2002)
   Temps polynomial avec certitude mais degré élevé.

Exemples

Stable maximum

Donnée : graphe (S, A)



Déterminer : un stable de taille maximum.

1. Bête
   Essayer tous les $E \subseteq S$ en ordre décroissant de taille.

Recherche en cours : trouver une méthode qui n'est pas bête.

- concevoir des méthodes efficaces de résolution de problèmes de calcul
- choisir la méthode appropriée pour un problème donné

Au fait, qu'est-ce qu'un problème ?

Problèmes et exemplaires ("problems and instances")

Un problème demande de

- calculer une valeur
  - ex : tri, déterminant
- ou de répondre à une question oui/non
  - ex : le nombre est premier ?, il existe un stable de taille k ?

à partir de données fournies en entrée.

Un problème possède une infinité d'exemplaires

- ex : tri
  - tableau 1, tableau 2, etc.
- ex : nombre premier
  - $0, 1, 2, \ldots$, etc.
- ex : stable
  - $\ldots$  $\ldots$

Le temps

Un algo A résout un problème P.

- A peut prendre un temps différent sur chaque exemplaire
- Qu'est-ce alors que le temps d'exécution de A ?

Simplification fréquente :

- paramétriser en fonction de la taille n des exemplaires

Et qu'est-ce que taille n d'un exemplaire ?

Ultimement, n = nombre de bits utilisés pour coder l'exemplaire.

En pratique, dépend du problème P et du but de l'analyse :

- ex : tri
  - souvent n = nombre d'éléments du tableau
- ex : évaluer une expression comme $((28783 + 410)/192) \times 159$
  - souvent n = nombre d'opérandes, ici $n = 4$
  - parfois n = nombre total de chiffres, ici $n = 14$.
- ex : stable
  - parfois n = nombre $|S|$ de sommets du graphe (S, A)
  - parfois $n = |S|^2$ = nombre de bits requis pour représenter la matrice d'adjacence du graphe

Diverses mesures de temps

- Pire cas (mesure la plus utilisée) :
  $t(n) = \max_{e \text{ exemplaire de taille n}} \{\text{temps que prend } A \text{ sur } e\}$

- En moyenne
  $$t(n) = \frac{\sum_{e \text{ exemplaire de taille } n} \{\text{temps que prend } A \text{ sur } e\}}{\text{nombre d'exemplaires de taille } n}$$

- Amorti (cas d'un A qui agit sur des données externes)
  Moyenne sur plusieurs appels successifs à A

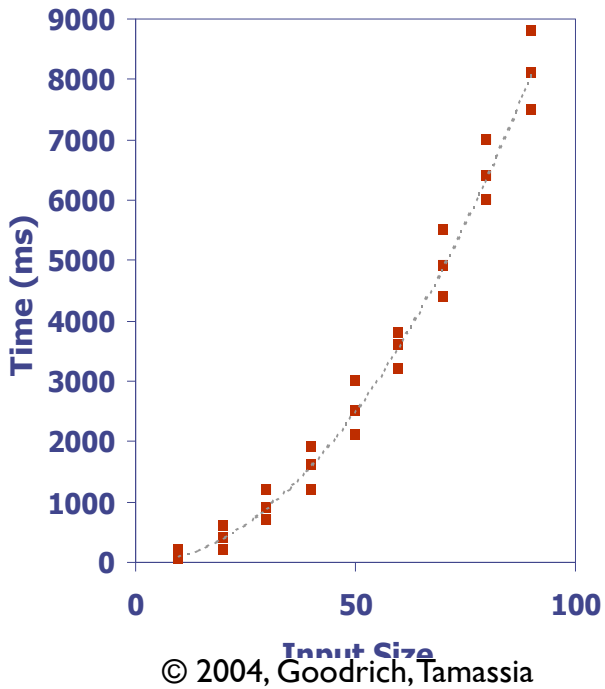
- Espéré (cas d'un A qui utilise l'aléat)
  Espérance mathématique du temps avant l'arrêt.

4 slides empruntées de Sylvie Hamel

Comment obtenir le temps $t(n)$ d'un algorithme ?

