

Chapitre 9 – Notion de complexité

Quand on tente de résoudre un problème, la question du choix d'un algorithme se pose (en terme d'efficacité).

Généralités

- Un algorithme
 - doit être simple à mettre en œuvre et à comprendre,
 - doit utiliser intelligemment les ressources de l'ordinateur (place mémoire et temps d'exécution).

- Ces deux aspects sont souvent contradictoires.

- Le temps d'exécution d'un programme dépend
 - des données entrant dans le programme,
 - de la qualité du code généré par le compilateur,
 - de la nature et la vitesse d'exécution des instructions du microprocesseur,
 - de la complexité algorithmique du programme.

- Temps d'exécution d'un programme et complexité algorithmique sont intimement liés.

Temps d'exécution $T(n)$

- Assez souvent, le temps d'exécution d'un programme ne dépend pas de la nature précise des données mais de leur « taille ». Dans ces conditions, il est habituel de parler d'un temps d'exécution de $T(n)$ pour un programme portant sur des données de taille n .
- *Exemple* : $T(n) = cn^2$, où c est une constante.
- Il est possible de se représenter $T(n)$ comme le nombre d'instructions « élémentaires » exécutées par une machine formelle.

Complexité

- ❑ Pour un certain nombre de programmes, la complexité est une fonction de la nature des données et non de leur taille seulement.
- ❑ Dans ce cas, on définit $T(n)$ comme la complexité dans le pire des cas : mesure de la complexité maximum sur tous les ensembles de données possibles de taille n pour un programme donné.
- ❑ Il est aussi possible de définir la *complexité en moyenne* $T_{\text{moy}}(n)$, sur tous les ensembles de données de taille n (souvent difficile à calculer).
- ❑ *Exemple de formulation* : « la complexité d'un tel algorithme est proportionnelle à n^2 ».
- ❑ La constante de proportionnalité n'est pas précisée car elle est déterminée par les performances du compilateur et de la machine.

Notation O et Ω

- ❑ Pour une complexité algorithmique en $O(f(n))$, il existe deux constantes positives c et n_0 :
$$\forall n \geq n_0, T(n) \leq cf(n)$$
- ❑ Dire que $T(n)$ est en $O(f(n))$, c'est garantir que $f(n)$ est un majorant de la fonction de complexité $T(n)$.
- ❑ $g(n)$ est un minorant de $T(n)$: $T(n)$ est en $\Omega(g(n))$,
$$\exists c \text{ (constante positive) t.q. } T(n) \geq cg(n).$$
- ❑ Les programmes peuvent être comparés sur la base de leur fonction de complexité, à la constante de proportionnalité près.

Règle de la somme

Supposons que deux modules P_1 et P_2 aient une complexité $T_1(n)$ en $O(f(n))$ et $T_2(n)$ en $O(g(n))$

Alors la complexité de P_1 suivi de P_2 est $T_1(n)+T_2(n) = O(\max(f(n),g(n)))$.

Analyse de programmes

- Le seul paramètre acceptable pour l'évaluation de la complexité d'un programme est n , la taille des données.
 - La complexité de toute affectation, opération de lecture ou d'écriture peut en général se mesurer par $O(1)$.
 - La complexité d'une suite d'instructions est déterminée par la règle de la somme.
 - La complexité conditionnelle (si) est celle des instructions exécutées (règle de la somme pour un si-alors-sinon), plus celle de l'évaluation de la condition (généralement égale à $O(1)$).
 - La complexité d'une boucle est la somme cumulée, sur toutes les itérations, de la complexité des instructions exécutées dans le corps de la boucle plus le temps consacré à l'évaluation de la condition de sortie de la boucle (généralement égale à $O(1)$). Assez souvent, cette complexité est le produit du nombre d'itérations par la plus grande complexité rencontrée dans l'exécution d'une boucle (règle du produit).
 - Appels de procédures non récursives : on commence par celles qui n'en appellent aucune autre, puis celles qui appellent uniquement celles qu'on vient de traiter en incluant les complexités qu'on vient de calculer, et ainsi de suite.
 - Appels de procédures récursives : il faut construire une relation de récurrence pour $T(n)$.

Exemple 1 : tri à bulles

```
Const max = 100
Type tab = tableau [1..max] d'entiers
  Procédure Tribulle (E/S t : tab)
Var i,j, temp : entier
Début
  Pour i←1 à n-1 Inc +1 Faire
    Pour j←n à i+1 Inc -1 Faire
      Si t[j-1]>t[j]
        Alors temp←t[j-1]
          t[j-1]←t[j]
          t[j]←temp
      FinSi
    FinPour
  FinPour
Fin
```

Analyse dans le pire des cas :

L'évaluation de la condition du si est en $O(1)$.

Dans le pire des cas, elle sera toujours vraie.

Les instructions faites dans le Alors, sont en $O(1)$.

Nombre d'itérations du Pour j : n-i

Nombre d'itérations du Pour i : n-1

n-1

$$T(n) = \sum_{i=1}^{n-1} (n-i) = n(n-1)/2.$$

$T(n)$ est en $O(n^2)$.

Exemple 2 : fonction factorielle

```
Fonction fac (n : entier) : entier
Var f : entier
Début
    Si n ≤ 1
        Alors f ← 1
        Sinon f ← n * fac(n-1)
    FinSi
Retourner(f)
Fin
```

$$T(n) = c + T(n-1)$$

$$T(n-1) = c + T(n-2)$$

$$T(n) = c + [c + T(n-2)] = 2*c + T(n-2)$$

$$T(n) = i*c + T(n-i) \text{ si } n > i.$$

Pour $i=n-1$, $T(n) = c*(n-1) + T(1) = c*n + d$, où d est une constante positive.

$T(n)$ est en $O(n)$.