Université Clermont Auvergne
ISIMA 1A par apprentissage
Algorithmique

Florent Foucaud
2022-2023

TD3 - Complexité en temps, suite

1 Rappel de cours

1.1 Complexité en temps d'un algorithme

Pour un algorithme donné, on souhaite déterminer le nombre d'instructions de base $C(n)$ qui seront effectuées *dans le pire des cas*, pour une entrée de taille n (où $C:\mathbb{N}\to\mathbb{N}$). La taille est le nombre de bits nécessaires pour représenter l'entrée. On considère généralement qu'un nombre (par exemple) a une taille constante, un tableau a donc une taille proportionnelle à sa longueur.

Les instructions de base sont les affectations, les opérations mathématiques, les comparaisons, l'accès à un élément de tableau... et on considère que chacune prend une unité de temps.

On s'intéresse surtout à l'ordre de grandeur de $C(n)$, pour cela on utilise les *notations asymptotiques* suivantes (étant données deux fonctions $f,g:\mathbb{N}\to\mathbb{N}$) :

- $f(n)\in O(g(n))$ si il existe une constante $c\in\mathbb{R}$ et un rang $r\in\mathbb{N}$ tels que, pour tout entier $i\geq r$, on a $f(i)\leq c\cdot g(i)$. Cela veut dire que f ne croît pas de façon plus rapide que g (à un facteur constant près) lorsque le paramètre n est sufisamment grand.
- $f(n)\in\Omega(g(n))$ si il existe une constante $c\in\mathbb{R}$ et un rang $r\in\mathbb{N}$ tels que, pour tout entier $i\geq r$, on a $f(i)\geq c\cdot g(i)$. Cela veut dire que f ne croît pas de façon moins rapide que g (à un facteur constant près) lorsque le paramètre n est sufisamment grand.
- $f(n)\in\Theta(g(n))$ si $f(n)\in O(g(n))$ et $f(n)\in\Omega(g(n))$. Cela signifie que f et g se comportent de la même façon (à facteurs constants près) lorsque le paramètre n est sufisamment grand.

Remarque : si $f(n)\in O(g(n))$, alors $g(n)\in\Omega(f(n))$, et inversement.

Un *problème algorithmique* est défini par le type d'entrée attendu (par exemple : un entier, un tableau d'entiers, etc) et une tâche à effectuer (généralement, une sortie à renvoyer). Plusieurs algorithmes qui résolvent un même problème algorithmique donné peuvent exister, on cherche bien entendu à trouver l'algorithme le plus efficace possible parmi ceux-ci.

Une *classe de complexité* est un ensemble de problèmes algorithmiques pour lesquels le meilleur algorithme possible se comporte de manière similaire (par exemple, en termes de temps de calcul).

Les notations asymptotiques vues ci-dessus permettent de définir des classes de complexité classiques pour les problèmes algorithmiques, en utilisant des fonctions usuelles, par exemple (du meilleur au pire) :

- $C(n)\in\Theta(1)$ (complexité constante)
- $C(n)\in\Theta(\log(\log(n)))$
- $C(n)\in\Theta(\log(n))$ (complexité logarithmique)
- $C(n)\in\Theta(\sqrt{n})$
- $C(n)\in\Theta(n)$ (complexité linéaire)
- $C(n)\in\Theta(n\log(n))$
- $C(n)\in\Theta(n^c)$ pour une constante $c>1$ (complexité polynômiale)
- $C(n)\in\Theta(c^n)$ pour une constante $c>0$ (complexité exponentielle)
- $C(n)\in\Theta(n!)$ (qui est dans $O(n^n)$)...

2 Exercices

Exercice 1 (Recherche dichotomique récursive)**.**

Quelle est la complexité en temps $C(N)$ de l'algorithme de recherche dichotomique ci-dessous, appelé avec $m=1$ et $n=N$? Pour simplifier, on supposera que $N=2^k$ est une puissance de deux.

