Algorithmique – écriture de programmes efficaces et sûrs

Romain Gille

15/02/2016

On reprend les programmes précédents (QuickSort)

```
void quickSort(int[] T){
  int m = T.length;
  qS(T, 0, n);
}

void qS(int[] T, int i, int j){
  if(j - i > 1){
    int k = segmenter(T, i, j);
    qS(T, i, k);
    qS(T, k + 1, j);
  }
}
```

Hypothèse pour trouver le temps de calcul :

- $n = 2^p$ et $k = \dfrac{i+j}{2}$ à chaque appel

$T(n = 2^p) = \Theta(n) + 2 * T(\dfrac{n}{2} = 2^{p-1})$ ($\Theta(n) =$temps `segmenter` + temps `j-i > 1`)

$T(n \leq 2^p) = c =$ constante

$T(n \leq 2^0) = c$

$T(n = 2^1) = (\alpha 2^1 + \beta) + 2T(2^0) = (\alpha 2^1 + \beta) + 2c$

$T(n = 2^2) = (\alpha 2^2 + \beta) + 2T(2^1) = (\alpha 2^2 + \beta) + 2((\alpha 2^1 + \beta) + 2c)$

$T(n = 2^3) = (\alpha 2^3 + \beta) + 2((\alpha 2^2 + \beta) + 2((\alpha 2^1 + \beta) + 2c))$

$T(n = 2^3) = \alpha 3 * 2^3 + \beta(2^0 + 2^1 + 2^2) + c2^3$

$T(n = 2^3) = \alpha 3 * 2^3 + \beta(2^3 - 1) + c2^3$

$T(n = 2^3) = \alpha n \log(n) + (c + \beta)\dfrac{n}{2^3} - \beta$

$T(n = 2^3) = \dfrac{A}{\alpha} n \log(n) + Bn + C$

$T(n = 2^3) = \Theta(n \log(n))$

- Si $n \neq 2^p \rightarrow 2^p < n < 2^{p+1}$
  La forme du temps de calcul reste en $\Theta(n\log(n))$

  - A chaque appel de `segmenter(T, i, j)`, si $k = i$
    $\rightarrow$ `T[0:n]` est trié par ordre croissant.

$T(0) = T(1) = C = \text{constante}$

$T(n \geq 2) = (\alpha n + \beta) + T(0) + T(n-1)$

$T(0) = C$

$T(1) = C$

$T(2) = (\alpha 2 + \beta) + T(0) + T(1)$

$T(3) = (\alpha 3 + \beta) + T(0) + T(2)$

$T(3) = (\alpha 3 + \beta) + C + (\alpha 2 + \beta + T(0) + T(1))$

$T(n \geq 2) = \alpha \sum_{i=2}^{n} i + \beta(n-1) + nC$

$T(n \geq 2) = \alpha(\dfrac{n(n+1)}{2} - 1) + \beta(n-1) + nC$

$\dfrac{\alpha}{2}n^2 + (\beta + C)n - \beta = An^2 + Bn + C \rightarrow \Theta(n^2)$

```java
void qSS(int[] T, int i, int j){ // QuickSort Stochastique (désordre)
  if(j - i) > 1){
    Random rand = new Random(); // Soit r in [i:j] choisi au hasard
    int r = i + rand.nextInt(j - i); // nextInt(j - i) in [0:j - i]
    permuter(T, i, r);
    int k = segmenter(T, i, j);
    qSS(T, i, k);
    qSS(T, k + 1, j);
  }
}
```

Dual Pivot QuickSort (amélioration dans le cas de répétition d'éléments)

Nouvelle version de tri tableau Java depuis 2010

`T[i:k]` $<$ `T[k1]`
`T[k1]` $\leq$ `T[k1+1:k2]` $\leq$ `T[k2]`
`T[k2]` $<$ `T[k2:j']`

Synthèse

Méthode séquentielle

- `sommeTab()`
- Tri par sélection
- `segmenter()`

Méthode "Diviser pour régner"

- `sommeTab()` en découpant `T` en deux sous-tableaux
- `quickSort()`

Équations de récurence de temps de calcul

`sommeTab()` : $T(n) = 2T(\frac{n}{2}) + \Theta(1) \Rightarrow \Theta(n)$ ($\Theta(1) \rightarrow$ temps constant)

`sommeTab()` en parallélisme : $T(n) = T(\frac{n}{2}) + \Theta(1) \Rightarrow \Theta(\log(n))$

bonne situation de `quickSort()` : $T(n) = T(\frac{n}{2}) + \Theta(n) \Rightarrow \Theta(n \log(n))$

mauvaise situation de `quickSort()` $T(n) = T(n-1) + \Theta(n) \Rightarrow \Theta(n^2)$

Recherche d'une valeur dans `T[0:n]`

Recherche séquentielle

Sortie : `k` = indice de la première occurence de `x` ou si $x! \in$ `T[0:n]`, $k = n$

Initialisation : `k = 0`

Arrêt : `k = n` ($x! \in$ `T[0:n]`) ou `T[k]` = x (k est la première occurence de x)

Progression : `I(k)` et $k \neq n$ et `T[k]` $\neq x$
$\Rightarrow$ `I(k+1)`

```
int rS(int x, int[] T){
  int n = T.length;
  k = 0; //I(k)
  while(k != n && T[k] != x){ // I(k+1)
    k++; //I(k)
  }
  return k;
}
```