Calcul de la complexité d'un algorithme

Dispensé par :

Jean Serge Dimitri Ouattara

jean.ouattara@ujkz.bf

Version 2021-2022

Calcul de la complexité d'un algorithme

Calcul de la complexité d'un algorithme

Rappels

Notations

- **n** : taille des données du problème ;
- **T(n)** : nombre d'opérations élémentaires pour résoudre le problème.

Évaluation de la complexité

Évaluer la complexité c'est déterminer **T(n)** en fonction de **n** :

- en considérant comme opérations élémentaires : l'affectation, la comparaison, les calculs élémentaires, l'affichage et l'écriture ;
- et en négligeant les opérations de déclarations et de retour.

On déterminera la forme générale ou la classe de complexité.

Rappels

Rappel sur la notation de Landau

Dans la notation de Landau on cherche une constante c et un seuil n_0 à partir duquel on a $f(n) < g(n)$ pour établir que $f = O(g)$. Dans la pratique, on considère le nombre d'opérations $T(n)$ comme $f(n)$ et une classe de complexité connue $g(n)$ pour prouver que $T(n)$ est de même classe de complexité.

Exercice d'application

Démontrer, dans chaque cas, que $T(n)$ est dans la classe de complexité :

- $T(n) = \frac{2n}{3} + 5$ est en $O(n)$;
- $T(n) = n^3 + 2n^2 + n + 5$ est en $O(n^3)$.

Rappels

Pistes de démonstration

Pour faire la démonstration il faut déterminer c et n_0. Ainsi :

- $f(n) = \frac{2n}{3} + 5$ est en $O(n)$ parce qu'à partir de $n_0 > \frac{5*3}{3*c-2}$ $f(n) < n$. On peut choisir $c = 1$ et avoir $n_0 = 16$;
- $g(n) = n^3 + 2n^2 + n + 5$ est en $O(n^3)$ car pour $n_0 = 3$ et $c = 2$ $g(n) < n^3$.

Rappels

Du nombre d'opérations $T(n)$ à la classe de complexité $O(X)$

Pour passer de $T(n)$ à la notation de Landau on peut négliger :

- les multiplications par des constantes : $K * O(n) \Leftrightarrow O(n)$;
- les additions avec des constantes : $K + O(n) \Leftrightarrow O(n)$;
- les termes de faibles degrés : $O(n^3 + n^2 + n) \Leftrightarrow O(n^3)$

Passer de $T(n)$ à $O(X)$

Si $T(n) = n^7 + 2n^2 + 15n + 3$, par simplifications on a :

- sans multiplications par des constantes : $T(n) = O(n^7 + n^2 + n + 3)$
- sans additions avec des constantes : $T(n) = O(n^7 + n^2 + n)$
- sans termes de faibles degrés : $T(n) = O(n^7)$.

Calcul de la complexité d'un algorithme

Complexité des enchaînements séquentiels

```
Traitement T1
Traitement T2
...
Traitement Tk
```

Complexité d'une séquence de traitement T_i

$$T(n) = T_1(n) + T_2(n) + ... + T_k(n)$$

Complexité des enchaînements séquentiels

Déterminer la complexité de cette procédure

```
Procedure Seq(T: Tableau d'entiers, N:Entier)
Debut
//Initialiser(T,N) remplit T avec N entrées utilisateur
Initialiser(T,N);
//Afficher(T, N) affiche les N valeurs de T
Afficher(T,N);
Fin
```

Complexité des enchaînements séquentiels

Piste pour la détermination

On considère que :

- *Initialiser*(T, N) a comme complexité $T_1(n)$;
- *Afficher*(T, N) a comme complexité $T_2(n)$.

Donc $T(n) = T_1(n) + T_2(n)$

Complexité des enchaînements conditionnels

```
Si(Condition C1) alors
Traitement T1
Sinon
Traitement T2
Fin si
```

Complexité d'une conditionnelle

$$T(n) = max(T_1(n), T_2(n))$$

Complexité des enchaînements conditionnels

Déterminer la complexité de cet algorithme

```
Procedure Cond(T: Tableau d'entiers, N:Entier)
Declarations
ordonne:Booléen;
i:Entier;
Debut
Initialiser(T,N);
//VerifierOrdre(T,N) vérifie que T est ordonné croissant
ordonne=VerifierOrdre(T,N);
//Si T n'est pas ordonné Ordonner(T) l'ordonne
Si (ordonne=Faux) alors
Ordonner(T,N);
Fin Si
Fin
```

Complexité des enchaînements conditionnels

Piste pour la détermination

On considère que :

- *Initialiser*(T, N) a comme complexité $T_1(n)$;
- *VerifierOrdre*(T, N) a comme complexité $T_2(n)$;
- *Ordonner*(T, N) a comme complexité $T_3(n)$;
- Deux branches possibles :
  - $T_{b1}(n)$: si *ordonne* $=$ *Faux* on va dérouler *Initialiser*(T, N), *VerifierOrdre*(T, N) et *Ordonner*(T, N)
  - $T_{b2}(n)$: sinon on va dérouler *Initialiser*(T, N) et *VerifierOrdre*(T, N) seulement.

