Cours de Programmation Impérative: Complexité, une introduction

Julien David

A101 - david@lipn.univ-paris13.fr

Efficacité d'une fonction

Mesurer l'efficacité d'un algorithme

Comparer des fonctions

On l'a vu en TD, il existe parfois plusieurs algorithmes pour résoudre un même problème.

Efficacité

On veut donc déterminer "la plus efficace".

Efficacité ?

Qu'est ce que l'efficacité ?

On veut connaître la quantité de ressources utilisée par l'algorithme.

Quel genre de ressources ?

Généralement, on s'interesse :

- Au temps.
- À l'espace mémoire.

Mais on pourrait aussi considérer :

- l'énergie, le coût, ...

Les autres paramètres sont généralement liés aux deux premiers.
Ils sont donc rarement étudiés en tant que tels.

Efficacité en temps

Efficacité en temps

On ne compare pas réellement le temps d'exécution des algorithmes.

Les raisons

- du mathématicien :
  - montrer qu'un algorithme est plus rapide qu'un autre sur des exemples n'est pas une preuve.
- de l'informaticien :
  - d'autres programmes tournent sur la machine et peuvent géner la mesure,
  - les mesures sont trop dépendantes de l'ordinateur.
- du flemmard :
  - il faudrait programmer les fonctions pour avoir la réponse.

Efficacité en temps

Unité de mesure

On veut estimer le nombre d'instructions effectuées par l'algorithme.

Et pour estimer le temps . . .

Il est possible d'estimer le temps nécessaire à partir du nombre d'instructions :

- La fréquence d'un processeur est le nombre d'instructions qu'il peut effectuer en 1 seconde.

Efficacité en espace mémoire

Unité de mesure

Pour la mémoire, on s'interesse au nombre de cases mémoire.

Exemple

```
1 void swap(int * x,int * y){
2   int tmp=*x;
3   *x=*y;
4   *y=tmp;
5 }
```

Coût en temps

3 instructions.

Coût en espace

3 cases mémoires (variables).

Précision

Précision

En réalité, on veut juste des ordres de grandeurs.

C'est à dire ?

On veut pouvoir donner un **ordre de grandeur** du nombre d'instructions **en fonction des paramètres de la fonction**.
On parle de :

- **Complexité en temps**
- **Complexité en espace**

Exemple

```
1 unsigned int addition(unsigned int x,unsigned int y){
2   for(;y>0;y--)
3     x++;
4   return x;
5 }
```

Coût en temps

- Le nombre d'instructions dépend de la valeur de **y**.
- Il est même proportionnel à **y**.
- On dit que la complexité en temps est $\mathcal{O}(y)$.

Exemple

```
1 unsigned int addition(unsigned int x,unsigned int y){
2   for(;y>0;y--)
3     x++;
4   return x;
5 }
```

Coût en espace

- le nombre de variables allouées ne dépend pas de la valeur de x et y.
- le nombre de variables allouées est donc **constant**.
- on dit que la complexité en espace est $\mathcal{O}(1)$.

Exemple 2

```
1 void swap(int * x,int * y){
2   int tmp=*x;
3   *x=*y;
4   *y=tmp;
5 }
```

Complexité en temps

La complexité en temps de la fonction `swap` est $\mathcal{O}(1)$.

Complexité en espace

La complexité en espace de la fonction `swap` est $\mathcal{O}(1)$.

Exemple d'ordres de grandeur

$\mathcal{O}(1)$	Constant
$\mathcal{O}(\log n)$	Logarithmique
$\mathcal{O}(n)$	Linéaire
$\mathcal{O}(n \log n)$	Quasi-linéaire
$\mathcal{O}(n^2)$	Quadratique
$\mathcal{O}(n^k)$ avec k constant	Polynomial
$\mathcal{O}(k^n)$ avec k constant	Exponentiel

Ordres de grandeur

$\log_2 n$	n	$10n$	n^2	1.1^n	2^n	n^n
~ 4	10	100	100	~ 3	1024	10^{10}
~ 6	50	500	2500	~ 118	$1,12 \times 10^{15}$	$8,88 \times 10^{84}$
~ 7	100	1000	10000	~ 13781	$1,26 \times 10^{30}$	1×10^{200}
~ 9	500	5000	$2,5 \times 10^5$	$4,9 \times 10^{20}$	$3,2 \times 10^{150}$	3×10^{1349}
~ 10	1000	10000	1×10^6	$2,4 \times 10^{41}$	$1,07 \times 10^{301}$	1×10^{3000}

Temps de calcul

Temps de calcul

Supposons que l'on dispose d'un ordinateur avec un processeur de 2 Ghz.
On peut donc effectuer 2×10^9 instructions par seconde.

