

Algorithmique Avancée et Complexité: Présentation du cours AAC

Sophie Tison-USTL-Master1 Informatique

Objectifs

Avoir des outils pour concevoir un "bon" -c.à.d. **correct et efficace** - algorithme pour résoudre un problème.

Cela pose de nombreuses questions...et
demande pas mal de savoir-faire...

Existe-il un algorithme pour résoudre le problème?

Connaître quelques Notions de Calculabilité et de décidabilité.

Cela pose de nombreuses questions...et
demande pas mal de savoir-faire.

Est-ce un problème classique?

Connaître et savoir reconnaître des grands classiques

Tris, méthodes de sélection, recherche
algorithmique des graphes,
méthodes de hachages
Programmation linéaire...

....

Cela pose de nombreuses questions...et demande pas mal de savoir-faire

Comment concevoir un algorithme?

Schémas d'algorithmes, Algorithmic design patterns

"Diviser pour régner"
Programmation Dynamique
Algorithmes gloutons

Cela pose de nombreuses questions...et demande pas mal de savoir-faire.

L'algorithme est-il correct?

Savoir prouver un algorithme...
ou tout du moins avoir un minimum de rigueur

Cela pose de nombreuses questions... et demande pas mal de savoir-faire.

L'algorithme est-il efficace?

Savoir analyser la complexité d'algorithmes

Cela pose de nombreuses questions...et demande de multiples compétences.

Peut-on trouver un algorithme plus efficace pour le problème?
Est-ce un problème dur?

...Avoir quelques notions de Complexité des problèmes

Cela pose de nombreuses questions...et demande pas mal de savoir-faire.

Si le problème est dur, comment l'appréhender!

Connaître quelques techniques d' Algorithmique Avancée:
Méta-heuristiques, Algorithmes probabilistes,...
Backtracking, minmax, séparation-évaluation

Ce que nous ferons

Quelques schémas d'algorithmes (2-3 cours)

Un peu de complexité de problèmes (2-3 cours)

Un peu d'algorithmique avancée (2-3-4 cours)

Quelques notions de décidabilité et calculabilité (1-2 cours)

Bibliographie : de nombreuses ressources en ligne

- . Le dictionnaire recensant les algorithmes et problèmes classiques du NIST
- . The Stony Brook Algorithm Repository" qui contient des implémentations d'algorithmes pour des dizaines de problèmes classiques,
- . Algorithms Courses on the WWW, qui contient une collection de cours d'algorithmique,
- . Le site du cours "Algorithms in the Real World",
- . "A compendium of NP optimization problems":

Bibliographie : de nombreux livres

- . Cormen, Leiserson, Rivest, "Introduction à l'algorithmique", Dunod (disponible à la BU) vraiment "une" référence essentielle en algorithmique,
- . S. Skiena, "Algorithm Design Manual", une "mine"! (Une version on-line proche du livre papier),
- . Sur l'aspect "algorithmic pattern", "Data Structures and Algorithms with object-oriented design patterns in Java". 2000, disponible sur le Web , ...

Organisation: les TPs

Il y aura 6 séances de TP (langage : Java) encadrées pour la mise en oeuvre directe des méthodes étudiées en cours:

- . Programmation dynamique, Algorithmes gloutons(2 séances)
- . Propriétés NP, réductions polynômiales (2 séances)
- . Heuristiques, Métaheuristiques ou Calculabilité (2 séances)

Organisation.... :l'évaluation

Le contrôle continu sera basé sur les TPs et un DS en milieu de semestre. La note de contrôle continu sera $1/3 * (note DS) + 2/3 * (note TP)$ avec éventuellement un bonus donné par des "devoirs maison".

Quelques exemples

- .Problème 1: Trouver le plus court chemin entre deux sommets d'un graphe
- .Problème 2: Trouver le chemin le plus long -sans cycle- dans un graphe

Quelques exemples

.Problème 1: Trouver un chemin qui passe une et une seule fois par tous les sommets d'un graphe

.Problème 2: Trouver un chemin qui passe une et une seule fois par tous les arcs d'un graphe

Pour quel problème peut-on trouver un algorithme efficace?

Si vous trouvez un algorithme polynômial pour le 1, vous gagnez 1 million de dollars.

Si vous montrez qu'on ne peut pas en trouver un, vous gagnez aussi 1 million de dollars!

