Complexité algorithmique

Jean-Pierre Becirspahic
Lycée Louis-Le-Grand

Complexité d'un algorithme

Déterminer la complexité d'un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

On ne mesure pas les valeurs exactes (du nombre d'octets nécessaires ou du nombre de secondes) mais uniquement des ordres de grandeurs en fonction des paramètres d'entrée.

Complexité d'un algorithme

Déterminer la complexité d'un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Ces deux fonctions calculent le nombre de diviseurs d'un entier n :

```python
def diviseurs1(n):
    d = 0
    k = 1
    while k <= n:
        if n % k == 0:
            d += 1
        k += 1
    return d
```

```python
def diviseurs2(n):
    d = 0
    k = 1
    while k * k < n:
        if n % k == 0:
            d += 2
        k += 1
    if k * k == n:
        d += 1
    return d
```

Complexité d'un algorithme

Déterminer la complexité d'un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Ces deux fonctions calculent le nombre de diviseurs d'un entier n :

```python
def diviseurs1(n):
    d = 0
    k = 1
    while k <= n:
        if n % k == 0:
            d += 1
        k += 1
    return d
```

```python
def diviseurs2(n):
    d = 0
    k = 1
    while k * k < n:
        if n % k == 0:
            d += 2
        k += 1
    if k * k == n:
        d += 1
    return d
```

Le temps d'exécution de `diviseurs1` est de l'ordre de $n\tau_1 + \tau_2$ donc proportionnel à n pour de grandes valeurs de n.

Complexité d'un algorithme

Déterminer la complexité d'un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Ces deux fonctions calculent le nombre de diviseurs d'un entier n :

```python
def diviseurs1(n):
    d = 0
    k = 1
    while k <= n:
        if n % k == 0:
            d += 1
        k += 1
    return d
```

```python
def diviseurs2(n):
    d = 0
    k = 1
    while k * k < n:
        if n % k == 0:
            d += 2
        k += 1
    if k * k == n:
        d += 1
    return d
```

Le temps d'exécution de `diviseurs2` est de l'ordre de $\sqrt{n}\tau_1' + \tau_2'$ donc proportionnel à $\sqrt{n}$ pour de grandes valeurs de n.

Complexité d'un algorithme

Déterminer la complexité d'un algorithme, c'est évaluer les ressources nécessaires à son exécution :

- quantité de mémoire requise ;
- temps de calcul à prévoir.

Ces deux fonctions calculent le nombre de diviseurs d'un entier n :

```python
def diviseurs1(n):
    d = 0
    k = 1
    while k <= n:
        if n % k == 0:
            d += 1
        k += 1
    return d
```

```python
def diviseurs2(n):
    d = 0
    k = 1
    while k * k < n:
        if n % k == 0:
            d += 2
        k += 1
    if k * k == n:
        d += 1
    return d
```

- multiplier n par 100 multiplie le temps d'exécution de `diviseurs1` par 100 ;
- multiplier n par 100 multiplie le temps d'exécution de `diviseurs2` par 10.

Instructions élémentaires

Il est nécessaire de préciser les instructions élémentaires disponibles, c′est-à-dire les opérations de coût constant :

- opérations arithmétiques ;
- comparaisons de données élémentaires ;
- transferts de données ;
- instructions de contrôle.

Instructions élémentaires

Il est nécessaire de préciser les instructions élémentaires disponibles, c'est-à-dire les opérations de coût constant :

- opérations arithmétiques ;
- comparaisons de données élémentaires ;
- transferts de données ;
- instructions de contrôle.

Mais on ne peut s'abstraire complètement d'une connaissance précise du fonctionnement interne de la machine.

Instructions élémentaires

Il est nécessaire de préciser les instructions élémentaires disponibles, c'est-à-dire les opérations de coût constant :

- opérations arithmétiques ;
- comparaisons de données élémentaires ;
- transferts de données ;
- instructions de contrôle.

Mais on ne peut s'abstraire complètement d'une connaissance précise du fonctionnement interne de la machine.

> *Les entiers* PYTHON *sont de type long : les opérations arithmétiques ne sont pas de coût constant.*

→ Sauf mention contraire on les supposera de taille bornée.

Instructions élémentaires

Il est nécessaire de préciser les instructions élémentaires disponibles, c'est-à-dire les opérations de coût constant :

- opérations arithmétiques ;
- comparaisons de données élémentaires ;
- transferts de données ;
- instructions de contrôle.

Mais on ne peut s'abstraire complètement d'une connaissance précise du fonctionnement interne de la machine.

> *La comparaison entre chaînes de caractères n'est pas de coût constant.*

→ Seule la comparaison entre deux caractères sera supposée de coût constant.

