Structures De Données Et Complexité

PROMOTION: MASTER SMART'COM M1, ISET'COM

ENSEIGNANT: SLAH BOUHARI

Références et Ressources

- Complexité et algorithmique avancée - Une introduction, Ivan Lavallée, Hermann,
- Data Structures and Algorithms, Granville Barnett, and Luca Del Tongo,
- Notes de cours ’Algorithmique et complexité’, Med Becha Kaaniche, Supcom, 2015
- Notes de cours ’Algorithmique et complexité’, K. Hamrouni, ENIT
- Notes de cours ‘Algorithmique et complexité’, Slim Mesfar

Plan

Complexité et optimalité

Algorithmes de Tri

La récursivité et le paradigme « diviser pour régner »

Structures de données avancées

NP-complétude

Les Heuristiques

Notions de base

Algorithmes:

P : un problème

M: une méthode pour résoudre le problème P

Algorithme : description de la méthode M dans un *langage algorithmique*

Un algorithme est séquence d'étapes de calcul qui transforment l'entrée en sortie.

Notions de base(suite)

Structures algorithmiques:

Structures de contrôle

- séquence
- embranchement (ou sélection)
- boucle (ou itération)

Structures de données

- constantes
- variables
- tableaux
- structures récursives (listes, arbres, graphes)

Notions de base(suite)

Complexité des algorithmes

On veut:

Evaluer l'efficacité de la méthode **M**

Comparer **M** avec une autre méthode **M'**

indépendamment de l'environnement (machine, système, compilateur, . . .)

La complexité d'un algorithme est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée.

Un algorithme est plus efficace qu'un autre si son temps d'exécution du cas le plus défavorable a un ordre de grandeur inférieur.

Notions de base(suite)

Complexité des algorithmes

Evaluation du nombre d'opérations élémentaires en fonction

- de la taille des données,
- de la nature des données.

Notations :

- n : taille des données,
- T(n) : nombre d'opérations élémentaires

Configurations caractéristiques

- meilleur cas,
- pire des cas,
- cas moyen.

Evaluation des coûts

Evaluation de : T(n) **(séquence):**

Somme des coûts

$$\left.\begin{array}{ll}\text{Traitement1} & T_1(n) \\ & \\ \text{Traitement2} & T_2(n)\end{array}\right\} \qquad T(n) = T_1(n) + T_2(n)$$

Evaluation des coûts (suite)

Evaluation de T(n) **(embranchement):**

Max des coûts.

$$\left.\begin{array}{lll} \textbf{si} & < \text{condition} > \ \textbf{alors} & \\ & \text{Traitement1} & T_1(n) \\ \textbf{sinon} & & \\ & \text{Traitement2} & T_2(n) \end{array}\right\} max(T_1(n), T_2(n))$$

Evaluation des coûts (suite)

Evaluation de T(n) **(boucle):**

Somme des coûts des passages successifs

$$\left.\begin{array}{l} \text{tant que } < \text{condition} > \text{ faire} \\ \qquad \text{Traitement} \qquad T_i(n) \\ \text{fin faire} \end{array}\right\} \quad \sum_{i=1}^{k} T_i(n)$$

$T_i(n)$: coût de la $i^{\text{ème}}$ itération

Evaluation des coûts (suite)

Evaluation de T(n) **(fonctions récursives):**

fonction FunctionRecursive (n)

