Algorithme et Structures de Données

Wurtz Jean-Marie

Université Louis Pasteur
Wurtz@igbmc.u-strasbg.fr

Plan du cours

- Analyse d'algorithmes
- Complexité/Formalisme
- Récursivité
- Tri
- Arbre
- graphe

Java

Analyse d'un algorithme

- But :
  - estimer son temps d'exécution
  - comparer 2 algorithmes
- Le temps d'exécution peut dépendre :
  - de l'ordinateur
  - de la nature des instructions
  - du compilateur
  - du programmeur
  - du système d'exploitation
  - des données à traiter
  - de la complexité

Références bibliographiques

- "Algorithms in Java"; Robert Sedgewick & Michael Shildlowsky; Third edition, Parts 1-4 : Fundamentals, data structures, sorting & searching; Addison-Wesley.
- "Algorithms in Java"; Third edition, Part 5 : Graph algorithm; Robert Sedgewick & Michael Shildlowsky; Addison-Wesley.
- "Data Structure and Algorithm"; Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman; Addison-Wesley.
- "Algorithmique du texte"; Maxime Crochemore, Christophe Hancart, Thierry Lecroq; Vuibert.

Implémentation d'un algorithme

- Ecriture en Java : mais peut-être implémenté dans n'importe quel autre langage
- Un algorithme fait parti d'un gros programme
  - utilisation de structure de données abstraites
  - peuvent être substituées
- Dès le début : connaître les performances d'un algorithme
- Tout le système en dépend (recherche, tri, …)

Implémentation d'un algorithme (2)

- 2 cas typiques qui nécessitent des algorithmes efficaces :
  - taille du problème énorme
  - code exécuter un grand nombre de fois
- Recherche d'algorithme plus efficace et si possible simple

Analyse empirique d'un algorithme ?
Analyse mathématique d'un algorithme ?

Analyse empirique

- Soit 2 algorithmes réalisant la même tâche
  - compare temps d'exécution sur des entrées typiques
  - 3 s ou 30 s d'attente pour l'utilisateur feront une grande différence
  - permet aussi de valider une analyse mathématique
- Si le temps d'exécution devient trop long, l'analyse mathématique s'impose
  - attente pouvant être de l'ordre de l'heure ou du jour

Temps d'exécution

10^{-6} s pour une instruction

```
int i, j, k, count=0;
   for (i=0; i < N; i++)
      for (j=0; j < N; j++)
            for (k=0; k < N; k++)
                  count++;
```

Temps pour N=	10,	100,	1000,	100000 ou	1 million
	10^{-3}s	1s	1000s	1 an	10 siècles

Analyse empirique (3)

- Nécessite l'implémentation d'une première version d'un algorithme !!!
- Nature des données à tester
  - données typiques
  - données aléatoires
  - données extrêmes
- Attention à la comparaison
  - soin accordé à l'implémentation
  - différences entre ordinateurs, compilateurs, systèmes d'exploitation, ….

Choix d'un algorithme

- Si plusieurs algorithmes sont disponibles ?
- Choix :
  - du plus facile à comprendre, à coder et à debugger
  - du plus rapide et utilisant efficacement les ressources

Choix de l'algorithme (2)

- Algorithme rapide souvent plus compliqué qu'un algorithme "brute force"
  - problèmes de grande taille -> algorithme rapide
- Rien ne sert de multiplié par 10 la performance de l'algorithme si celui-ci
  - ne prend que quelques millisecondes
  - et est très peu utilisé
- Le temps de développement peut-être prohibitif
- un programme non écrit ne peut-être testé!

Outil mathématique

- Mesure générale des performances d'un algorithme
- Comparer 2 algorithmes pour la même tâche
- Prédire la performance dans un nouvel environnement

Analyse mathématique

- L'analyse mathématique d'un algorithme peut-être complexe
- Une spécialité en informatique
- Paramètres hors de portée du programmeur Java :
  - temps d'exécution d'une instruction (MV/compilateur)
  - ressources partagées
  - dépendance des données en entrée
  - certains algorithmes sont tout simplement complexes et il n'y a pas de résultat mathématique pour les décrire

Analyse mathématique (2)

- Néanmoins il est possible de prédire
  - la performance d'un algorithme
  - si un algorithme est meilleur qu'un autre
- Premières étapes:
  - identifier les opérations abstraites de l'algorithmes
  - exemple de la triple boucle, le nombre d'exécutions de l'instruction : count++
  - sans tenir compte du temps en milliseconde
  - cette séparation permet de comparer des algorithmes