Instructions élémentaires

Il est nécessaire de préciser les instructions élémentaires disponibles, c'est-à-dire les opérations de coût constant :

- opérations arithmétiques ;
- comparaisons de données élémentaires ;
- transferts de données ;
- instructions de contrôle.

Mais on ne peut s'abstraire complètement d'une connaissance précise du fonctionnement interne de la machine.

La recopie d'un tableau entier n'est pas de coût constant.

$\rightarrow$ Seule la copie d'une case d'un tableau sera supposée de coût constant.

Instructions élémentaires

Quelques questions restent en suspens :

- coût de l'ajout d'un élément dans un tableau (méthode append) ?
- coût de la suppression d'un élément dans un tableau (méthode pop) ?

Pour y répondre, il faut étudier sur l'implémentation des tableaux en PYTHON.

Notations mathématiques

La taille de l'entrée est un (ou plusieurs) entiers dont dépendent les paramètres du problème :

- nombre d'éléments d'un tableau ;
- nombre de bits nécessaire à la représentation des données ;
- nombre de sommets et d'arêtes d'un graphe ;
- etc.

Notations mathématiques

La taille de l'entrée est un (ou plusieurs) entiers dont dépendent les paramètres du problème :

- nombre d'éléments d'un tableau ;
- nombre de bits nécessaire à la représentation des données ;
- nombre de sommets et d'arêtes d'un graphe ;
- etc.

Pour exprimer l'*ordre de grandeur* du nombre d'opérations élémentaires requis par l'algorithme, on utilise les notations de LANDAU :

- $f(n) = \mathrm{O}(\alpha_n) \iff \exists B > 0 \;\Big|\; f(n) \leqslant B\alpha_n$
- $f(n) = \Omega(\alpha_n) \iff \exists B > 0 \;\Big|\; A\alpha_n \leqslant f(n)$
- $f(n) = \Theta(\alpha_n) \iff f(n) = \mathrm{O}(\alpha_n) \text{ et } f(n) = \Omega(\alpha_n)$

$$\iff \exists A, B > 0 \;\Big|\; A\alpha_n \leqslant f(n) \leqslant B\alpha_n$$

Notations mathématiques

La taille de l'entrée est un (ou plusieurs) entiers dont dépendent les paramètres du problème :

- nombre d'éléments d'un tableau ;
- nombre de bits nécessaire à la représentation des données ;
- nombre de sommets et d'arêtes d'un graphe ;
- etc.

Pour exprimer l'*ordre de grandeur* du nombre d'opérations élémentaires requis par l'algorithme, on utilise les notations de LANDAU :

- $f(n) = \mathrm{O}(\alpha_n) \iff \exists B > 0 \,\big|\, f(n) \leqslant B\alpha_n$
- $f(n) = \Omega(\alpha_n) \iff \exists B > 0 \,\big|\, A\alpha_n \leqslant f(n)$
- $f(n) = \Theta(\alpha_n) \iff f(n) = \mathrm{O}(\alpha_n) \text{ et } f(n) = \Omega(\alpha_n)$

$$\iff \exists A, B > 0 \,\big|\, A\alpha_n \leqslant f(n) \leqslant B\alpha_n$$

La notation la plus fréquente est le O, en sous-entendant qu'il existe des configurations de l'entrée pour lesquelles $f(n)$ est effectivement proportionnel à α_n.

Ordre de grandeur et temps d'exécution

En s'appuyant sur une base de 10^9 opérations par seconde on obtient :

	$\log n$	n	$n \log n$	n^2	n^3	2^n
10^2	7 ns	100 ns	0,7 µs	10 µs	1 ms	$4 \cdot 10^{13}$ a
10^3	10 ns	1 µs	10 µs	1 ms	1 s	10^{292} a
10^4	13 ns	10 µs	133 µs	100 ms	17 s	
10^5	17 ns	100 µs	2 ms	10 s	11,6 j	
10^6	20 ns	1 ms	20 ms	17 mn	32 a	

Ordre de grandeur et temps d'exécution

En s'appuyant sur une base de 10^9 opérations par seconde on obtient :

	$\log n$	n	$n \log n$	n^2	n^3	2^n
10^2	7 ns	100 ns	0,7 µs	10 µs	1 ms	$4 \cdot 10^{13}$ a
10^3	10 ns	1 µs	10 µs	1 ms	1 s	10^{292} a
10^4	13 ns	10 µs	133 µs	100 ms	17 s	
10^5	17 ns	100 µs	2 ms	10 s	11,6 j	
10^6	20 ns	1 ms	20 ms	17 mn	32 a	

Les coûts *raisonnables* sont les coûts :

- logarithmique en $\mathrm{O}(\log n)$;
- linéaire en $\mathrm{O}(n)$;
- semi-linéaire en $\mathrm{O}(n \log n)$;
- quadratique en $\mathrm{O}(n^2)$.