1 **si** $(n > 1)$ **alors**
2 FunctionRecursive($n/2$), coût $T(n/2)$
3 Traitement(n), coût $C(n)$
4 FunctionRecursive($n/2$), coût $T(n/2)$

$$T(n) = 2 * T(n/2) + C(n)$$

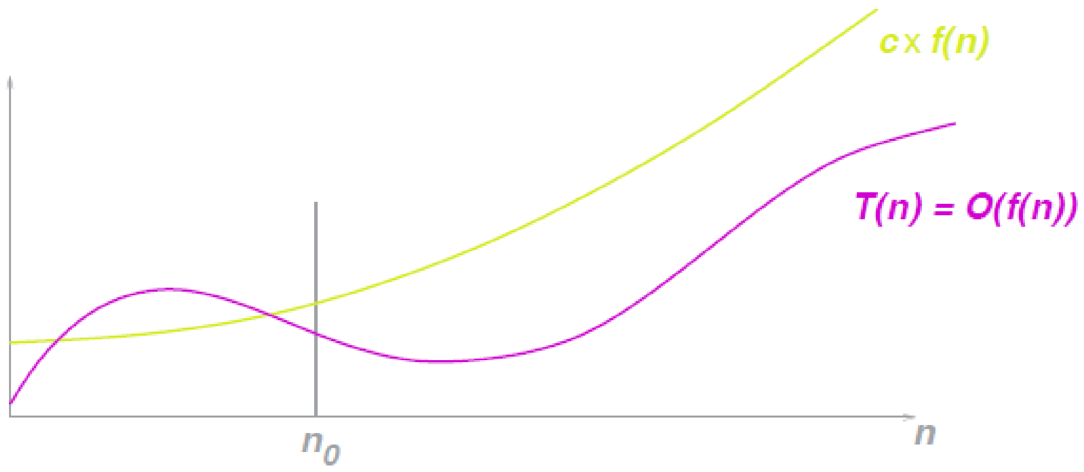
si $C(n) = 1$ alors $T(n) = K \times n$

si $C(n) = n$ alors $T(n) = K \times n \times \log n$

Notation de Landau $O(f(n))$

Caractérise le comportement asymptotique (i.e. quand n → ∞).

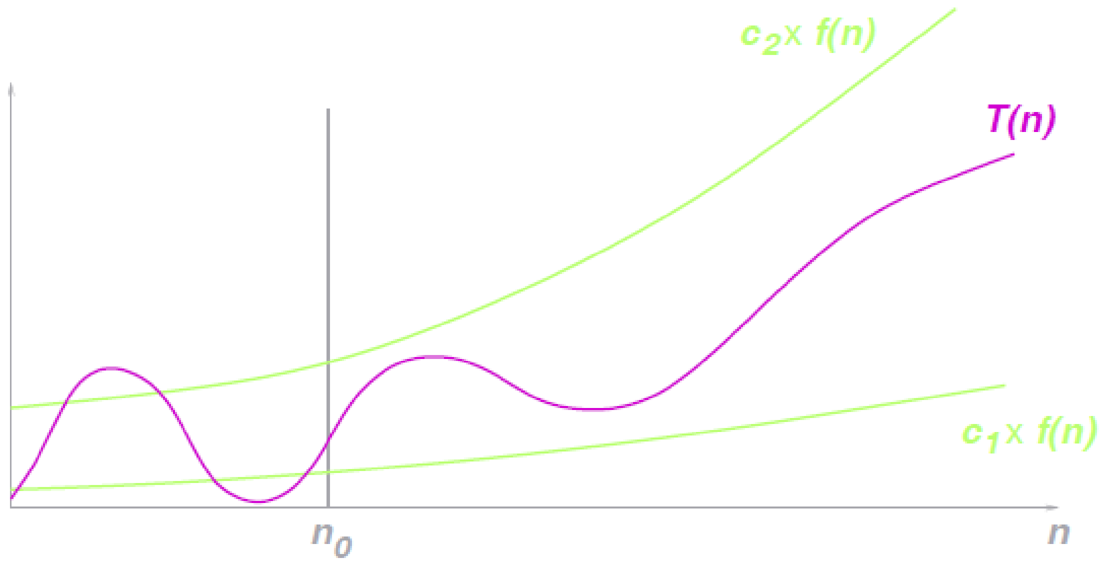
Domination asymptotique



$$T(n) = O(f(n)) \text{ si } \exists c\ \exists n_0 \text{ tels que } \forall n > n_0,\ T(n) \leq c \times f(n)$$