Temps d'exécution

10^{-6} s pour une instruction

```
int i, j, k, count=0;
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      for (k=0; k < N; k++)
        count++;
```

Le nombre d'instructions peut varier

Temps pour N=	10,	100,	1000,	100000 ou	1 million
	10^{-3}s	1s	1000s	1 an	10 siècles

Les entrés testées par l'algorithme

- Plusieurs cas sont envisagés:
  - le cas moyen : fiction mathématique
  - le pire des cas : construction qui se produit que rarement
  - donnent néanmoins des informations clés
- Exemple :
  - recherche séquentiel d'un nombre dans un tableau

```
static int search(int a[], int v, int l, int r) {
  int i;
  for (i = l; i <= r; i++)
        if (v == a[i]) return i;
  return -1;
}
```

Les fonctions d'accroissement

- La plupart des algorithmes ont un paramètre N qui les caractérise :
  - le nombre d'assurés sociaux
  - le nombre de fichiers
  - le nombre de caractères
- Une mesure abstraite du problème
- Plusieurs paramètres peuvent exister :
  - par exemple M et N (producteurs et consommateurs, usine de productions et sites de vente)
  - garder un paramètre fixe en faisant varier l'autre paramètre
- But : exprimer les besoins en fonction de N (ou M et N) par des formules mathématiques

Type de fonctions

- 1 instructions exécutées un nombre limité de fois; exécution constante
- log N division du problème pour le résoudre; exécution logarithmique
- N exécution linéaire
- N*log N résolution de pbs plus petits, puis recombinaison des solutions
- N^2 utilisable sur de petits pbs; exécution quadratique
- N^3 triple boucle sur les données
- 2^N exécution exponentielle; N=20 alors 2^N=1 million

Fonctions rencontrées

seconds							
10^2	1.7 minutes						
10^4	2.8 hours						
10^5	1.1 days						
10^6	1.6 weeks						
10^7	3.8 months						
10^8	3.1 years						
10^9	3.1 decades						
10^{10}	3.1 centuries						
10^{11}	*never*						
$\lg N$	$\sqrt{N}$	N	$N \lg N$	$N(\lg N)^2$	$N^{3/2}$	N^2	
3	3	10	33	110	32	100	
7	10	100	664	4414	1000	10000	
10	32	1000	9966	99317	31623	1000000	
13	100	10000	132877	1765633	1000000	100000000	
17	316	100000	1660964	27588016	31622777	10000000000	
20	1000	1000000	19931569	397267426	1000000000	1000000000000	

Temps requis pour résoudre de gros problèmes

operations per second	*problem size 1 million*			*problem size 1 billion*		
	N	$N \lg N$	N^2	N	$N \lg N$	N^2
10^6	seconds	seconds	weeks	hours	hours	never
10^9	instant	instant	hours	seconds	seconds	decades
10^{12}	instant	instant	seconds	instant	instant	weeks

Fonctions spéciales

fonctions	noms	valeurs types	approximation
$\lfloor x \rfloor$	floor function	$\lfloor 3.14 \rfloor = 3$	x
$\lceil x \rceil$	ceiling function	$\lceil 3.14 \rceil = 4$	x
$\lg N$	binary logarithm	$\lg 1024 = 10$	$1.44 \ln N$
F_N	Fibonacci numbers	$F_{10} = 55$	$\phi^N / \sqrt{5}$
H_N	harmonic numbers	$H_{10} \approx 2.9$	$\ln N + \gamma$
$N!$	factorial function	$10! = 3628800$	$(N/e)^N$
$\lg(N!)$		$\lg(100!) \approx 520$	$N \lg N - 1.44N$

$e = 2.71828\ldots$
$\gamma = 0.57721\ldots$
$\phi = (1+\sqrt{5})/2 = 1.61803\ldots$
$\ln 2 = 0.693147\ldots$
$\lg e = 1/\ln 2 = 1.44269\ldots$

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N} \approx \ln N + \gamma + \frac{1}{12N}$$

γ=0,57721, constante d'Euler

Une mesure générale des performances d'un algorithme

La notation O :
Soit deux fonctions f(x) et g(x) avec $x \geq 0$
On dit que f(x)=O(g(x)) si

$\exists \; N_0 \geq 0, \; \exists \; c \rangle 0, \forall \; N \geq N_0 \;\; f(N) \leq c \; . \; g(N)$

A partir d'un certain rang, la fonction « g » majore la fonction « f » à un coefficient multiplicatif près.