Quelques exemples

.Problème 1: Planifier n cours dont les horaires sont donnés dans k salles

. Problème 2: Planifier n cours dans k créneaux, en connaissant les incompatibilités entre deux cours

Complexité des algorithmes:
quelques rappels

Analyser la complexité d'un algorithme c'est:

Evaluer les ressources utilisées par l'algorithme, plus précisément

Exprimer le coût de l'algorithme - la quantité de ressources utilisées - en fonction de la taille des données

Quelles ressources?

la mémoire: *complexité spatiale*

le temps : *complexité temporelle*

ou le nb de processeurs, les communications,....

Comment mesurer la taille des données

.a priori la taille des données est **le nombre de bits de leur représentation en binaire**;

. même si dans certains cas, on étudie la complexité en fonction d'une autre notion de taille, comme le nombre d'éléments pour une liste, la dimension pour une matrice, ...

Comment mesurer la taille des données

Q? Quelle est la taille d'un entier n ?

$$\log_2 n$$

Quel est le coût de l'algorithme A pour une d : $cout_A(d)$?

. Pour évaluer ce coût indépendamment de la machine (physique) , on utilise un *modèle* de machines séquentielles "classiques", en général la RAM ("Random Access Machine").

. Chaque instruction "simple" (+, *, -, affectation, comparaison, appel, accès mémoire ...) a un coût de **1**: coût uniforme, i.e. indépendant de la taille de ses opérandes par opposition au coût logarithmique où on tient compte de la taille des données.

. On se borne souvent à "compter" certaines instructions: comparaisons pour un tri par comparaisons, accès à la mémoire externe pour un tri externe, opérations arithmétiques de base pour des produits de matrices...

Quelle complexité?

Pour deux données de taille n , le coût peut être différent!

- *La complexité dans le meilleur des cas:*

$$Inf_A(n) = \inf\{cout_A(d)/d \text{ de taille } n\}$$

- *La complexité dans le pire des cas:*

$$Sup_A(n) = \sup\{cout_A(d)/d \text{ de taille } n\}$$

- *La complexité en moyenne:* pour la définir, il faut disposer pour tout n d'une mesure de probabilité p sur l'ensemble des données de taille n ;

$$Moy_A(n) = \sum_{d \text{ de taille } n} p(d) * cout(d)$$

En résumé...

Quand on parle de la complexité d'un algorithme sans préciser laquelle, c'est souvent de la complexité temporelle dans le pire des cas qu'on parle.

Ordres de grandeur...

On ne calcule pas en général la complexité exacte mais on se contente de calculer son *ordre de grandeur asymptotique* voire de borner celui-ci.

Par exemple, si on fait $n^2 + 2n - 5$ opérations, on retiendra juste que l'ordre de grandeur est n^2 . On utilisera donc les notations classiques sur les ordres de grandeur:

Ordres de grandeur: définitions

Soient f et g deux fonctions de \mathcal{N} dans \mathcal{R} :

. $f \in O(g)$ *Ssi*_{def} $\exists c \in \mathcal{R}^{+*}, \exists A$ tels que
 $\forall n > A, f(n) \leq c * g(n)$

On dit que f est dominée asymptotiquement par g . On notera souvent $f = O(g)$

. $f \in \Theta(g)$ *Ssi*_{def} $\exists c, C \in \mathcal{R}^{+*}, \exists A$, tels que
 $\forall n > A, C * g(n) \leq f(n) \leq c * g(n)$

On dit alors que f et g sont de même ordre de grandeur asymptotique. On notera souvent $f = \Theta(g)$.

. $f \in o(g)$ *Ssi*_{def} $\forall \epsilon \in \mathcal{R}^{+*}, \exists A$, tels que
 $\forall n > A, f(n) \leq \epsilon * g(n)$

On dit que f est négligeable asymptotiquement devant g . On notera souvent $f = o(g)$

Ordres de grandeur: exemples

$$. n^3 + 3n + 7 \in \theta(n^3)$$

$$. 5 * n^3 + 3n + 7 \in \theta(n^3)$$

$$. 5 * n^3 + 3n + 7 \in O(n^3)$$

$$. 5 * n^3 + 3n + 7 \in O(n^4)$$

$$. \log n \in o(n)$$

Remarque

Evaluer l'ordre de grandeur asymptotique du coût de l'algorithme en fonction de la taille des données... plutôt qu'à la complexité exacte se justifie si les données manipulées sont de grande taille.

Il ne faut pas négliger trop vite les constantes .