Temps de calcul

Les calculs qui suivent sont faits mains et bidouillés pour être plus lisibles.
Le but est de rendre compte des ordres de grandeurs.

$\log_2 n$	$n(valeur)$	$n(temps)$	n^2	1.1^n	2^n	n^n
$2ns$	10	$5ns$	$50ns$	$1,5ns$	$512ns$	5s
$3ns$	50	$25ns$	$1,25\mu s$	$59ns$	175 heures	10^{64} siècles
$3,5ns$	100	$50ns$	$5\mu s$	$6,8\mu s$	10^{11} siècles	–
$4,5ns$	500	$250ns$	$125\mu s$	76 siècles	–	–
$5ns$	10^3	$500ns$	$500\mu s$	-	–	–
$7ns$	10^4	$5\mu s$	$50ms$	–	–	–
$8ns$	10^5	$50\mu s$	$5s$	–	–	–
$10ns$	10^6	$0,5ms$	8 *min* 20	–	–	–
$15ns$	10^9	$0,5s$	16 ans	–	–	–

Notation de Landau

Les bornes

Borne supérieure

On dit qu'une fonction f est **majoré** par une fonction g,

$$\text{noté } f(n) = \mathcal{O}(g(n))$$

s'il existe une constante $c \in \mathbb{R}_+^*$ telle que :

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c$$

Borne supérieure : exemple 1

```
1 void affiche_pair(int n){
2   if(n%2==0) /*n est pair*/
3     printf("oui\n");
4 }
```

Pire des cas

- Dans le pire des cas, la fonction effectue 2 instructions.
- C'est le cas où n est pair.
- Soit $f(n)$ la fonction qui mesure la complexité en temps de la fonction `affiche_pair`.
- $f(n) = \mathcal{O}(1)$ car

$$\lim_{n \to \infty} \frac{f(n)}{1} \leq 2$$

Borne supérieure : exemple 2

```
1 void affiche_tout(int n){
2   int i;
3   if(n%2==0){
4     for(i=0;i<n;i++){
5       printf("%d\n",i);
6     }
7   }
8 }
```

Pire des cas

- Dans le pire des cas, la fonction effectue $3n + 4$ instructions.
- C'est le cas où n est pair.
- Soit $f(n)$ la fonction qui mesure la complexité en temps de la fonction `affiche_tout`.
- $f(n) = \mathcal{O}(n)$ car

$$\lim_{n \to \infty} \frac{f(n)}{n} \leq 4$$

Les bornes

Borne inférieure

On dit que f est **minorée** par une fonction g,

$$\text{noté } f(n) = \Omega(g(n))$$

s'il existe une constante $c \in \mathbb{R}_+^*$ telle que :

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \geq c$$

Borne inférieure : exemple 1

```
1 void affiche_pair(int n){
2   if(n%2==0) /*n est pair*/
3     printf("oui\n");
4 }
```

Meilleur des cas

- Dans le meilleur des cas, la fonction effectue 1 instruction.
- C'est le cas où n est impair.
- Soit $f(n)$ la fonction qui mesure la complexité en temps de la fonction `affiche_pair`.
- $f(n) = \Omega(1)$ car

$$\lim_{n \to \infty} \frac{f(n)}{1} \geq 1$$

Borne inférieure : exemple 2

```
1 void affiche_tout(int n){
2   int i;
3   if(n%2==0){
4     for(i=0;i<n;i++){
5       printf("%d\n",i);
6     }
7   }
8 }
```

Meilleur des cas

- Dans le meilleur des cas, la fonction effectue 1 instruction.
- C'est le cas où n est impair.
- Soit $f(n)$ la fonction qui mesure la complexité en temps de la fonction `affiche_tout`.
- $f(n) = \Omega(1)$ car

$$\lim_{n \to \infty} \frac{f(n)}{1} \geq 1$$

Les bornes

Équivalence asymptotique

$$f(n) = \Theta(g(n))$$

$$\Longleftrightarrow$$

$$f(n) = \mathcal{O}(g(n)) \text{ et } f(n) = \Omega(g(n))$$

Borne inférieure : exemple 1

```
1 void affiche_pair(int n){
2   if(n%2==0) /*n est pair*/
3     printf("oui\n");
4 }
```

Équivalent asymptotique

On sait que :

- $f(n) = \mathcal{O}(1)$,
- $f(n) = \Omega(1)$,
- donc $f(n) = \Theta(1)$.

ça fait beaucoup de choses à retenir...

La plupart du temps

On s'intéresse à la complexité dans le pire des cas.

- on cherche donc l'algorithme le plus efficace dans le pire des cas.

