Algorithmique et structures de données

Chapitre 12 – Complexité d'un algorithme

Samar MOUCHAWRAB, PhD Eng

2A Cycle Préparatoire – Semestre 1
2017/2018

Introduction

Parmi les objectifs d'écrire des algorithmes est d'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?
Comment modéliser le problème?
Est-ce un problème classique?
Comment concevoir un algorithme?
L'algorithme est-il correct?
L'algorithme est-il efficace?
Peut-on trouver un algorithme plus efficace pour le problème?

Introduction

Lors de l'étude d'un problème, nous pouvons être amenés à estimer et comparer les temps d'exécution requis par différentes méthodes.

Avant d'entrer directement dans une phase de développement et de test, nous pouvons déjà en amont chercher à évaluer de manière théorique la quantité d'appels aux opérations les plus coûteuses en temps.

Nous pouvons alors estimer le temps d'exécution suivant deux optiques distinctes. Il s'agit de l'étude en moyenne et de l'étude dans le pire cas.

L'approche en moyenne consiste à évaluer la durée moyenne d'exécution d'un algorithme donné. Elle se base sur des modèles probabilistes complexes. L'approche dans le pire cas consiste à estimer la durée maximale d'exécution d'une méthode donnée. Ce type d'étude est beaucoup plus simple.

Exemples

Problème 1 : Placement des invités

Un traiteur organise un colloque. La seule hôtesse d'accueil présente ne dispose pas de la liste des invités. Elle n'a ni feuille ni crayon. Les cartons d'invitation ont été disposés aux places respectives des invités. Malheureusement les noms ont été écrits dans une police trop petite et l'hôtesse doit donc prendre à chaque fois quelques secondes pour les lire.

Sachant qu'il y a **n** invités et **m** tables. Exprimez dans le pire cas le nombre de fois où l'hôtesse devra examiner les cartons. Nous négligerons les parcours entre les chaises et nous supposerons que l'hôtesse est tellement stressée qu'elle ne parvient pas à retenir l'emplacement des noms précédemment lus.

Réponse : $n(n-1)/2$

Exemples

Problème 2 : les boules de bowling

J'ai n boules de bowling numérotées de 1 à n disposées sur des emplacements portant leurs numéros. A la fin de la journée, les clients ont remis les boules complètement dans le désordre. Combien dois-je effectuer de déplacement dans le pire cas pour que les boules soient toutes à leur place ?

Réponse : n

Méthode d'évaluation

Prenons une fonction calculant le minimum des valeurs présentes dans un tableau de n éléments :

```
int min(int *T, int n) {
      int m=T[0];
      for (int i=1 ; i < n ; i++)
            if ( T[i] < m )
                m=T[i];
      return m;
}
```

Le temps d'exécution est fonction du nombre de lecture en mémoire et du nombre de comparaisons effectuées. Dans le pire cas, nous effectuons n lectures et n - 1 comparaisons. La durée nécessaire à l'appel de cette fonction est donc proportionnelle aux nombres de valeurs présentes. La constante de proportionnalité dépendra du coût en temps d'une lecture en mémoire et d'une comparaison. Ces paramètres dépendent entièrement de l'architecture utilisée.

Méthode d'évaluation

Dans l'étude du pire cas, nous pouvons écrire :

Appel séquentiel

```
f(...) {
      g(...);
      h(...);
}
```

Coût (ƒ (...))=Coût (g (...))+Coût (h(...))

Appel conditionnel

```
f(...) {
      if (...)
            g(...)
      else
            h(...);
}
```

Coût (ƒ (...))=max (Coût (g (...)) , Coût (h(...)))

Méthode d'évaluation

Attention aux appels itératifs

```
f(..., int n) {
        for (i = 1 ; i <= n ; i++)
                g(...);
}
```

Nous aurons Coût (ƒ (... , n))= n·Coût (g (...)) .

Cependant ce n'est pas toujours le cas, c'est-à-dire que lorsque l'appel à la fonction g() est dépendant de l'indice de boucle :

```
f(..., int n) {
        for (i = 1 ; i < n ; i++)
                g(...,i);
}
```

Nous devons réécrire le coût en comptabilisant les différents appels, ainsi nous avons :

Coût (ƒ (... , n))= $\sum_{i=1}^{n}$ Coût (g (... , i))

Notation grand O et classe de complexité

Principe général:

Supposons qu'une méthode effectue C (n)=3 n^2+6 n+18 opérations coûteuses dans le pire cas. Lorsque C (n) est grand, cette valeur devient équivalente à celle du monôme de plus haut degré.