`rechercheDicho(`T`,`m`,`n`,`x`): calcul d'une position de` x `dans` T

`Entrée : Un tableau d'entiers` $T[1...N]$ `triés par ordre croissant, un entier` x`, deux positions` m `et` n

`Sortie :` 0 `si` $x \notin T$`, et une position de` x `dans` T `(entre` m `et` n`) sinon`

- `Si` $m == n$ `:`
  - ⋆ `Si` $T[m] == x$`:`
    `Retourner` m
  - ⋆ `Sinon :`
    `Retourner` 0
- `Sinon :`
  - ⋆ $k = \lfloor \frac{m+n}{2} \rfloor$
  - ⋆ `Si` $T[k] < x$ `:`
    `Retourner rechercheDicho(`T`,`$k+1$`,`n`,`x`)`
  - ⋆ `Sinon :`
    `Retourner rechercheDicho(`T`,`m`,`k`,`x`)`

Exercice 2 (Diviser pour régner : le tri par fusion)**.**

Quelle est la complexité en temps $C(N)$ de l'algorithme de tri par fusion ci-dessous ? Pour simplifier, on supposera que $N=2^k$ est une puissance de deux. Estimer d'abord la complexité de la sous-fonction `interClassement(`T_1`,`T_2`)`, puis représenter le comportement récursif de l'algorithme par un arbre des appels récursifs, pour en déduire la complexité globale.

`triFusion(`T`,`m`,`n`): tri du sous-tableau` $T[m...n]$ `d'entiers par ordre croissant`

`Entrée : Un tableau d'entiers` $T[1...N]$ `et deux positions` m`,`n `avec` $m \leq n$

`Sortie : Le sous-tableau` $T[m...n]$ `trié`

- `Si` $m < n$ `:`
  - ⋆ $k = \lfloor \frac{m+n}{2} \rfloor$
  - ⋆ T_1 = `triFusion(`T`,`m`,`k`)`
  - ⋆ T_2 = `triFusion(`T`,`$k+1$`,`n`)`
  - ⋆ $T[m...n]$ = `interClassement(`T_1`,`T_2`)`
- `Retourner` $T[m...n]$

interClassement(T_1,T_2):

Entrée : Deux tableaux d'entiers $T_1[1...N_1]$, $T_2[1...N_2]$ triés par ordre croissant

Sortie : L'union T de T_1 et T_2 triée par ordre croissant

- $i = 1$, $j = 1$, $k = 1$
- T est un tableau vide à N_1+N_2 éléments
- Tant que $i \leq N_1$ et $j \leq N_2$:
  - ⋆ Si $T_1[i] < T_2[j]$:
    $T[k] = T_1[i]$
    $i = i+1$
  - ⋆ Sinon :
    $T[k] = T_2[j]$
    $j = j+1$
  - ⋆ $k = k+1$
- Tant que $i \leq N_1$:
  $T[k] = T_1[i]$
  $i = i+1$
  $k = k+1$
- Tant que $j \leq N_2$:
  $T[k] = T_2[j]$
  $j = j+1$
  $k = k+1$
- Retourner T

Exercice 3 (Structures de données linéaires)**.**

On considère les structures de données linéaires de base :

- tableau
- tableau trié
- liste chaînée
- liste chaînée triée
- liste doublement chaînée
- liste doublement chaînée triée

Quelle est la classe de complexité des opérations suivantes, en fonction du nombre N d'éléments, pour chacune des structures de données ci-dessus ? Compléter le tableau.

- Accès/modification au/du $i$ème élément
- Insertion à la position i
- Recherche d'un élément x
- Accès à l'élément précédent
- Accès à l'élément suivant
- Recherche du minimum

- Recherche du maximum

	tableau	tableau trié	liste chaînée	liste chnée triée	liste doubl. chnée	liste doubl. chnée triée
Accès $i$ème élt						
Insertion pos. i						
Rech. élt x						
Rech. élt préc.						
Rech. élt suiv.						
Rech. min.						
Rech. max.						

Exercice 4 (Programmation dynamique pour trouver le plus grand sous-tableau croissant)**.**

Dans le TD2, nous avions vu un algorithme en temps exponentiel pour calculer le plus long sous-tableau d'éléments croissants (pas forcément consécutifs) dans un tableau donné de taille N. Cet algorithme testait tout simplement tous les $2^N - 1$ sous-tableaux possibles et vérifiait s'il était croissant.

On peut cependant résoudre ce problème de façon plus astucieuse, en temps polynomial, par la technique dite de la *programmation dynamique*. L'idée est de construire les solutions possibles de façon incrémentale.

On suppose qu'on a un tableau d'entiers $T[1...n]$. Soit $d[i]$ la longueur d'un plus long sous-tableau de T croissant dont le dernier élément est $T[i]$.

1. Exprimer $d[i]$ en fonction des valeurs $d[j]$ avec $j < i$.
2. En déduire un algorithme récursif pour calculer *la longueur* d'un plus long sous-tableau de T croissant.
3. L'algorithme récursif précédent est néanmoins exponentiel. Le transformer en algorithme incrémental.
4. Quelle est sa complexité en temps ?
5. Quelle petite modification peut-on faire pour aussi calculer, sur le même principe, le plus long sous-tableau croissant ?