Algorithmique – écriture de programmes efficaces et sûrs

Romain Gille

03/02/2016

Invariant : I(s, k) : $s = \sum_{i \in [0:n]}$

On utilise le premier programme du cours précédent

```
int somme (int[] T){
  int n = T.length;
  int k = 0, s = 0; // I(s, k)
  while(k != n){    // I(s, k) et k != n => I(s + T[k], k + 1)
    s = s + T[k];   // I(s, k + 1)
    k++;            // I(s, k)
  }
  return s;
}
```

Temps de calcul

L'initialisation est de temps constant α.

Pour la boucle while :

- L'addition est une opération de temps constant => β
- L'accès à un élément d'un tableau est également de temps constant => β
- La vérification de `k != n` est de temps constant γ et exécuté (n + 1) fois

Donc le corps de boucle est de temps constant $\beta + (n+1)\gamma$.

Le temps d'exécution est : $T(n) = \alpha + n\beta + (n+1)\gamma$

$T(n) = (\alpha + \gamma) + n(\beta + \gamma) = A + Bn$

Efficacité d'un programme

C'est la forme du temps de calcul dans le pire cas d'éxécution exprimé en fonction de n (taille du problème).

On utilise le dernier programme du cours précédent

```
int somme(int[] T, int i, int j){
  if(j - i <= 0) return 0;
  if(j - i == 1) return T[i];
  else{
    int k = (i + j) / 2;
    int sg = somme(T, i, k);
    int sd = somme(T, k, j);
    return sg + sd;
  }
}
```

On traite le cas `somme(T, 0, n)`

- $T(n \leq 0) \rightarrow$ le programme s'exécute en temps constant α
- $T(n = 1) \rightarrow$ le programme s'éxecute en temps constant $\beta \neq \alpha$
- $n > 1$ et $n = 2^p$ avec $p \geq 1$
  - γ est le temps constant de la division par 2 et du `return sg + sd;`
  - $\delta = \alpha + \beta + \gamma$
  - le programme s'exécute en $\delta + 2 * T(2^{p-1})$

On teste avec des valeurs de p

$T(n = 2^0) = \beta$

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

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

$T(n = 2^3) = \delta + 2T(2^2) = \delta + 2(\delta + 2(\delta + 2\beta))$

$T(n = 2^3) = \delta(2^0 + 2^1 + 2^2) + 2^3\beta$

$T(n = 2^3) = \delta(2^3 - 1) + 2^3\beta$

$T(n = 2^3) = (\delta + \beta)2^3 - \delta$

On peut généraliser à $n = 2^p$

$T(n = 2^p) = (\beta + \gamma)2^p - \delta$

$T(n = 2^p) = A * 2^p + B$

$T(n = 2^p) = \Theta(2^p) = \Theta(n)$

En posant le problème en parallèle :

$T_{//}(n = 2^p) = p * \gamma + \beta = \Theta(log_2 n)$

Un premier problème de tri de tableau

Nous étudierons 4 algorithmes de tri

1. Tri par sélection
2. Tri “rapide” (quickSort) (ou tri par segmentation)
3. Tri par fusion (mergeSort)
4. Tri par tas (heapSort)

TRI interne (on ne travaille pas sur une copie du tableau) : `T[0:n]` est un tableau d’entiers

`T[0:n] = [t_0, t_1, ..., t_{n-1}]`

Calculer une permutation croissante de (t_0, t_1, . . . , t_{n-1})

`I(k) :` `T[0:k]` croissant et `T[0:k]`$\leq$`T[k:n]`

Initialisation : k = 0

Arrêt : k = n

Progression : `I(k)` et $k \neq n$ et $t_m =$ `min T[k:n]` et `T[k]`$= t_m$ et `T[m]`$= t_k \rightarrow I(k+1)$

Programme :

```
void triSelection(int[] T){
  int n = T.length;
  int k = 0;
  while(k != n){ // I(k) et k != n
    int m = indiceMin(T, k, n);
    int x = T[k];
    T[k] = T[m];
    T[m] = x; // I(k + 1)
    k = k + 1; // I(k)
  } // I(n) donc T[0:n] trié dans l'ordre croissant
}
```