Complexité de itérative

Les instructions usuelles

Les opérations suivantes s'effectuent en temps constant :

- $+, -, *, /, \%,$
- $\&, \&\&, ||, =, ==,$
- $>, <, \ldots$

En effet ces opérations sont effectuées sur des variables scalaires.
La taille de ces variables étant fixées, le temps nécessaire pour les manipuler aussi.

La fonction `printf`

- Si on affiche des variables scalaires, la fonction `printf` s'effectue en temps constant.
- Si on affiche une chaîne de caractère de longueur ℓ, la complexité en temps de `printf` est $\mathcal{O}(\ell)$

Allocation de mémoire

- La complexité en espace d'une fonction itérative est la quantité de mémoire allouée au début de la fonction.

Complexité : les boucles

Complexité en temps

Le coût en temps d'une boucle :

$$\text{coût d'une itération} \times \text{nombre d'itérations}$$

Complexité : les boucles `while`

Complexité en temps

Dans le cas de la boucle `while`, on ne sait pas exactement combien d'itérations la boucle effectue.
On étudie donc :

- le pire des cas,
- le meilleur des cas.

Complexité : les boucles `while`

```
1 int recherche_element(int * tab,int n,int x){
2   int i=0;
3   while(i<n){
4     if(tab[i]==x)
5       return i;
6   }
7   return -1;
8 }
```

Complexité en temps

- Le pire des cas : x n'appartient pas au tableau, on parcourt tout le tableau.

$$\mathcal{O}(n)$$

- Le meilleur des cas : `tab[0]=x` sort de la boucle en 1 étape.

$$\Omega(1)$$

Complexité : les boucles `for`

Complexité en temps

Dans le cas de la boucle `for`, on sait exactement combien d'itération la boucle effectue.

Complexité : les boucles `for`

```
1 void afficher_tab(int * tab,int n){
2   int i;
3   for(i=0;i<n;i<++){
4     printf("%d\n",tab[i]);
5   }
6 }
```

Complexité en temps

On parcours toutes les cases du tableau.

$$\Theta(n)$$

Complexité de récursive

Complexité d'une fonction récursive

Complexité en temps

Le coût en temps d'une fonction récursive est égal à :

$$\text{coût en temps d'un appel} \times \text{nombre d'appels}$$

Complexité en espace

La complexité en espace d'une fonction récursive est égal à :

$$\text{coût en espace d'un appel} \times \text{nombre maximum d'appels imbriqués}$$

Exemples

```
1 unsigned int puissance(unsigned int a, unsigned int n){
2   if(n==0)
3     return 1;
4   return a*puissance(a,n-1);
5 }
```

Complexité en temps

- Coût d'un appel : $\mathcal{O}(1)$,
- Nombre d'appel : $\mathcal{O}(n)$,

La complexité en temps est donc $\mathcal{O}(n)$.

Exemples

```
1 unsigned int puissance(unsigned int a, unsigned int n){
2   if(n==0)
3     return 1;
4   return a*puissance(a,n-1);
5 }
```

Complexité en **espace**

- Coût d'un appel : $\mathcal{O}(1)$,
- Nombre d'appel : $\mathcal{O}(n)$,

La complexité en espace est donc $\mathcal{O}(n)$.

Exemples

```
1 unsigned int puissance(unsigned int a,unsigned int n){
2   if(n==0)
3     return 1;
4   if(n%2==0) /* Si n est pair */
5     return puissance(a*a,n/2);
6   return a* puissance(a*a,n/2);
7 }
```

Complexité en temps

- Coût d'un appel : $\mathcal{O}(1)$
- Nombre d'appel : À chaque appel, le paramètre n est divisé par deux.

La complexité en temps est donc $\mathcal{O}(\log n)$.

Fibonacci en récursif

```
1 unsigned int fibo(unsigned int n){
2   if(n==0||n==1)
3     return 1;
4   return fibo(n-1)+fibo(n-2);
5 }
```

- Quelle est la complexité en temps ?
- Quelle est la complexité en espace ?

Fibonacci en itératif

```
1 unsigned int fibo_it(unsigned int n){
2   unsigned int cur=1, prec=1;
3   unsigned int i;
4   for(i=0;i<n;i++){
5     cur=cur+prec;
6     prec=cur-prec;
7   }
8   return cur;
9 }
```

- Quelle est la complexité en temps ?
- Quelle est la complexité en espace ?

Réponses

Récursif

- Complexité en temps : $\Theta(1,618^n)$
- Complexité en espace : $\Theta(n)$

Itératif

- Complexité en temps : $\Theta(n)$
- Complexité en espace : $\Theta(1)$

À titre d'exemple

Il faut plusieurs siècles à la première fonction pour calculer *fibonacci*(100) et seulement quelques nano-secondes à la deuxième.