Université Polytechnique de Bingerville

Cours
Algorithme avancé 1

Introduction aux structures de données

Dr MAMBE

2020 - 2021

Semestre 3 ASSRI - MIAGE

Organisation du cours

Volume horaire

15h de CM
15h de TD
10h de TP

Évaluation

2 exos de maison
1 dévoir commun
un examen final

PLAN

I/ Rappels sur l'algorithmique

2/ Structure de données

3/ Complexité algorithmique

Algorithme : Quelques rappels 1

- Qu'est-ce qu'un algorithme ?
- En quoi l'algorithme est-il important?
- L'impact d'une solution algorithmique

Algorithme : Quelques rappels 2

- **Affectation** (ex. mois <-- 6, jours[1] <-- 31)
- **Condition/Comparaison** (ex. mois <= 12)
- **Appel de fonction** (ex. lire(mois))
- **Structure de contrôle**
  - Branchements conditionnels (multiples) (si .. Alors .. Sinon)
  - Boucles (tant que..faire, pour.. faire)
- **Bloc d'instructions** (début ... fin)

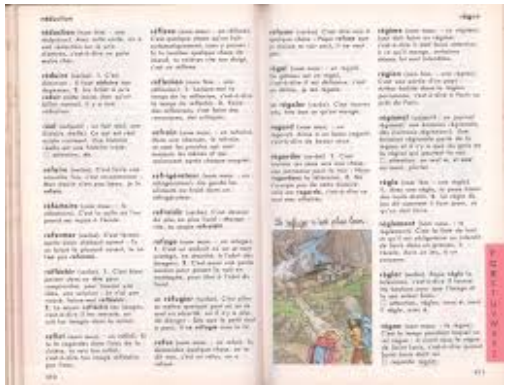
Structure de données

Définition

Une *structure de données* est un format spécialisé pour organiser, traiter, récupérer et stocker des données. ...

En programmation, *une structure de données* peut être sélectionnée ou conçue pour stocker des données dans le but de les travailler avec divers algorithmes.

Exemple : Un dictionnaire



Retenons

Algorithmes → méthodes

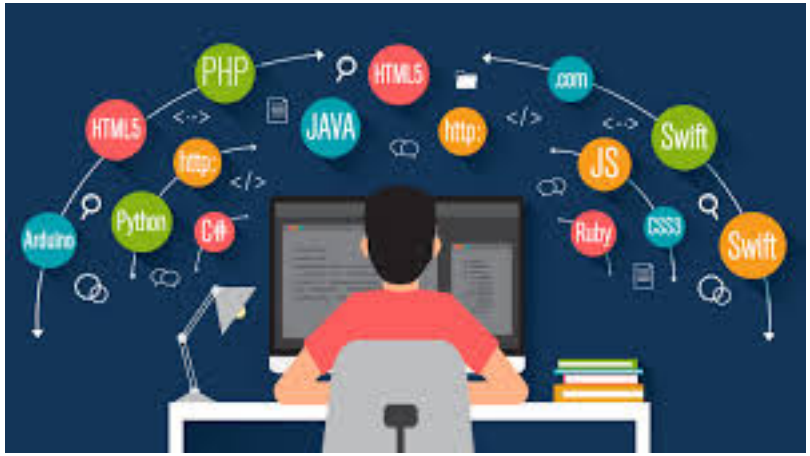
Structures de données → organisation

La complexité algorithmique

Notion de complexité algorithmique

Rédaction de l'Algo

Transcription en Langage



Correction de l'algorithme

1- Vérifier que son algorithme est correct

2- Evaluer l'efficacité de l'algorithme

Ce n'est pas Implémenter et exécuter

Notion de complexité algorithmique

Le terme de 'complexité' est un peu trompeur parce qu'on ne parle pas d'une difficulté de compréhension, mais d'**efficacité** : "complexe" ne veut pas dire "compliqué".

Un algorithme de forte complexité a un comportement asymptotique moins efficace qu'un algorithme de faible complexité, il est donc généralement plus lent.

Notion de complexité algorithmique

Considérons les deux algorithmes suivants, dépendant de N:

```
1 faire N fois l'opération A
2 faire N fois (l'opération B puis l'opération C)
```

Dans le premier cas, on fait **N fois l'opération A**, et dans le deuxième cas on fait au total **N fois l'opération B**, et **N fois l'opération C**.

En admettant que ces deux algorithmes *résolvent le même problème* (donc sont corrects), et que *toutes les opérations sont prises en compte pour la mesure de la complexité*, **le premier algorithme fait N** opérations et **le deuxième 2N.**

Notation "grand O"

La complexité est une approximation. Pour la représenter, on utilise une notation spécifique, la notation **O** appelée aussi notation de Landau.

Dans le premier cas, on fait **N fois l'opération A**, et dans le deuxième cas on fait au total **N fois l'opération B**, et **N fois l'opération C**.

