ALGORITHMES: ELEMENTS D´ANALYSE

[Marion SZPIEG – Frédéric PEURIERE]

Analyser la complexité d´un algorithme

Etablir la preuve d´un algorithme:

- *variant de boucle*
- *Correction partielle par utilisation d'un invariant de boucle*

1. LA COMPLEXITE D´UN ALGORITHME:

☞ ETUDE D´UN EXEMPLE:

```
1 def RechercheMin (liste):
2     n= len(liste)
3     min=liste[0]
4     for i in range (1,n):
5         if min>liste[i]:
6             min=liste[i]
7     return min
```

Reprenons l'implémentation de l´algorithme étudié au chapitre précédent dans le langage PYTHON.

Essayons d´évaluer le nombre d´**opération élémentaires** (appelé *coût de l´algorithme* noté C(n)) effectuées par cet algorithme dans le pire des cas, c´est à dire lorsqu´on a un tableau de taille `n` contenant des valeurs dans l´ordre décroissant:

indice	**0**	**1**	**2**	**....**	**n-1**
valeur	**12**	**10**	**8**	**....**	**1**

ligne 2 → 1 opération (affectation simple)
ligne 3 → 1 opération (affectation simple)
ligne 4 → n-1 opérations (incrémentation de i)
ligne 5 → n-1 opérations (à chaque tour de boucle une comparaison est effectuée)
ligne 6 → n-1 opérations (à chaque tour de boucle une affectation est effectuée)
ligne 7 → 1 opération (renvoi de la valeur de `min`)

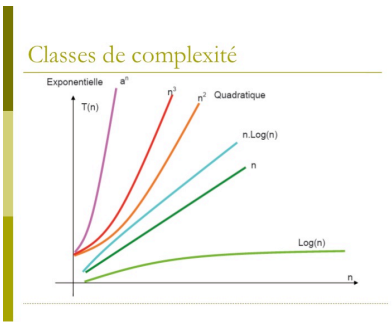
Bilan: Dans le pire des cas, l´algorithme effectue `C(n)=1+1+3(n-1)+1=3n` opérations élémentaires.

Et dans le meilleurs des cas? Montrez que: **`C(n)=2n+1`**

La seule chose que nous pouvons affirmer: **Le nombre d´opération élémentaires de cet algorithme dépend de la taille des données, elle est comprise entre `2n+1` et `3n`:**
$2n+1 \leq C(n) \leq 3n$

Nous observons que dans tous les cas, le coût de cet algorithme est *proportionnel* à la taille du tableau (n).

☞ DEFINITION DE LA COMPLEXITE EN TEMPS:



L´observation précédente nous permet d´affirmer que la complexité en temps de l´algorithme étudié est linéaire et nous écrirons simplement: La **complexité de cet algorithme** est O(n), elle est donc *linéaire*.

Nous verrons au chapitre prochain des algorithmes plus coûteux de complexité $O(n^2)$. On dit que leur complexité est *quadratique*.

De manière plus générale, on constate qu´un algorithme contenant une boucle simple est de complexité linéaire alors qu´une boucle imbriquée engendrera une complexité quadratique.

2. CORRECTION PARTIELLE D´UN ALGORITHME

"Testing shows the presence, not the absence of bugs"

E. W. Dijkstra

Comment prouver qu´un algorithme réalise bien ce qui est demandé ?
On le fait en deux étapes:

- **Correction partielle**: On vérifie qu´il renvoie bien ce qui est demandé. On utilise un `invariant de boucle`.
- **Terminaison**: On prouve d´abord que l´algorithme se termine. On utilise alors un `variant de boucle`. Nous n´étudions dans ce chapitre que la première étape.

☞ PRINCIPE DE LA CORRECTION PARTIELLE:

On se concentre dans cette étape sur l´analyse des boucles. On cherche à écrire une phrase qui reste vraie pendant tout son déroulement. Cette phrase est une **propriété** que l´on appelle **invariant de boucle**. Le raisonnement se déroule en trois étapes:

→ **Initialisation**: On vérifie que la propriété est vraie avant d´entrer dans la boucle.
→ **Conservation**: Si la propriété est vraie lors de l´exécution d´un tour de boucle (i), on montre qu´elle le reste au tour de boucle suivant (i+1).
→ **Terminaison**: La propriété restant vraie à la sortie de la dernière boucle, elle doit permettre de vérifier que l´algorithme fait ce qu´on attend de lui.

☞ ETUDE D'UN EXEMPLE

```
1  def puissance(n):
2      p=1
3      i=0
4      # avant la boucle
5      while i<n:
6          i+=1
7          p=p*2
8          # fin de boucle
9      # sortie de boucle
10     return p
```

Il semble que cet algorithme renvoie une puissance de deux. L´exposant étant l´entier n passé en entrée. On doit obtenir en sortie 2^n.

Proposons une *propriété invariante* dans la boucle: " ***Après la $k^{ème}$ itération, nous avons $p_k=2^k$*** "

→ **Initialisation**: Avant la première itération, la propriété $p_{avant} = 2^{k_{avant}}$ est vraie puisque $k_{avant}=0$ et $p_{avant}=1$ et: $2^0 = 1$

→ **Conservation**: Plaçons-nous maintenant en fin de boucle d´une itération quelconque (k). A cet endroit, la propriété $p_k = 2^k$ est considérée comme vraie. Est-elle toujours vraie à la fin de la boucle suivante? A cet endroit: $p_{k+1} = 2^k \times 2$ (calcul ligne 7). Ce qui s´écrit: $p_{k+1} = 2^{k+1}$. La propriété reste bien vérifiée au rang k+1.

→ **Terminaison**: A la sortie de la dernière boucle: $k_{fin}=n$ et **p_{fin} contient la valeur 2^n**. A cette dernière étape, nous vérifions bien que l´algorithme calcule ce que nous voulions.