Notation $\Theta(f(n))$

Equivalence asymptotique



$T(n) = \Theta(f(n))$ si $\exists c_1, c_2, n_0$ tels que

$$\forall n > n_0, \;\; c_1 \times f(n) \leq T(n) \leq c_2 \times f(n)$$

Exemple

$$f(n) = n^3 + 2\,n^2 + 4\,n + 2 = O(n^3)$$

(si $n \geq 1$ alors $f(n) \leq 8 \times n^3$)

$$f(n) = n \log n + 12\,n + 888 = O(n \log n)$$

$$f(n) = 1000\,n^{10} - n^7 + 12\,n^4 + \frac{2^n}{1000} = O(2^n)$$

Les principales classes de complexité

O(1) : complexité constante, pas d'augmentation du temps d'exécution quand le paramètre croit

O(log(n)) : complexité logarithmique, augmentation très faible du temps d'exécution quand le paramètre croit. *Exemple : algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (dichotomie).*

O(n) : complexité linéaire, augmentation linéaire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double). *Exemple : algorithmes qui parcourent séquentiellement des structures linéaires.*

O(nlog(n)) : complexité quasi-linéaire, augmentation un peu supérieure a O(n). Exemple : algorithmes qui décomposent un problème en d'autres plus simples, traites indépendamment et qui combinent les solutions partielles pour calculer la solution générale.

Les principales classes de complexité

$O(n^2)$: complexité quadratique, quand le paramètre double, le temps d'exécution est multiplie par 4. *Exemple : algorithmes avec deux boucles imbriquées.*

$O(n^i)$: complexité polynomiale, quand le paramètre double, le temps d'exécution est multiplie par 2^i. *Exemple : algorithme utilisant i boucles imbriquées.*

$O(i^n)$: complexité exponentielle, quand le paramètre double, le temps d'exécution est élevé a la puissance 2.

$O(n!)$: complexité factorielle, asymptotiquement équivalente a n^n

Les algorithmes de trie et complexité

Tri bulle

Tri insertion

Tri pivot

Tri fusion

Tri par arbre

Trie bulle

```
PROCEDURE tri_bulle ( TABLEAU T[1:n])
i,j : entiers
    pour i de 2 à n  faire
        pour j de 1 à n-1  faire
            SI T[j] > T[j+1] alors
                permuter T[i] et T[j+1]
            FIN SI
      FIN POUR
  FIN POUR
```

Complexité

$1^{ère}$ itération : (n-1) comparaisons

$2^{ème}$ itération : (n-2) comparaisons

…

$T(n) = (n-1) + (n-2) + ... +1 = n * (n-1) /2$

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

Trie bulle (2)

Une amélioration possible du trie bulle consiste à utiliser une variable booléenne drapeau qui permet de stopper le tri si aucune permutation n’a lieu.

```
PROCEDURE tri_bulle ( TABLEAU T[1:n])
   i,j : entiers
   b : boolean
  debut
  b<- faux
  i <- n-2
   tant que (non b) et ( i > 0)  faire
      b<- vrai
       pour j de 1 à i  faire
              SI T[j] > T[j+1] alors
                 permuter T[i] et T[j+1]
                b<- faux
              FIN SI
       FIN POUR
         i <- i-1
      FIN tant que
Fin
```

La complexité reste en moyenne et au pire en $O(n^2)$

Tri insertion

```
PROCEDURE tri_insertion ( TABLEAU T[1:n])
i,j, val : entiers
Debut
    pour i de 1 à n-1  faire
        val <- T[i]
        j <- i
            Tant que (j>0) et  (T[j-1] > val) alors
                T[j] <- T[j-1]
                j <- j-1
             FIN Tant que
      T[j] <- val
     FIN POUR
  FIN
```

Complexité

Au pire: si le tableau est trié en ordre inverse

A chaque itération de la boucle pour on fait i tours de la boucle tant que (comparaison)

$T(n) = 1 + 2 + … + (n-1) = n * (n-1) /2$

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

Au mieux : si le tableau est déjà trié

Une seule comparaison par élément sera réalisé (sans permutation) sauf pour le premier élément

$T(n) = n-1$: $T(n) = O(n)$ complexité linéaire.