Q?: à partir de quelle taille de données, un algorithme de complexité exacte $20 * n * \log_2 n$ sera-t-il plus intéressant qu'un algorithme en n^2 ?.

Un peu de vocabulaire: un algorithme est dit...

- . en temps **constant** si sa complexité dans le pire des cas est bornée par une constante.
- . **linéaire** (resp. linéairement borné) si sa complexité (dans le pire des cas) est en $\Theta(n)$ (resp. $O(n)$).
- . **quadratique** (resp. au plus quadratique) si sa complexité (dans le pire des cas) est en $\Theta(n^2)$ (resp. en $O(n^2)$).
- . **polynômial** ou polynômialement borné, si sa complexité (dans le pire des cas) elle est en $O(n^p)$ pour un certain p .
- . (au plus) **exponentiel** si elle est en $O(2^{n^p})$, pour un certain $p > 0$.

Etre praticable ou ne pas être praticable?...

Par "convention", un algorithme est dit **praticable** si il est polynômial, c.à.d. si sa complexité temporelle dans le pire des cas est polynômiale.

Les limites de la convention:

- Si un algorithme est exponentiel dans le pire des cas, il peut être polynômial en moyenne; si les pires cas sont exceptionnels, il peut être praticable ...en pratique: certains algos impraticables selon la définition ci-dessus sont ... pratiqués tous les jours (comme le simplexe).
- L'exécution d'un algorithme dont la complexité moyenne est de l'ordre de grandeur de n^5 microsecondes prendrait 30 ans si $n = 1000$.

Mais ce n'est pas une si mauvaise convention:

- + En pratique, il s'avère que la plupart des algorithmes polynômiaux ont un comportement asymptotique équivalent à un polynôme de faible degré, 2 ou 3.
- + De plus, la classe des algorithmes polynomiaux a de bonnes propriétés de clôture (par exemple, une "séquence" de deux algorithmes polynomiaux est polynomiale).
- + Enfin, et surtout, elle est indépendante du modèle séquentiel "classique" choisi: un algorithme polynômial pour le modèle des machines de Turing, le sera pour une RAM et vice-versa .

Ordres de grandeur: exemples

Exemples de temps d'exécution en fonction de la taille de la donnée et de la complexité de l'algorithme, si on suppose qu'une instruction est de l'ordre de la μs ;

T.\C.	$\log n$	n	$n \log n$	n^2	2^n
10	$3\mu s$	$10\mu s$	$30\mu s$	$100\mu s$	$1000\mu s$
100	$7\mu s$	$100\mu s$	$700\mu s$	$1/100s$	10^{14} siècles
1000	$10\mu s$	$1000\mu s$	$1/100s$	$1s$	astronomique
10000	$13\mu s$	$1/100s$	$1/7s$	$1,7mn$	astronomique
100000	$17\mu s$	$1/10s$	$2s$	$2,8h$	astronomique

Quelques exemples:

Quelle est la complexité de:

```
int T1(int n){
  int c=0;
  for (int j=0;j<n;j++)
    for (int i=1;i<n;i++)  c++;
  return c;}
```

Quelques exemples:

Quelle est la complexité de:

```
int T1(int n){
  int c=0;
  for (int j=0;j<n;j++)
    for (int i=1;i<j;i++)  c++;
  return c;}
```

Quelques exemples:

Quelle est la complexité de:

```
int T3(int n){
  int c=0;
  for (int i=1;i<n;i=2*i) c++;
  return c;}
```

Quelques exemples:

Quelle est la complexité de:

```
//n entier >1
boolean A(int n){
  if (n<=1) return 1;
  else return A(n-1)+A(n-2);
}
```

Quelques exemples:

Soit l'algorithme suivant pour tester si un nombre est premier:

```
//n entier >1
boolean Premier(int n){
  for (int i = 2; i*i<=n; i++)
    if (n mod i=0) return false;
  return true;
}
```

Cet algorithme est-il polynômial?

Exemple

Non!!

supposons que le coût d'une division soit constant, indépendant de la taille de la donnée, donc en $\Theta(1)$ alors la complexité dans le pire des cas de l'algo est $\Theta(n)$..

mais la taille de la donnée est $\log n$ donc le coût est en $\Theta(2 * |d|)$

A retenir:

- . Quand on parle de la Complexité c'est souvent la complexité temporelle dans le pire des cas
- . Praticable=polynômial
- . Evaluer l'ordre de grandeur de la complexité d'un algorithme ... n'exempte pas de tester et d'expérimenter
- . Attention à la taille de la donnée!