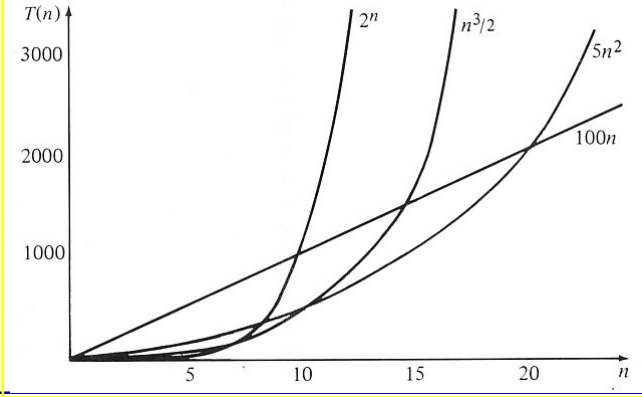
On peut encore écrire : $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$

g est une borne supérieure de f

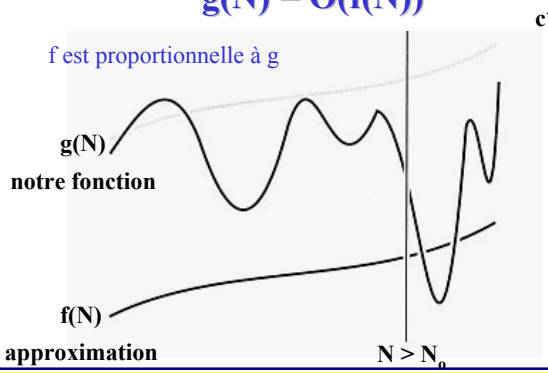
La notation grand-O

- $\exists \; N_0 \geq 0, \; \exists \; c \rangle 0, \forall \; N \geq N_0 \;\; g(N) \leq c \; . \; f(N)$
- Cette notation permet de :
  - limiter l'erreur quand on ignore les termes plus petits dans une formule mathématique
  - limiter l'erreur quand on ignore certaines parties du programme qui prennent peut de temps
  - classer les algorithmes en fonction de leur comportement asymptotique (avec N grand)
- Les constantes N_0 et c_0 cachent des détails d'implémentation
- si $N < N_0$ on ne connaît pas le comportement de l'algorithme
- On ne s'intéresse qu'aux termes les plus grands : comparer N^2 nanosecondes et logN siècles

Temps d'exécution de 4 programmes

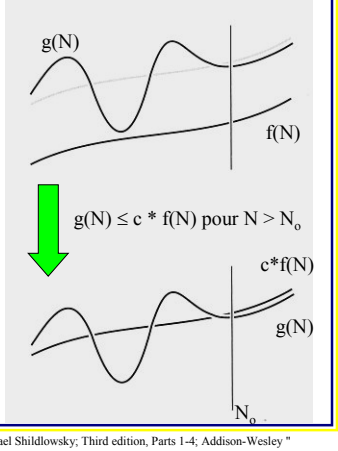


Approximation de « grand-O » : g(N) = O(f(N))



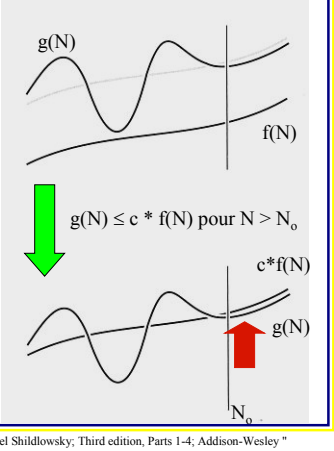
Approximation de la fonction

- g(N) proportionnelle à f(N)
- g(N) croit comme f(N) à une constante "c" près
- Estimation pour N grand



Approximation de la fonction

- g(N) proportionnelle à f(N)
- g(N) croit comme f(N) à une constante "c" près
- Estimation pour N grand



Propriétés de la notation O (1)

- Les constantes peuvent être ignorées :
  $\forall$ k > 0, k*f est O(f)
- Les puissances supérieures croissent plus rapidement :
  si f est un polynôme de degré d alors f est $O(n^d)$
- Les termes croissant rapidement dominent la somme
  si f est O(g), alors f + g est O(g)
  ex: $an^4 + bn^4$ est $O(n^4)$
- Une croissance polynomiale est dictée par la puissance la plus élevée
  n^r est $O(n^s)$ si $0 \le r \le s$

Propriétés de la notation O (2)