Au delà, les coûts sont souvent prohibitifs au delà des grandeurs moyennes (coût polynomial) voire petites (coût exponentiel).

Exercice

Donner la complexité des algorithmes suivants :

```python
def f1(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

Exercice

Donner la complexité des algorithmes suivants :

```python
def f1(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

$\mathrm{O}(n^2)$

```python
def f2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

Exercice

Donner la complexité des algorithmes suivants :

```
def f1(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

$O(n^2)$

```
def f2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

$O(n^2)$

```
def f3(n):
    x = 0
    for i in range(n):
        j = 0
        while j * j < i:
            x += 1
            j += 1
    return x
```

Exercice

Donner la complexité des algorithmes suivants :

```python
def f1(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

$\mathrm{O}(n^2)$

```python
def f2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x += 1
    return x
```

$\mathrm{O}(n^2)$

```python
def f3(n):
    x = 0
    for i in range(n):
        j = 0
        while j * j < i:
            x += 1
            j += 1
    return x
```

$\mathrm{O}(n\sqrt{n})$

Exercice

Donner la complexité des algorithmes suivants :

```python
def f4(n):
    x, i = 0, n
    while i > 1:
        x += 1
        i //= 2
    return x
```

Exercice

Donner la complexité des algorithmes suivants :

```
def f4(n):
    x, i = 0, n
    while i > 1:
        x += 1
        i //= 2
    return x
```

$O(\log n)$

```
def f5(n):
    x, i = 0, n
    while i > 1:
        for j in range(n):
            x += 1
        i //= 2
    return x
```

Exercice

Donner la complexité des algorithmes suivants :

```python
def f4(n):
    x, i = 0, n
    while i > 1:
        x += 1
        i //= 2
    return x
```

$O(\log n)$

```python
def f5(n):
    x, i = 0, n
    while i > 1:
        for j in range(n):
            x += 1
        i //= 2
    return x
```

$O(n \log n)$

```python
def f6(n):
    x, i = 0, n
    while i > 1:
        for j in range(i):
            x += 1
        i //= 2
    return x
```

Exercice

Donner la complexité des algorithmes suivants :

```python
def f4(n):
    x, i = 0, n
    while i > 1:
        x += 1
        i //= 2
    return x
```

$O(\log n)$

```python
def f5(n):
    x, i = 0, n
    while i > 1:
        for j in range(n):
            x += 1
        i //= 2
    return x
```

$O(n \log n)$

```python
def f6(n):
    x, i = 0, n
    while i > 1:
        for j in range(i):
            x += 1
        i //= 2
    return x
```

$O(n)$

Différents types de complexité

Certains algorithmes ont un temps d'exécution qui dépend non seulement de la taille mais des données elles-mêmes. Dans ce cas on distingue :

- la complexité dans le pire des cas : c'est un majorant du temps d'exécution possible pour toutes les entrées possibles d'une même taille. On l'exprime en général à l'aide de la notation O.
- la complexité dans le meilleur des cas : c'est un minorant du temps d'exécution possible pour toutes les entrées possibles d'une même taille. On l'exprime en général à l'aide de la notation Ω.
- la complexité en moyenne : c'est une évaluation du temps d'exécution moyen portant sur toutes les entrées possible d'une même taille supposées équiprobables.

Différents types de complexité

Certains algorithmes ont un temps d'exécution qui dépend non seulement de la taille mais des données elles-mêmes. Dans ce cas on distingue :

- la complexité dans le pire des cas : c'est un majorant du temps d'exécution possible pour toutes les entrées possibles d'une même taille. On l'exprime en général à l'aide de la notation O.
- la complexité dans le meilleur des cas : c'est un minorant du temps d'exécution possible pour toutes les entrées possibles d'une même taille. On l'exprime en général à l'aide de la notation Ω.
- la complexité en moyenne : c'est une évaluation du temps d'exécution moyen portant sur toutes les entrées possible d'une même taille supposées équiprobables.

On peut définir la complexité spatiale d'un algorithme : évaluation de la consommation en espace mémoire.
La complexité spatiale est bien moins que la complexité temporelle un frein à l'utilisation d'un algorithme : on dispose aujourd'hui le plus souvent d'une quantité pléthorique de mémoire vive.

Algorithmes de recherche dans un tableau

Recherche séquentielle

```python
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

- Dans le meilleur des cas une seule comparaison est effectuée : la complexité est un $\Theta(1)$;

Algorithmes de recherche dans un tableau

Recherche séquentielle

```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

- Dans le meilleur des cas une seule comparaison est effectuée : la complexité est un $\Theta(1)$;
- dans le pire des cas n comparaisons sont effectuées : la complexité est un $\Theta(n)$.

