Introduction à l'analyse des algorithmes

Opération élémentaire

La complexité consiste à évaluer l'efficacité de d'une méthode M1 et de comparer cette dernière avec une autre méthode M2. Cette comparaison est indépendante de l'environnement (machine, système, compilateur, langage, etc.).

L'efficacité dépend du nombre d'opérations élémentaires. Ces dernières dépendent de la taille des données et de la nature des données.

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.

La complexité temporelle d'une fonction (ou d'un ensemble d'instructions) est considérée comme étant O(1) si elle ne contient ni boucle, ni récursivité, ni appel à aucune autre fonction temporelle non constante.

Les structures de contrôle communes dans chaque algorithme sont

- Structure conditionnelle IF-ELSE
- Boucle WHILE
- Boucle FOR

Supposons que notre algorithme se compose de deux parties A et B.

A prend le temps tA et B prend le temps tB pour le calcul. Le calcul total tA + tB est conforme à la règle maximale, donc le temps de calcul est max(tA, tB).

La complexité est une question de comptage. Pour comprendre comment analyser un algorithme, nous devons savoir compter le nombre de comparaisons, d'affectations, etc. En général, nous devons compter les opérations élémentaires.

La liste suivante montre les différentes opérations élémentaires

- Affectation
- Addition, soustraction, division
- Comparaison

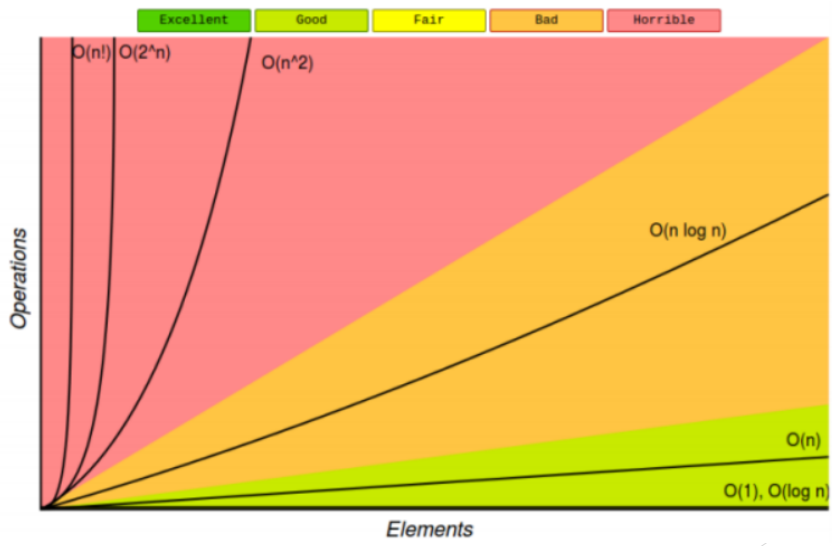
Comme compter devient difficile quand on a beaucoup d'opérations élémentaires, on compte toutes ces opérations élémentaires comme O(1)

Dans le cas d'une structure de type séquence, l'évaluation est égale à la somme des coûts. Par exemple, si l'algorithme possède un traitement T1(n) suivit de T2(n), alors T(n)=T1(n)+T2(n).

Dans le cas d'un embranchement, l'évaluation est égale au maximum des embranchements. Par exemple, l'algorithme exécute T1(n) sinon T2(n), alors T(n)=max(T1(n), T2(n)).

Dans le cas d'une boucle, l'évaluation est égale à la somme des coûts des passages successifs. Par exemple, si l'algorithme est un « tant que faire Ti(n) avec la complexité de Ti(n) dépendant du numéro i de l'itération. Alors la complexité est T(n) = somme(i=1 à n) Ti(n).

^ signifie puissance



Exemples introductifs

fonction permutation (S, i, j)

1	$tmp := S[i],$	coût c_1
2	$S[i] := S[j],$	coût c_2
3	$S[j] := tmp,$	coût c_3
4	**renvoyer** S	coût c_4

$$T(n) = c_1 + c_2 + c_3 + c_4 = O(1)$$

fonction recherche (x, S, n)

1 $i := 1,$
2 **tant que** $((i < n)$ **et** $(S[i] \neq x))$ **faire** (n fois)
3 $\quad i := i + 1,$
4 **renvoyer** $(S[i] = x)$

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

fonction Tri (S, n)

1 **pour** $i := n$ **à** 2 **faire** ($n - 1$ fois)
2 $\quad$ **pour** $j := 1$ **à** $i - 1$ **faire** ($i - 1$ fois)
3 $\quad\quad$ **si** $(S[j] > S[j + 1])$ **alors**
4 $\quad\quad\quad$ permuter $S[j]$ et $S[j + 1]$ dans S,

$$T(n) = C_{perm} \times \sum_{i=1}^{n-1} i = \frac{C_{perm} \times n \times (n-1)}{2} = O(n^2)$$

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

Exemple

Supposons que $tA = n^3$ et $tB = 2n + 1$, alors le calcul total est :

$$\begin{aligned} Calcul\ total &= max(tA, tB) \\ &= max(O(n^3), O(2n+1)) \\ &= O(n^3) \end{aligned}$$

Exemple 1 :

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

Si nous n'avons aucune boucle ou appel à une fonction contenant une boucle, la fonction de temps du programme ci-dessus est :

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

Exemple 2 :

```
for(int i=0;i<n;i++){
   for(j=0;j<i;j++){
      console.log("Salut");
   }
}
```

Pour comprendre clairement comment nous pouvons trouver la complexité du programme ci-dessus, nous allons utiliser un tableau pour tracer les itérations la première colonne contenant les valeurs de i, la deuxième colonne contenant les valeurs de j et la troisième colonne contenant le nombre de fois que Salut est affiché.

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

Donc le temps total est de

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

Exemple 3

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

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

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

Puisque $i = 2^k$

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

Donc

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

Exemple 4

```
for(int i=0;i<n;i++){
    console.log(i);
}
for(int j=0;j<n;j++){
    console.log(i);
}
```

Dans cet exemple les deux boucles étant indépendantes, la complexité de ce code est égale à la somme de la complexité des deux boucles.

La première boucle est exécutée n fois

La deuxième boucle est exécutée n fois

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

Suite avec un tri par comparaison
Voir code fait en classe avec VSC