- f est O(g) est transitif :
  si f est O(g) et g est O(h) alors f est O(h)
- Le produit des bornes supérieures est la borne supérieure du produit :
  si f est O(g) et h est O(r) alors f*h est O(g*r)
- Les fonctions exponentielles croissent plus vite que les puissances :
  n^k est $O(b^n)$ $\forall$ b > 1 et k ≥ 0
  ex : n^{20} est $O(1.05^n)$
- Les logarithmes croissent plus lentement que les puissances :
  $\log_b n$ est $O(n^k)$ $\forall$ b > 1 et k > 0
  ex: $\log_2 n$ est $O(n^{0.5})$

Propriétés de la notation O (3)

- Tous les logarithmes croissent de manière similaire
  $\log_b n$ est $O(\log_d n)$ $\forall$ *b, d > 1*
- La somme des n premières puissances de r croit comme la puissance de (r+1)

$$\sum_{k=1}^{n} k^r$$ est $O(n^{r+1})$

ex : $$\sum_{k=1}^{r} k = \frac{n(n+1)}{2}$$ is $O(n^2)$

Algorithmes polynomiaux et difficiles

- Algorithme polynomiale
  - s'il est $O(n^d)$ pour un entier "d" donnée
  - les algorithmes polynomiaux sont dits efficaces
    - Ils peuvent être résolus en un temps raisonnable
- Algorithmes difficiles
  - algorithmes pour lesquels il n'y a pas d'algorithme avec un temps polynomiale

Coût des instructions

- Instructions simples : a=b; a+=b; … O(1)
  s_1; s_2; …. ; s_k O(1) si k est une constante
- boucle simple : O(N)
  for(i=1; i <N; i++) inst;
  avec inst O(1) donc N*O(1)=O(N)
- double boucle imbriquée : $O(N^2)$
  for(i=1; i <N; i++)
  for(j=1; j <N; j++) inst;
  complexité N*O(N) donc $O(N^2)$
- triple boucle imbriquée : $O(N^3)$

Exemple avec O(N)

- T(0)=1
- T(1)=4
- $T(N)=(N+1)^2$
- T(N) est $O(N^2)$ avec $N_0=1$ et c=4
- $T(N) \leq 4*N^2$ pour $N \geq N_0=1$
- $N_0=0$ n'est pas possible car T(0)=1 or $c*0^2=0$ pour toutes constantes c

Exercice avec O(N) (1)

- montrer que O(1) est la même chose que O(2) ?
  $\exists \ N_0 \geq 0, \ \exists \ c \ \rangle \ 0, \forall \ N \geq \ N_0 \ \ g(N) \leq \ c \ . \ f \ (N)$
- Cela revient à trouver 2 entiers N_0 et c
  avec : f(N)=1 et la fonction g(N)=2
- $N_0 = 0$ et c = 2 on a bien $2 \leq \ c*1=2$
  pour tout $N \geq N_0 = 0$

Exercice avec O(N) (2)

- Montrer que la fonction $T(n) = 3n^3 + 2n^2$ est $O(n^3)$
- Trouver N_0 et c tel que :
  $\exists \ N_0 \geq 0, \ \exists \ c \ \rangle \ 0, \forall \ n \geq \ N_0 \ \ g \ (n) \leq \ c \ . \ f \ (n)$
- avec $N_0 = 0$ et c = 5
- $3n^3 + 2n^2 \leq \ 5n^3$

Exercice avec O(N) (3)

- montrer que 3^N n'est pas $O(2^N)$
- Supposons qu'il existe 2 constantes c et N_0 tel que $N \geq N_0$ pour lesquelles l'inégalité suivante est vérifiée : $3^N \leq c*2^N$
- alors $c \geq (3/2)^N$, c peut devenir arbitrairement grand pour N grand, donc il n'existe pas de constante supérieure $(3/2)^N$ quelque soit N

Exercice avec O(N) (4.1)

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N} \approx \ln N + \gamma + \frac{1}{12N}$$

- Algorithme :
  - Initialisation exécutée 1 fois (temps : a1 ns)
  - une boucle interne parcourue $2N*H_N$ (temps : a2 ns; avec H_N le nombre harmonique)
  - une section supplémentaire exécutée N fois (temps : a3 ns)
- Temps moyen d'exécution de l'algorithme?
- $a_1 + 2a_2N*H_N + a_3*N$ ns

Exercice avec O(N) (4.2)

$$H_N = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{N} \approx \ln N + \gamma + \frac{1}{12N}$$

- Avec la notation O(N) que devient cette estimation ?
  $a_1 + 2a_2N*H_N + a_3*N$ ns
- $2a_2N*H_N + O(N)$
- Une forme plus simple qui indique que ce n'est pas la peine de chercher a_1 et a_3
- Temps d'exécution exprimé en notation « grand-O » ?
- $H_N=\ln(N) + O(1)$ on obtient ainsi $2a_2N*Ln(N)+ O(N)$
- l'expression asymptotique du temps total d'exécution : temps proche de $2a_2N*Ln(N)$ pour des N grands