Donc $T(n) = max(T_{b1}(n), T_{b2}(n)) = T_{b1}(n) = T_1(n) + T_2(n) + T_3(n)$

Complexité des enchaînements itératifs

```
Tant que (Condition C) faire
Traitement Ti;
Fin Tant que
```

Complexité de k itérations de traitement *Ti*

$$T(n) = \sum_{i=1}^{k} Ti(n_i)$$

Complexité des enchaînements itératifs

Déterminer la complexité de cet algorithme

```
Procedure Histo(T: Tableau d'entiers, n:Entier)
Declarations
i:Entier;
Debut
Initialiser(T,n);
Pour i allant de 1 à n Faire
//AfficherEtoile(T[i]) affiche T[i] étoiles et va à la ligne
AfficherEtoile(T[i]);
Fin Pour
Fin
```

Complexité des enchaînements itératifs

Piste pour la détermination

On considère que :

- *Initialiser*(T, n) a comme complexité $T_1(n)$;
- *AfficherEtoile*$(T[i])$ a comme complexité $T_2(i)$.

Donc $T(n) = T_1(n) + \sum_{i=1}^{n} T_2(i)$

Un peu d'exercice

Exercice d'application

On considère un tableau d'entiers de taille n.

1. Écrire une fonction **Somme** qui prend en entrée le tableau et qui calcule la somme de ces éléments.
2. Quelle est la complexité de la fonction **Somme** ?
3. Chaque cellule contient un entier correspondant à l'âge d'un des n étudiants de la classe. Écrire une fonction **Compte** qui détermine le nombre de classes d'âges différentes dans le tableau sans utiliser un autre tableau.
4. Quelle est la complexité de **Compte** ?
5. Si on permet l'utilisation d'un autre tableau et qu'on fixe les âges entre entre 20 et 50, peut-on écrire **Compte** avec une moindre complexité ? Comment et quelle complexité alors ?

Calcul de la complexité d'un algorithme

Notions sur les fonctions récursives

Définition

Une fonction est récursive si elle se fait appel à elle même.

Caractéristiques

- Chaque appel récursif traite une taille de données plus petite ;
- Il y a deux phases dans l'exécution d'une fonction récursive :
  - Une phase de **descente** où chaque appel récursif fait un autre appel récursif jusqu'à ce qu'un des appels atteigne une **condition terminale**. La fonction doit alors retourner une valeur au lieu de faire un appel récursif.
  - La phase de descente est suivie d'une phase de **remontée** qui s'effectue jusqu'à ce que l'**appel initial** soit terminé, ce qui met fin à la récursion.

Exemple de fonction récursive : calcul de la factorielle

Fonction récursive de calcul de la factorielle

```
Fonction Factorielle(n:Entier):Entier
Début
Si (n=0) alors
Retourne 1;
Sinon
Retourne n*Factorielle(n-1);
Fin Si
Fin
```

- Tant que que $n \neq 0$ on est dans la phase descendante avec des appels *Factorielle(n-1)* ;
- Lorsque $n = 0$, on commence la phase de remontée par le remplacement de *Factorielle(n-1)* par une valeur ;

Exemple de fonction récursive : calcul de la factorielle

Fonction récursive de calcul de la factorielle

```
Fonction Factorielle(n:Entier):Entier
Début
Si (n=0) alors
Retourne 1;
Sinon
Retourne n*Factorielle(n-1);
Fin Si
Fin
```

- $T(n) = ce + oe + T(n-1)$ si $n \neq 1$;
- $T(0) = ce$;

D'où $T(n) = (n+1) * ce + n * oe = O(n)$.

Types de fonctions récursives

Récursion simple

Une fonction est *récursive simple* si elle contient un seul appel récursif dans son corps.

Exemple : fonction factorielle

$f(n) = n * f(n-1)$.

Récursion multiple

Si la fonction contient plus d'un appel récursif dans son corps, on dit qu'elle admet une récursivité multiple.

Exemple : fonction combinaison

$$C_n^p = \begin{cases} 1 \ si \ p = 0 \ ou \ n = p \\ C_{n-1}^p + C_{n-1}^{p-1} \ sinon \end{cases}$$

Types de fonctions récursives

Récursion mutuelle

Deux fonctions sont mutuellement récursives si leurs définitions dépendent l'une de l'autre.