Méthode 1: Études expérimentales

- Implémenter l'algorithme en Java (ou autre)
- Faire fonctionner le programme avec des entrées de taille et de composition différentes
- Utiliser une méthode pour obtenir une mesure réelle du temps d'exécution
- Dessiner le graphique des résultats



Limitation de cette méthode

- On doit implémenter l'algorithme
  - On veut connaître la complexité en temps d'un algorithme avant de l'implémenter, question de sauver du temps et de l' $$$$
- Les résultats trouvés ne sont pas représentatifs de toutes les entrées
- Pour comparer 2 algorithmes différents pour le même problème, on doit utiliser le même environnement (hardware, software)

Méthode 2 : analytique

Compter les opérations élémentaires :

- Opérations de base effectuées par l'algorithme, par ex. :
  - Évaluer une expression
  - Affecter une valeur à une variable
  - Appeler une méthode
  - Incrémenter un compteur
  - etc.
- Indépendantes du langage de programmation choisi
- On suppose que chacune prend un temps d'exécution constant

Exemple : max d'un tableau

Compter les opérations élémentaires

En inspectant le pseudocode d'un algorithme, on peut déterminer le nombre maximum d'opérations élémentaires exécuté par un algorithme, comme une fonction de la taille de l'entrée

Algorithm *arrayMax*(A, n)	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	2 (n-1)
if $A[i] > currentMax$ **then**	2 (n-1)
$currentMax \leftarrow A[i]$	2 (n-1)
return $currentMax$	1
	total = 6n - 3

Du nombre d'opérations élémentaires au temps

- On cherche $t(n)$ = temps en pire cas, lorsque n = taille du tableau
- L'algo *arrayMax* exécute $6n - 3$ opérations élémentaires en pire cas
  - a = temps d'exécution de la plus rapide opération élémentaire
  - b = temps d'exécution de la plus lente opération élémentaire
- Alors le temps en pire cas $t(n)$ de *arrayMax* vérifie :

$$\boxed{\forall n,\ a \times (6n-3) \leq t(n) \leq b \times (6n-3)}$$

- Souvent le comportement asymptotique suffit. Ici :
  - de $\leq$ on tire $t(n) \in O(n)$
  - de $\geq$ on tire $t(n) \in \Omega(n)$
  - $t(n) \in O(n) \cap \Omega(n)$ d'où $t(n) \in \Theta(n)$

- arrayMax est particulier en ce que le pire cas est facile à identifier

Autres exemples de l'intelligence consacrée à l'algorithmique

Multiplication de grands entiers

Plusieurs méthodes pour calculer $xxxxxxxxxxxx \times yyyyyyyyyyyy$

1. Classique
2. À la “façon russe”
3. À la “façon arabe”

Multiplication de grands entiers

Plusieurs méthodes pour calculer $xxxxxxxxxxxx \times yyyyyyyyyyyy$

1. Classique
2. À la "façon russe"
3. À la "façon arabe"
4. Récursive

   Exprimer $xxxxxxxxxxxx = A \times 10^6 + B$

   Exprimer $yyyyyyyyyyyy = C \times 10^6 + D$

   Calculer $AC \times 10^{12} + (AD + BC) \times 10^6 + BD$.

Multiplication de grands entiers

Plusieurs méthodes pour calculer $xxxxxxxxxxxx \times yyyyyyyyyyyy$

1. Classique
2. À la "façon russe"
3. À la "façon arabe"
4. Récursive

   Exprimer $xxxxxxxxxxxx = A \times 10^6 + B$

   Exprimer $yyyyyyyyyyyy = C \times 10^6 + D$

   Calculer $AC \times 10^{12} + (AD + BC) \times 10^6 + BD$.
5. Interpolation

   Calculer $xxxxxxxxxxxx \times yyyyyyyyyyyy$ modulo 2

   Calculer $xxxxxxxxxxxx \times yyyyyyyyyyyy$ modulo 3

   Calculer $xxxxxxxxxxxx \times yyyyyyyyyyyy$ modulo 5

   $\vdots \qquad \vdots \qquad \vdots$

   Calculer $m \leq 2 \times 3 \times 5 \times \cdots$ vérifiant ces congruences.

   Théorème ("des restes chinois") : Ce m (positif) est unique.

Plus grand commun diviseur

On cherche $pgcd(xxxxxxxxxxxx, yyyyyyyyyyyy)$.
Ex : $pgcd(140, 98) = 2 \times pgcd(70, 49) = 2 \times 7 \times pgcd(10, 7) = 2 \times 7 = 14$.

1. Bête
   Essayer $xxxxxxxxxxxx, xxxxxxxxxxxx - 1, \ldots$ éventuellement jusqu'à 2
2. Euclide

```
def pgcd(a,b):
  while b != 0:
    a,b = b, a % b
  return(a)
```

Recherche en cours : méthode efficace en parallèle ?

Plus grand commun diviseur

Comment s'assurer que

```
def pgcd(a,b):
  while b != 0:
    a,b = b, a % b
  return(a)
```

est un algorithme correct ?