En admettant que ces deux algorithmes *résolvent le même problème* (donc sont corrects), et que *toutes les opérations sont prises en compte pour la mesure de la complexité*, **le premier algorithme fait N** opérations et **le deuxième 2N.**

Calcul de la complexité

L'efficacité d'un algorithme est liée à la bonne utilisation des ressources temps de traitement et espace mémoire pour résoudre un problème donné.

On mesure l'efficacité d'un algorithme par son coût qui est lié donc à deux paramètres essentiels :

1- **Le temps d'exécution** appelé **"Complexité temporelle"** : il est liée au nombre d'opérations effectuées par l'algorithme ;

2- **L'espace mémoire** requis appelé **"Complexité spatiale"** : il correspond à l'encombrement mémoire.

Calcul de la complexité

Problème : calculer x^n

<u>données : x : réel, n: entier</u>

Méthode 1 : $x^0 = 1$; $x^i = x * x^{i-1}$ avec i >0 → complexité O(n)

Méthode 2 : $x^0 = 1$;

$x^i = x^{i/2} * x^{i/2}$, si i est pair;

$x^i = x*x^{i/2} * x^{i/2}$ si i est impair → complexité O(log n)

…

<u>résultats</u> : $y = x^n$

Quelle méthode choisir? et pourquoi?

Calcul de la complexité

Complexité temporelle

En pratique, le temps d'exécution dépend de la taille et de la valeur des données.

On définit trois (03) sortes de complexité temporelle :

- **Complexité maximale :** c'est le temps d'exécution d'un algorithme dans le cas le plus défavorable (temps le plus long).

- **Complexité moyenne :** c'est le temps moyen d'exécution de l'algorithme appliqué à n données quelconques équiprobables (même probabilité). on calcule le coût pour chaque donnée possible puis on divise la somme de ces coûts par le nombre de données différentes.

- **Complexité dans le meilleur des cas :** on calcule le coût en se plaçant dans le meilleur des cas (temps le plus court).

Calculer la complexité temporelle ???

Pour analyser un code de programmation ou un algorithme, il convient de noter que chaque instruction affecte les performances globales de l'algorithme. Par conséquent, chaque instruction doit être analysée séparément pour analyser les performances globales.

Méthode de comptage

Il s'agit de compter les opérations élémentaires :

1- Affectation

2- Addition, soustraction, division

3- Comparaison

Supposons que notre algorithme se compose de deux parties A et B. A prend le temps t_A et B prend le temps t_B pour le calcul. Le calcul total "$t_A + t_B$" est conforme à la règle maximale, donc le temps de calcul est ($max(t_A, t_B)$).

Calculer la complexité temporelle ???

On compte toutes les opérations élémentaires comme O(1)

Exemple :

```
a = 2+3
b = 2
c = a+b
```

La complexité est O(1)

Calculer la complexité temporelle ???

Structure conditionnelle SI

```
1  if(condition){
2      // bloc A
3  }
4  else{
5      // bloc B
6  }
```

Supposons que le bloc A prenne le temps t_A et le bloc B prenne le temps t_B, alors selon la règle maximale, ce temps de calcul est $max(t_A,t_B)$.

Supposons que $t_A=n^2$ et $t_B=3n+2$, alors le calcul total est :

$$Calcul\ total = max(t_A,t_B)$$
$$= max(O(n^2), O(3n+2)$$
$$= O(n^2)$$

Calculer la complexité temporelle ???

Boucle *Pour*

```
1  a=3+b
2  for(int i=0;i<n;i++){
3      printf("%d",i);
4  }
```

Ligne 1 ----> O(1)

Ligne 2 ----> ???

- Initialisation : une seule fois

- A chaque itération, on incrémente "i" de 1, l'incrémentation est donc faite n fois

- Chaque fois que nous incrémentons i, nous vérifions si la nouvelle valeur devient égale ou supérieure à n, nous effectuons donc la comparaison n + 1 fois d'où ***Ligne 2 ----> max(1,n+1,n) = n+1***

Complexité du programme : T(n)=2n+2=O(n)

Calculer la complexité temporelle ???

Boucle *Pour*

si la boucle est incrémentée ou décrémentée d'une valeur constante, la complexité est d'ordre ***O(n)***.

```
1  for(int i=0;i<n;i+=c){                          // O(n+1)
2      // quelques expressions d'ordre O(1)         // O(n)
3
4  }
```

$$T(n) = \sum_{i=0}^{n-1} O(1) = O(n)$$

Calculer la complexité temporelle ???