Exercice avec O(N) (4.3)

- l'expression asymptotique du temps total d'exécution : temps proche de $2a_2N*Ln(N)$ pour des N grands
- que se passe-t-il si N double : taille 2N?

$$\frac{2a_2(2N)\ln(2N) + O(2N)}{2a_2N\ln N + O(N)} = \frac{2\ln(2N) + O(1)}{\ln N + O(1)}$$

$$= 2 + O\left(\frac{1}{\ln N}\right)$$

$2a_2N$ en facteur

- sans connaître a_2 on peut prédire que le temps doublera pour le traitement des données de taille 2N

Application : Recherche du maximum

- Recherche séquentielle :
-


```
static int search(int a[], int v, int l, int r)
{
      int i;
      for (i = 1; i <= r; i++)
            if (v == v[i]) return i;
      return -1;
}
```

Analyse de l'algorithme?

paramètre clé?

Recherche séquentielle : coût?

-
```
static int search(int a[], int v, int l, int r)
{
      int i;
      for (i = 1; i <= r; i++)
            if (v == v[i]) return i;
      return -1;
}
```

- Coût moyen
  (1 + 2 + 3 + 4 +…+ N) / N = N (N + 1) / 2N = (N + 1)/2
- Coût extrême
  N car on parcourt tout le tableau

Accélération de la recherche

- Comment?
- Trier les nombres (N nombres) : O(NlogN)
  Coût faible comparé au nombre de comparaisons si on fait une recherche très souvent (M : nombre de recherche; M >>N)
- La recherche séquentielle : M*N
- La recherche binaire : NLogN + M*?

Recherche binaire

```
static int search(int a[], int v, int l, int r)
{
   while(r>=l) {
         int m = (l+r)/2;
         if (v == a[m]) return m;
         if (v  <  a[m]) r = m-1; else l = m+1;
   }
   return -1;
}
```

Propriété : la recherche binaire examine au plus $\lfloor \lg N \rfloor + 1$

Recherche binaire

```
1488 1488
1578 1578
1973 1973
3665 3665
4426 4426
4548 4548
5435 5435 5435 5435 5435
5446 5446 5446 5446
6333 6333 6333
6385 6385 6385
6455 6455 6455
6504
6937
6965
7104
7230
8340
8958
9208
9364
9550
9645
9686
```

11 nombres

23 nombres

1. Recherche de 5025
2. Comparaison à 6504; 1ère moitié
3. Comparaison à 4548; 2ème moitié
4. ….

Recherche binaire (2)

```
static int search(int a[], int v, int l, int r)
{
   while(r>=l) {
        int m  =  (l+r)/2;
        if (v == a[m]) return m;
        if (v  <  a[m]) r = m-1; else l = m+1;
   }
   return -1;
}
```

Coût ?

recherches infructueuses

recherches avec succès

Recherche binaire (3)

```
static int search(int a[], int v, int l, int r)
{
   while(r>=l) {
        int m = (l+r)/2;
        if (v==a[m]) return m;
        if (v < a[m]) r=m-1; else l=m+1;
   }
   return -1;
}
```

2 situations :

recherches infructueuses

recherches avec succès

$T_N \leq T_{\lfloor N/2 \rfloor} + 1$, pour $N \geq 2$ avec $T_1 = 1$

$N=2^n$; $T_N \leq n+1 = \lfloor \lg N \rfloor + 1$

$N=10^9$ recherche en au plus 30 comparaisons

Etude empirique de la recherche séquentielle et recherche binaire

- Résultat : temps relatif
- M recherches : assurés sociaux
- S : proportionnelle à M*N
- B : proportionnelle à M*lgN
- S impossible pour N grand

	M = 1000		M = 10000		M = 100000	
N	S	B	S	B	S	B
125	3	3	36	12	357	126
250	6	2	63	13	636	130
500	13	1	119	14	1196	137
1250	28	1	286	15	2880	146
2500	57	1	570	16		154
5000	113	1	1172	17		164
12500	308	2	3073	17		173
25000	612	1		19		183
50000	1217	2		20		196
100000	2682	2		21		209

Autres notation

$\Omega(g)$: l'ensemble des fonctions dont la croissance est similaire à g

g est une borne inférieure

$\exists \; N_0 \geq 0, \; \exists \; c \rangle 0, \forall \; N \geq \; N_0 \quad f(N) \; \geq \; c*g(N)$