Exemple : fonctions pair et impair

$$Pair(n) = \begin{cases} vrai\ si\ n = 0 \\ Impair(n-1)\ sinon \end{cases} \qquad Impair(n) = \begin{cases} faux\ si\ n = 0 \\ Pair(n-1)\ sinon \end{cases}$$

Types de fonctions récursives

Récursion imbriquée

Une fonction admet une récursion imbriquée si dans l'un de ses appels un paramètre est défini par un appel à elle même.

Exemple : fonction d'Ackermann

$$Ackermann(m,n) = \begin{cases} n+1 \ si \ m = 0 \\ Ackermann(m-1,1) \ si \ m > 0 \ et \ n = 0 \\ Ackermann(m-1, Ackermann(m,n-1)) \ sinon \end{cases}$$

Types de fonctions récursives

Récursivité terminale

- Un appel récursif est **terminal** :
  - s'il s'agit de la dernière instruction dans le corps de la fonction ;
  - et si sa valeur de retour ne fait pas partie d'une expression.
- Un fonction récursive est **récursive terminale** si tous ses appels récursifs sont **terminaux** ;
- Un fonction **récursive terminale** ne fait rien dans la phase de remontée.

Un peu d'exercice

Récursivité terminale et impact sur la terminaison

```
Procedure GaucheVersDroite(T:Tableau d'entiers, N:entier, i:entier)
Si (i<=N) Alors
Ecrire(Tab[i]) ;
GaucheVersDroite (Tab,N,i+1) ;
Fin Si
Fin

Procédure DroiteVersGauche(T:Tableau d'entiers, N:entier, i:entier)
Si (i<=N) Alors
DroiteVersGauche (Tab,N,i+1);
Ecrire(Tab[i]) ;
Fin Si
Fin
```

Dérouler pour $T = 1,3,5,7,9$, $N = 5$ et $i = 1$

Élimination de la récursivité

Première approche

- Une fonction récursive peut avoir un équivalent non récursif ;
- Une fonction **récursive terminale simple** peut être remplacée par une fonction **itérative**.

Exemple

```
Fonction Recursion(X)
Si (Condition C) Alors
Traitement T1
Recursion(SousEnsemble(X))
Sinon Traitemant T2
Fin Si
Fin
```

```
Fonction Iteration(X)
Tant que (Condition C) faire
Traitement T1
X <-- SousEnsemble(X)
Fin tant que
Traitemant T2
Fin
```

Élimination de la récursivité

```
Fonction Recursion(X)
Si (Condition C) Alors
Traitement T1
Recursion(SousEnsemble(X))
Sinon Traitemant T2
Fin Si
Fin
```

```
Fonction Iteration(X)
Tant que (Condition C) faire
Traitement T1
X <-- SousEnsemble(X)
Fin tant que
Traitemant T2
Fin
```

Exercice d'application

Proposer une approche itérative pour la fonction Diviseur :

```
Fonction Diviseur(a,b) : Booléen
Si (a <=0) alors Retourner(Faux)
Sinon Si (a>=b) Retourner (a=b)
Sinon Retourner (Diviseur (a,b-a))
Fin Si
Fin Si
Fin
```

Élimination de la récursivité

Seconde approche

Si la fonction est **récursive non terminale** elle peut être remplacée par une fonction **itérative** utilisant une pile pour sauvegarder les données des différents appels récursifs.

Exemple

```
Fonction Recursion(X)
Si (Condition c) alors
Traitement 1;
Recursion(SousEnsemble(X));
Traitement 2 ;
Sinon
Traitement 3
Fin Si
Fin
```

```
Fonction Iteration(X)
P:Pile;
Tant que (Condition c) Faire
Traitement 1;
P.empiler(X); X <-- SousEnsemble(X)
Fin Tant que
Traitement 3
Tant que (P.vide()=Faux) Faire
P.dépiler(X); Traitement 2;
Fin Tant que
Fin
```

Élimination de la récursivité

```
Fonction Recursion(X)
Si (Condition c) alors
Traitement 1;
Recursion(SousEnsemble(X));
Traitement 2 ;
Sinon
Traitement 3
Fin Si
Fin
```

```
Fonction Iteration(X)
P:Pile;
Tant que (Condition c) Faire
Traitement 1;
P.empiler(X); X <-- SousEnsemble(X)
Fin Tant que
Traitement 3
Tant que (P.vide()=Faux) Faire
P.dépiler(X); Traitement 2;
Fin Tant que
Fin
```

Exercice d'application

Proposer une approche itérative pour la procédure DroiteVersGauche :

```
Procédure DroiteVersGauche(T:Tableau d'entiers, N:entier, i:entier)
Si (i<=N) Alors
DroiteVersGauche (Tab,N,i+1); Ecrire(Tab[i]) ;
Fin Si
Fin
```

Récursion ou stratégie de *Diviser pour Régner*

Pourquoi la récursion ?