Boucle *Pour*

*Une boucle ou une récursion qui s'exécute un nombre de fois constant est considérée comme un . Par exemple, la boucle suivante est **O(1)**.*

```
1  // Ici c est une constante
2  for(int i=0;i<c;i++){
3      // quelques expressions d'ordre O(1)
4  }
5
6  // exemple :
7  for(int i=0;i<50;i++){
8      // quelques expressions d'ordre O(1)
9  }
```

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

Combien de comparaison dans la boucle suivante???

```
for(int i=0;i<n;i++){                          //
    for(j=0;j<m;j++){                          //
        //quelques expressions d'ordre O(1)   //
    }
}
```

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  for(int i=0;i<n;i++){
2      for(j=0;j<i;j++){
3          printf("hello");
4      }
5  }
```

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  for(int i=0;i<n;i++){
2      for(j=0;j<i;j++){
3          printf("hello");
4      }
5  }
```

i	j	nombre
0	0	0
1	0	1
2	0 1	2
...	...	...
n-1	0 1 2 ... n-2	n-1

$$
\begin{aligned}
T(n) &= 0 + 1 + 2 + 3 + \ldots + n - 1 \\
&= \sum_{i=0}^{n-1} i \\
&= \frac{n * n}{2} \\
&= O(n^2)
\end{aligned}
$$

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  p=0;
2  for(int i=0;p<n;i++){
3      printf("%d",i);
4      p=p+i;
5  }
```

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  p=0;
2  for(int i=0;p<n;i++){
3      printf("%d",i);
4      p=p+i;
5  }
```

i	p
1	0+1=1
2	1+2=3
3	1+2+3
4	1+2+3+4
...	...
k	1+2+3+...+k

la boucle s'arrête lorsque p devient supérieur à n. donc on suppose que p>n:

$$
\begin{aligned}
p &= 1+2+3+4+5+\ldots+k \\
&= \frac{k*(k+1)}{2} > n \\
&\Rightarrow k^2 > n \\
&\Rightarrow k > \sqrt{n}
\end{aligned}
$$

$$T(n) = O(\sqrt{n})$$

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  p=0;
2  for(int i=0;i<n;i*=2){
3      printf("%d",i);
4  }
```

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  p=0;
2  for(int i=0;i<n;i*=2){
3      printf("%d",i);
4  }
```

la boucle s'arrête lorsque i devient supérieur ou égale à n. donc on suppose que i ≥ n

Puisque $i = 2^k$

$$2^k \geqslant n \quad \Rightarrow \quad 2^k = n \quad \Rightarrow k = log_2 n$$

Donc $T(n) = O(log_2 n)$

Itération	i	
0	1	2^0
1	2	2^1
2	4	2^2
3	8	2^3
...	...	...
k	2^k	

Calculer la complexité temporelle ???

Boucle *Pour*

Retenons

La complexité temporelle d'une boucle est considérée comme ***O(log n)*** si la variable de boucle est divisée / multipliée par une valeur constante.

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  for(int i=0;i<n;i++){
2      printf("%d",i);
3  }
4  for(int j=0;j<n;j++){
5      printf("%d",i);
6  }
```

La première boucle est exécutée n fois
La deuxième boucle est exécutée n fois

Donc

$$T(n) = n + n = 2n = O(n)$$

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  p=0
2  for(int i=0;i<n;i=i*2){
3      p++;
4  }
5  for(int j=0;j<p;j=j*2){
6      printf("%d",i);
7  }
```

Calculer la complexité temporelle ???

Boucle *Pour*

Exemple :

```
1  p=0
2  for(int i=0;i<n;i=i*2){
3      p++;
4  }
5  for(int j=0;j<p;j=j*2){
6      printf("%d",i);
7  }
```

La complexité de la première boucle est $log_2 n$

La complexité de la deuxième boucle est $log_2 p$

Puisque $p = log_2\, n$

Donc $T(n) = O(log(log\, n))$

Calculer la complexité temporelle ???

Boucle *Pour*

Application

```
1  for(int i=1;i<n;i++){
2      for(j=1;j< n; j=j*2){
3          // instructions
4      }
5  }
```

Calculer la complexité temporelle ???

Boucle *Pour*

Conclusion

En général, si vous avez une boucle for, vous pouvez appliquer l'une des formules ci-dessous.

```
1 | for(i=0 ; i < n ; i+=c)
2 | for(i=n ; i > 0; i-=c)
```

$\Rightarrow O(n)$

```
1 | for(i=0 ; i < n ; i*=c)
2 | for(i=n ; i> 1 ; i/=c)
```

$\Rightarrow O(log_c\, n)$

Calculer la complexité temporelle ???

Boucle *Tant que*

*Pour analyser les boucles "tant que", on peut utiliser la même procédure que dans la boucle "**pour**"*

```
1  i=0;
2  while(i<n){
3      i++;
4  }
```

$$T(n) = O(n)$$

Calculer la complexité temporelle ???

Boucle *Tant que*

*Pour analyser les boucles "tant que", on peut utiliser la même procédure que dans la boucle "**pour**"*

```
1  a=1;
2  while(a<b){
3      a = a*2;
4  }
```

$$T(n) = O(log_2\, n)$$

Classes de complexité

O(1)		Complexité constante
O(log n)		Complexité logarithmique
O(n)		Complexité linéaire
O(n log n)		Complexité quasi-linéaire
$O(n^a)$	$O(n^a)$	Complexité polynomiale
	$O(n^2)$	Complexité quadratique
	$O(n^3)$	Complexité cubique
$O(a^n)$		Complexité exponentielle
O(n!)		Complexité factorielle

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^a) < O(n^2) < O(n^3) < O(a^n) < O(n!)$$