Matière: Algorithmique Avancée et Langage C

Complexité temporelle

récursivité et récurrence

Enseignant(e)s: **Feyrouz HAMDAOUI & Sofiane Ben Ahmed**

Niveau: **3ème Génie Informatique**

2024/2025 SU2

Plan

- Complexité itérative
- Complexité d'algorithmes récursifs
- Résolution des récurrences
- Théorème

Comment calculer la complexité en pratique ?

- Quelques règles pour les algorithmes itératifs :

➢ Affectation, accès à un tableau, opérations arithmétiques, appel de fonction : O(1)

➢ Instruction If-Then-Else : O(complexité max des deux branches)

➢ Séquence d'opérations : l'opération la plus couteuse domine (règle de la somme)

➢ Boucle simple : O(n f(n)) si le corps de la boucle est O(f(n))

Comment calculer la complexité en pratique ?

- Double boucle : $O(n^2 f(n))$ où $f(n)$ est la complexité du corps de la boucle
- Boucles incrémentales : $O(n^2)$ (si corps $O(1)$)

```
for i = 1 to n
  for j = 1 to i
      . . .
```

- Boucles avec un incrément exponentiel :$O(\log n)$ (si corps $O(1)$)

```
i = 1
while i ≤ n
   . . .
   i = 2i
```

Plan

- Complexité itérative
- Complexité d'algorithmes récursifs
- Résolution des récurrences
- Théorème

Complexité d’algorithmes récursifs

- La complexité d’algorithme récursif mène généralement à une équation de récurrence
- Il existe diverses techniques pour la résolution des équations de récurrence (méthode des fonctions génératrices et décomposition des fractions rationnelles, transformée en Z, …). Mais la résolution de cette équation n’est généralement pas triviale
- On se contentera d’étudier quelques cas particuliers importants dans ce cours

Récurrence

- ❖ **Suite récurrente:** la définition d'une suite est la donnée
- ➢ d'un terme général défini en fonction du (ou des) terme(s) précédant(s)
- ➢ D'un terme initial qui permet d'initialiser le calcul
- ❖ **Principe de récurrence :**
- Soit P un prédicat (ou propriété) sur IN qui peut être soit vrai soit faux (on écrira souvent P(n) a la place de P(n) = vrai).
- On suppose que
  - ➢ P(0) vrai
  - ➢ $\forall n \in IN, P(n) \Rightarrow P(n+1)$
- Alors , pour tout $n \in IN$, P(n) est vrai.
- Si on considère le prédicat suivant
- ➢ P(n) : je sais résoudre le problème pour n
- ➢ alors le principe de récurrence nous dit que si je sais résoudre le Pb pour n=0
- ➢ et que si je sais exprimer la solution pour n en fonction de la solution pour n+1
- ➢ alors je sais résoudre le Pb pour n'importe quel n.

Calcul du temps d'exécution d'une procédure récursive

- Considérons une procédure récursive et analysons son temps d'exécution Tp(n). L'analyse de la procédure récursive passe par l'étude de deux cas :

➢ La taille des arguments est suffisamment petite pour qu'aucun appel récursif ne soit effectué. Ce cas correspond à *<u>l'étude de la base de récursivité</u>*.

➢ La taille des arguments est grande pour que des appels récursifs puissent paraître. Cependant on suppose que tout appel récursif effectué par I vers lui même ou autre procédure Q sera effectué avec des arguments plus petits. Ce cas correspond à *<u>l'étape de récursivité de la définition de Tp(n)</u>*.

Exemple (1)

```
    Fonction fact (n :entier) :entier
    Début
1)  Si (n ≤ 1) alors
2)      Fact ← 1
    Sinon
3)      Fact ← n*Fact(n-1)
    Fin si
    Fin
```

- Soit T(n) le temps d'exécution de la procédure Fact, n étant la taille de l'argument. Il est clair que les appels récursifs se font avec un argument de taille plus petit.

Exemple (2)

- ***<u>Etude de la base de la récursivité :</u>***

  La base correspond à n = 1.

  A n = 1, uniquement les instructions 1) et 2) sont exécutées. Le temps d'exécution de Fact pour le cas n = 1 est alors un temps constant, c'est-à-dire O(1). T(1) = O(1).

- ***<u>Cas n > 1 :</u>*** La condition de la ligne 1) n'est pas vérifiée, les instructions 1) et 3) sont exécutées. L'instruction de 1) se fait en temps constant O(1) et la ligne 3) se fait en temps T(n-1).

  → Finalement, nous avons la relation de récurrence suivante qui décrit le temps d'exécution de Fact :

$$T(n) = \begin{cases} O(1) \text{ si } n = 1 \\ O(1) + T(n-1) \text{ sinon} \end{cases}$$

- Il faut résoudre cette récurrence pour déterminer le terme général de T(n).