Ainsi C (n) ~ 3 ·n2 .

Si nous travaillons sur un processeur pouvant effectuer 10^9 de ces opérations par seconde, nous obtenons les résultats suivants :

n	ƒ (n)=n	ƒ (n)=n^2	ƒ (n)=n^4
10^1	~ 0	~0	~0
10^3	~ 0	~0	∞
10^6	~ 0	∞	∞
10^9	~ 1 s	∞	∞
10^{12}	∞	∞	∞

Notation grand O et classe de complexité

Nous remarquons que dans certains cas, l'algorithme est réalisable dans un temps quasi- immédiat (inférieur à une seconde : ~0). Dans d'autres cas, la durée dépasse largement l'heure, la méthode peut être considérée comme trop lente.

Uniquement dans le cas ƒ (n)=n lorsque la quantité d'opérations à effectuer est de l'ordre de la puissance du processeur, nous nous trouvons dans une tranche où la durée du programme s'avère acceptable (ni immédiat, ni trop long).

Nous sommes amenés à penser qu'il existe une zone de cassure à partir de laquelle une méthode ne peut plus être utilisée. Ainsi lorsque ƒ (n)=n nous pouvons traiter de l'ordre de 10^9 données, seulement 10^4 lorsque ƒ (n)=n^2 et uniquement 179 lorsque ƒ (n)=n^4 . Au final, l'importance du coefficient multiplicateur est relative.

Classes de complexité

Quand nous calculerons la complexité d'un algorithme, nous ne calculerons généralement pas sa complexité exacte, mais **son ordre de grandeur**. Pour ce faire, nous avons besoin de notations asymptotiques. La plus utilisée est la notation grand O.

Les classes de complexité les plus usuelles sont :

O (n) linéaire
O (n^2) quadratique
O (n^3) cubique
O (log n) logarithmique
O (c^n) exponentielle

Classes de complexité

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :

– Les algorithmes sub-linéaires dont la complexité est en général en O(log n).

– Les algorithmes linéaires en complexité O(n) et ceux en complexité en O(n log n) sont considérés comme rapides.

– Les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ sont considérés comme lents, sans parler des algorithmes exponentiels (dont la complexité est supérieure à tout polynôme en n) que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

Illustration : cas du tri par insertion

Problématique du tri

Entrée : une séquence de n nombres, $a_1, ..., a_n$.

Sortie : une permutation, $a'_1, ..., a'_n$, de la séquence d'entrée, telle que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Principe du tri par insertion

De manière répétée, on retire un nombre de la séquence d'entrée et on l'insère à la bonne place dans la séquence des nombres déjà triés (ce principe est le même que celui utilisé pour trier une poignée de cartes).

```
void tri_insertion(int tab[], int taille){
    int i, j, elem;
    for (i = 1; i < taille; ++i) {
        elem = tab[i];
        for (j = i; j > 0 && tab[j-1] > elem; j--)
            tab[j] = tab[j-1];
        tab[j] = elem;
    }
}
```

Illustration : cas du tri par insertion

```
void tri_insertion(int tab[], int taille){          Coût
   int i, j, elem;                                  soit taille = n)
   for (i = 1; i < taille; ++i) {                   n - 1
       elem = tab[i];                               n - 1
       for (j = i; j > 0 && tab[j-1] > elem; j--)   n(n-1)/2
           tab[j] = tab[j-1];                       n(n-1)/2
       tab[j] = elem;                               n - 1
   }
}
```

Dans le pire cas, la complexité est **fonction quadratique** de *n*.

Dans le meilleur cas, la liste est déjà triée, la complexité est linéaire car il faut parcourir tout le tableau une seule fois pour s'assurer qu'il n'y a pas d'échanges à faire.

meilleur cas : $O(n)$.

pire cas : $O(n^2)$.

En général, on considère qu'un algorithme est plus efficace qu'un autre si sa complexité dans le pire cas a un ordre de grandeur inférieur.