- rarement simple
- demande ingéniosité

Tirer davantage de l'algorithme d'Euclide

Algorithme d'Euclide étendu

```
def pgcd_etendu(a,b):
    """ Calcule s, t et pgcd(a,b)=sa+tb """
    s, t, u, v = 1, 0, 0, 1
    while b != 0:                    # invariant : sa+tb=a
        q = a//b
        a,s,t,b,u,v = b,u,v, a-q*b, s-q*u, t-q*v
    return(s,t,a)
```

Transformée de Fourier

Étant donnés une matrice M de forme particulière et un vecteur x, on veut calculer les vecteurs Mx et $M^{-1}x$.

1. Bête
   Multiplier sans se soucier de la forme particulière.
2. Diviser pour régner
   A révolutionné les télécommunications et le traitement des signaux.
   Sous-tend le format JPEG.

Appartenance à un groupe de permutations

Rappel : permutations d'un ensemble $\{1, 2, 3, 4, 5, 6\}$ de "points"

- $\varepsilon = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \in S_6$ est la permutation identité
- $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 2 & 6 & 5 & 1 & 3 \end{pmatrix} \in S_6$ est aussi représentée $(145)(36)$
- Le produit de deux permutations :
  $(145)(26) * (134)(256) = (1)(2)(3465) = (3465)$
- L'inverse d'une permutation :
  $[(145)(26)]^{-1} = (541)(62) = (154)(26)$

Le problème de l'appartenance

Donnée : $p, p_1, \ldots, p_k \in S_m$
Déterminer : si $p \in \underbrace{\langle p_1, \ldots, p_k \rangle}_{\text{toutes les permutations engendrées par composition des } p_i}$.

1. Bête
2. Intelligent
   Prochains transparents.
3. Super-intelligent
   Rapide en parallèle, repose sur 5000 pages de mathématiques.

Appartenance : l'algo bête

$S \leftarrow \emptyset$
$S' \leftarrow \{p_1, p_2, \ldots, p_k\}$

`while` $S \neq S'$
$\quad S \leftarrow S'$
$\quad S' \leftarrow S' \cup \{s * t : s, t \in S\}$

`if` $p \in S$ `then` TRUE `else` FALSE

Appartenance : l'algo intelligent

Coeur de l'algo : tamiser une permutation dans un tableau en construction

T : tableau $m \times m$ de permutations des points $\{1, 2, \ldots, m\}$
r : permutation à traiter

tamiser(r)

```
while r ≠ ε
    i ← min{i : i^r ≠ i}      {i ← plus petit point déplacé par r}
    j ← i^r                   {j ← le point où r envoie i}
    if T[i,j] == ε then
        T[i,j] ← r            {insérer r dans le tableau}
    else
        r ← r * (T[i,j])^-1
```

Appartenance : l'algorithme intelligent complet

Donnée : $p, p_1, \ldots, p_k \in S_m$
Déterminer : si $p \in \langle p_1, \ldots, p_k \rangle$.

fill $m \times m$ table T everywhere with ε

for $i = 1, \ldots, k$
    tamiser(p_i)

`while` there exist q, r in table T such that $q * r$ was never sifted
    tamiser($q * r$)

`if` tamiser(p) modifies T then
    "p n'appartient pas au groupe $\langle p_1, \ldots, p_k \rangle$"
`else`
    "p appartient au groupe $\langle p_1, \ldots, p_k \rangle$"

Appartenance : l'algorithme intelligent

Cet algorithme est-il correct ? ? ?

Appartenance

Principale propriété du tamisage de r, $r \neq \varepsilon$

Supposons :

- r vient d'être tamisé
- s était le plus petit point non fixé par r
- $T[t, j] =$ fut la dernière entrée examinée lors du tamisage de r.

Alors :

- $s \leq t$
- r s'exprime maintenant sous la forme

$$T[t, j] * T[t-1, j_{t-1}] * \cdots * T[s+1, j_{s+1}] * T[s, j_s].$$

Preuve : induction sur le nombre de tours du `while` lors du tamisage.

Problème de l'ordre d'un groupe de permutations

On l'a gratuitement du tableau T

Donnée : permutations $p_1, p_2, \ldots, p_k$
Déterminer : nombre de permutations du groupe $\langle p_1, p_2, \ldots, p_k \rangle$

former T en tamisant $p_1, p_2, \ldots, p_k$ puis en "fermant" T
$\mathtt{N} \leftarrow 1$
`for` $i = 1, \ldots, m$
    $\ell \leftarrow |\{j : T[i,j] \neq \varepsilon\}|$
    $\mathtt{N} \leftarrow \mathtt{N} \times (\ell + 1)$
`return` N