Exemple (3)

MÉTHODE DES SUBSTITUTIONS

- **<u>Etape 1 :</u>** Commencer par introduire des termes constants pour remplacer les O(1).

$$T(n) = \begin{cases} a \text{ si } n = 1 \\ b + T(n-1) \text{ sinon} \end{cases}$$

- **<u>Etape 2 :</u> Conjoncture sur le résultat**

  Essayons de déterminer le résultat d'une manière non formelle. Pour cela on peut résoudre la série d'équations.

$$\begin{aligned} T(n) &= b + T(n-1) \\ T(n-1) &= b + T(n-2) \\ T(n-2) &= b + T(n-3) \\ &\dots \\ T(2) &= b + T(1) \end{aligned}$$

  En remplaçant les termes T(i) un par un dans les équations, nous obtenons finalement

$$T(n) = (n-1)b + a$$

Exemple (4)

- **<u>Etape 3 :</u> Prouver le résultat par récurrence**

  Il faut maintenant prouver le résultat par récurrence.

1. Pour n = 1, T(n) = a, ce qui est vrai
2. Supposons que le résultat est vrai jusqu'à l'ordre n – 1, alors :

$$\begin{aligned} T(n) &= b + T(n-1) \\ &= b + (n-2)b + a \\ &= (n-1)b + a \end{aligned}$$

d'où le résultat a et b sont des constantes, il faut les cacher :

$$T(n) = (n-1)\,b + a = nb + (b-a).$$

Exemple (5)

- T(n) est un polynôme en n donc T(n) = O(n), ce qui est logique.
- Le coût du calcul de la factorielle c'est n fois le coût d'appel de la fonction elle même qui se fait en temps constant.
- La complexité de la factorielle récursive est donc linéaire, comme celle de la factorielle itérative.
- A l'exécution, la fonction récursive est un peu moins rapide (pente de la droite plus forte) du fait des appels récursifs.

Plan

- Complexité itérative
- Complexité d'algorithmes récursifs
- Résolution des récurrences
- Théorème

Résolution des récurrences (1)

- Equation : **$c(n) = c(n-1) + b$**

  Solution : $c(n) = c(0) + b*n =$ **$O(n)$**

  Exemples : factorielle, recherche séquentielle récursive dans un tableau

- Equation : **$c(n) = a*c(n-1) + b$, $a \neq 1$**

  Solution : $c(n) = a^n*(c(0) – b/(1-a)) + b/(1-a) =$ **$O(a^n)$**

  Exemples : répétition a fois d'un traitement sur le résultat de l'appel récursif

- Equation : **$c(n) = c(n-1) + a*n + b$**

  Solution : $c(n) = c(0) + a*n*(n+1)/2 + n*b =$ **$O(n^2)$**.

  Exemples : traitement en coût linéaire avant l'appel récursif, tri à bulle

Résolution des récurrences (1)

- <u>Equation :</u> **c(n) = c(n/2) + b**

  <u>Solution :</u> c(n) = c(1) + b*log2(n) = **O(log(n))**

  <u>Exemples :</u> élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive

- <u>Equation :</u> **c(n) = a*c(n/2) + b, a ≠ *1***

  <u>Solution :</u> c(n) = $n^{\log 2(a)}$ *(c(1) – b/(1-a)) + b/(1-a) = **O($n^{\log 2(a)}$)**

  <u>Exemples :</u> répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique

Résolution des récurrences (2)

- Equation : **c(n) = c(n/2) + a*n +b**

  Solution : c(n) = **O(n)**

  Exemples : traitement linéaire avant l'appel récursif dichotomique

- Equation : **c(n) = 2*c(n/2) + a*n + b**

  Solution : c(n) = **O(n*log(n))**

  Exemples : traitement linéaire avant double appel récursif dichotomique, tri fusion

Plan

- Complexité itérative
- Complexité d'algorithmes récursifs
- Résolution des récurrences
- Théorème

Théorème

1. Soit T(n) une fonction définie par l'équation de récurrence suivante, où $k \geq 0$, $b > 0$, et $c \geq 1$:

$$\boxed{T(n) = cT(n-1) + b*n^k}$$

- Si c=1 alors $T(n) = O(n^{k+1})$
- Si c>1 alors $T(n) = O(c^n)$

2. Soit T(n) une fonction définie par l'équation de récurrence suivante, où $d \geq 2$, $k \geq 0$, $c>0$, $b > 0$:

$$\boxed{T(n) = cT(n/d) + b*n^k}$$

La relation entre c, d et k détermine la fonction T(n) comme suit :

$$\boxed{\begin{array}{l} \text{Si } c > d^k, \text{ alors } T(n) = O(n^{\log_d(c)}) \\ \text{Si } c = d^k, \text{ alors } T(n) = O(n^k * \log(n)) \\ \text{Si } c < d^k, \text{ alors } T(n) = O(n^k) \end{array}}$$

Bibliographie

- Pierre Geurts. «Partie 2 Outils d'analyse ». Matière « Introduction à la théorie de l'informatique », 2ème Bachelier en sciences informatiques, Année préparatoire au master en sciences informatiques, Université de Liège, Institut Montefiore, 2011.
- Pierre Geurts. «Chapitre 7 récurrence». Matière « Introduction à la théorie de l'informatique », 2ème Bachelier en sciences informatiques, Année préparatoire au master en sciences informatiques, Université de Liège, Institut Montefiore, 2012/2013.
- Frédéric Fürst. « complexité ». Cours «Algorithmique et programmation », Licence Informatique, Université de Picardie. 2015/2016
- Chebbar, « récursivité »; Algorithmique II. Faculté des sciences Rabat.