En moyenne : chaque élément sera inséré au milieu de ceux déjà triés

$T(n) = 0 + 1 + 3/2 … + n/2 = n * (n+1) /4$

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

Tri rapide (tri pivot)

```
Fonction Partition (TABLEAU T[1:n], a, b :entier) : entier
Pivot, i, temp : entier
Debut
Pivot <- a
Pour i de a+1 à b faire
  si T[i] < T[pivot] alors
      temp <- T[i]
      T[i] <- T[pivot + 1]
      T[pivot+1] <- T[pivot]
      T[pivot] <- temp
      pivot <- pivot +1
FinSi
FinPour
Partition <- pivot
Fin
```

```
PROCEDURE tri_Rapide ( TABLEAU T[1:n], deb, fin :entier)
 pivot : entier
 début
  SI (fin>deb) alors
   pivot <- Partition (T, deb, fin)
   tri_Rapide (T, deb, pivot -1)
    tri_Rapide (T, pivot +1, fin)
 FinSi
Fin
```

Complexité

La fonction de partitionnement est linéaire de la forme a*n (a est une constante)

Au mieux: dans le cas ou le partitionnement coupe le tableau en deux parties égales (à 1 près) , la complexité de la fonction Tri est telle que:

$T(n) = a*n + 2 * T(n/2) + b$ (b est une constante)

Donc $T(n) = O(n*\log(n))$

Au pire: dans le cas ou le partitionnement conduit systématiquement à ce qu'une des partie soit vide. la complexité de la fonction Tri est telle que:

$T(n) = a*n + c(n-1)+ b$ (b est une constante)

Donc $T(n) = O(n^2)$

Tri fusion

Cette procédure fusionne les deux parties de T d'indice dans [a, b] et [b+1, c] en supposant que les éléments de ces parties sont triés en ordre croissant.

PROCEDURE Fusion (TABLEAU T[1:n], a, b, c :entier)

i, j, k : entier

T1 : tableau[1: (b-a+1)], T2 : tableau [1: (c-b)]

Debut

Pour i de 1 à « taille T1 » faire

T1[i] <- T[a+i]

FinPour

Pour j de 1 à « taille T2 » faire

T2[j] <- T[b+1+j]

FinPour

i<- 1 j<- 1 k<-a

Tantque (k <= c) faire

Si (i > = « taille T1 »)alors

T[k] <- T2[j] j<- j+1

Sinon

Si (j > = « taille T2 ») alors

T[k] <- T1[i] i<- i+1

Sinon

si (T1[i] <= T2[j]) alors

T[k] <- T1[i] i<- i+1

sinon

T[k] <- T2[j] j<- j+1

FinSi ***FinSi*** **FinSi**

K<-k+1

FinTantque Fin

Tri fusion

```
PROCEDURE tri_Fusion ( TABLEAU T[1:n], deb, fin :entier)
début
  SI (fin>deb) alors
    tri_Fusion ( T, deb, (deb+fin)/2 )
    tri_Rapide (T, (deb+fin)/2 +1, fin)
    Fusion (T, deb, (deb+fin)/2 , fin)
 FinSi
Fin
```

Complexité

La complexité de la fusion est linéaire de la forme a*n (a est une constante)

la complexité de la fonction Tri est telle que:

$T(n) = a*n + T(n/2) + T(n/2) + b$ (b est une constante)

Donc $T(n) = O(n*\log(n))$ dans tous les cas