Algorithmes de recherche dans un tableau

Recherche séquentielle

```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

- Dans le meilleur des cas une seule comparaison est effectuée : la complexité est un $\Theta(1)$;
- dans le pire des cas n comparaisons sont effectuées : la complexité est un $\Theta(n)$.

Dans tous les cas la complexité est un $O(n)$.

Algorithmes de recherche dans un tableau

Recherche dichotomique dans un tableau trié

```python
def cherche_dicho(x, l):
    i, j = 0, len(l)
    while i < j:
        k = (i + j) // 2
        if l[k] == x:
            return True
        elif l[k] > x:
            j = k
        else:
            i = k + 1
    return False
```

- Dans le meilleur des cas une seule comparaison est effectuée : la complexité est un $\Theta(1)$;

Algorithmes de recherche dans un tableau

Recherche dichotomique dans un tableau trié

```
def cherche_dicho(x, l):
    i, j = 0, len(l)
    while i < j:
        k = (i + j) // 2
        if l[k] == x:
            return True
        elif l[k] > x:
            j = k
        else:
            i = k + 1
    return False
```

- Dans le meilleur des cas une seule comparaison est effectuée : la complexité est un $\Theta(1)$;
- dans le pire des cas le nombre de comparaison effectuées est proportionnel à $\log n$: la complexité est un $\Theta(\log n)$.

En effet, $C(n) = C(n/2) + \Theta(1) \Longrightarrow C(n) = \Theta(\log n)$.

Algorithmes de recherche dans un tableau

Recherche dichotomique dans un tableau trié

```python
def cherche_dicho(x, l):
    i, j = 0, len(l)
    while i < j:
        k = (i + j) // 2
        if l[k] == x:
            return True
        elif l[k] > x:
            j = k
        else:
            i = k + 1
    return False
```

- Dans le meilleur des cas une seule comparaison est effectuée : la complexité est un $\Theta(1)$;
- dans le pire des cas le nombre de comparaison effectuées est proportionnel à $\log n$: la complexité est un $\Theta(\log n)$.

En effet, $C(n) = C(n/2) + \Theta(1) \Longrightarrow C(n) = \Theta(\log n)$.

Dans tous les cas, la complexité est donc un $O(\log n)$.

Complexité moyenne

de la recherche séquentielle

```python
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

On suppose que les éléments du tableau sont des entiers distribués de façon équiprobable entre 1 et $k \in \mathbb{N}^*$.

- $(k-1)^n$ tableaux ne contiennent pas l'élément que l'on cherche et dans ce cas l'algorithme procède à n comparaisons.

Complexité moyenne

de la recherche séquentielle

```python
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

On suppose que les éléments du tableau sont des entiers distribués de façon équiprobable entre 1 et $k \in \mathbb{N}^*$.

- $(k-1)^n$ tableaux ne contiennent pas l'élément que l'on cherche et dans ce cas l'algorithme procède à n comparaisons.
- Dans le cas contraire, l'entier recherché est dans le tableau et sa première occurrence est dans la ie case avec la probabilité $\dfrac{(k-1)^{i-1}}{k^i}$. L'algorithme réalise alors i comparaisons.

Complexité moyenne

de la recherche séquentielle

```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

On suppose que les éléments du tableau sont des entiers distribués de façon équiprobable entre 1 et $k \in \mathbb{N}^*$.
La complexité moyenne est donc égale à :

$$C(n) = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right).$$

Complexité moyenne

de la recherche séquentielle

```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

On suppose que les éléments du tableau sont des entiers distribués de façon équiprobable entre 1 et $k \in \mathbb{N}^*$.
La complexité moyenne est donc égale à :

$$C(n) = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right).$$

Lorsque k est petit devant n nous avons $C(n) \approx k$: la complexité moyenne est constante.

Complexité moyenne

de la recherche séquentielle

```
def cherche(x, l):
    for y in l:
        if y == x:
            return True
    return False
```

On suppose que les éléments du tableau sont des entiers distribués de façon équiprobable entre 1 et $k \in \mathbb{N}^*$.
La complexité moyenne est donc égale à :

$$C(n) = \frac{(k-1)^n}{k^n} \times n + \sum_{i=1}^{n} \frac{(k-1)^{i-1}}{k^i} \times i = k\left(1 - \left(1 - \frac{1}{k}\right)^n\right).$$

Lorsque k est petit devant n nous avons $C(n) \approx k$: la complexité moyenne est constante.
Lorsque n est petit devant k, $C(n) \approx n$ et la complexité moyenne rejoint la complexité dans le pire des cas.