Plusieurs problèmes peuvent être résolus par un algorithme récursif en utilisant le paradigme du **Diviser pour Régner**.

Diviser pour régner

Ce paradigme parcourt 3 étapes à chaque appel récursif :

- **Diviser** le problème en sous-problèmes de taille plus petite ;
- **Régner** sur les sous-problèmes en les résolvant de façon récursive ou directe si la taille du sous-problème est élémentaire ;
- **Combiner** les solutions des sous-problèmes en une solution globale pour résoudre le problème initial.

Récursion ou stratégie de *Diviser pour Régner*

Diviser pour régner

- Si la taille du problème est suffisamment réduite i.e. $n \leq c$ où c est une constante, la résolution est en $O(1)$;
- Sinon, on divise le problème en a sous-problèmes chacun de taille $\frac{n}{b}$. Le temps d'exécution total se décompose alors en trois parties :
  - $D(n)$: le temps nécessaire à la division du problème en sous-problèmes ;
  - $aT(n/b)$: le temps de résolution des a sous-problèmes ;
  - $\tau(n)$: le temps nécessaire pour combiner les solutions aux sous-problèmes et déterminer la solution finale.

En maths ...

$$T(n) = \begin{cases} \Theta(1) \; si \; n \leq c \\ aT(\frac{n}{b}) + D(n) + \tau(n) \; sinon \end{cases}$$

Récursion ou stratégie de *Diviser pour Régner*

Diviser pour régner

En posant $f(n) = D(n) + \tau(n)$ on a

$$T(n) = \begin{cases} \Theta(1) \ si \ n \leq c \\ aT(\frac{n}{b}) + f(n) \ sinon \end{cases}$$

équivalent de

$$T(n) = \begin{cases} \Theta(1) \ si \ n \leq c \\ aT(\frac{n}{b}) + cn^k \ sinon \end{cases}$$

Théorème de résolution de la récurrence

- si $a > b^k$ alors $T(n) = O(n^{log_b a})$
- si $a = b^k$ alors $T(n) = O(n^k log_b n)$
- si $a < b^k$ alors $T(n) = O(f(n)) = O(n^k)$

Récursion ou stratégie de *Diviser pour Régner*

Résolution de la récurrence linéaire

$$T(n) = aT(n-1) + f(n) \leftrightarrow T(n) = a^n(T(0) + \sum_{i=1}^{n} \frac{f(i)}{a^i})$$

Exemple

Déterminer la complexité de $T(n) = 2T(n-1) + 1$. Pour rappel

$$\Sigma_{i=0}^{n} \frac{1}{2^i} \rightarrow 2$$

Récursion ou stratégie de *Diviser pour Régner*

Résolution de la récurrence linéaire sans second membre $f(n)$

$$T(n) - a_1 T(n-1) - a_2 T(n-2) - \ldots - a_{n-k} T(n-k) = c$$
$$\Leftrightarrow T(n) = c_1 r_1^n + c_2 r_2^n + \ldots + c_k r_k^n$$

où les r_i sont des racines du polynôme
$x^k - a_1 x^{k-1} - a_2 x^{k-2} - \ldots - a_k$.

Résolution de la récurrence linéaire sans second membre $f(n)$

Appliquer à la fonction Fibonacci.
Pour rappel, la suite de Fibonacci est définie par

$$F_{n+1} = F_n + F_{n-1}$$

Complexités connues

Type	Solution	Exemple
$T(n) = T(n-1) + b$	$O(n)$	factorielle, recherche séquentielle récursive dans un tableau
$T(n) = a * T(n-1) + b, a \neq 1$	$O(a^n)$	a fois 1 traitement sur résultat de l'appel récursif
$T(n) = T(n-1) + a * n + b$	$O(n^2)$	traitement en coût linéaire avant appel récursif
$T(n) = T(n/2) + b$	$O(log(n))$	élimination de la moitié des éléments en temps constant avant appel récursif : dichotomique récursive
$T(n) = a * T(n/2) + b, a \neq 1$	$O(n^{log2(a)})$	a fois 1 traitement sur le résultat de l'appel récursif dichotomique
$T(n) = T(n/2) + a * n + b$	$O(n)$	traitement linéaire avant appel récursif dichotomique
$T(n) = 2 * T(n/2) + a * n + b$	$O(n * log(n))$	traitement linéaire avant double appel récursif dichotomique

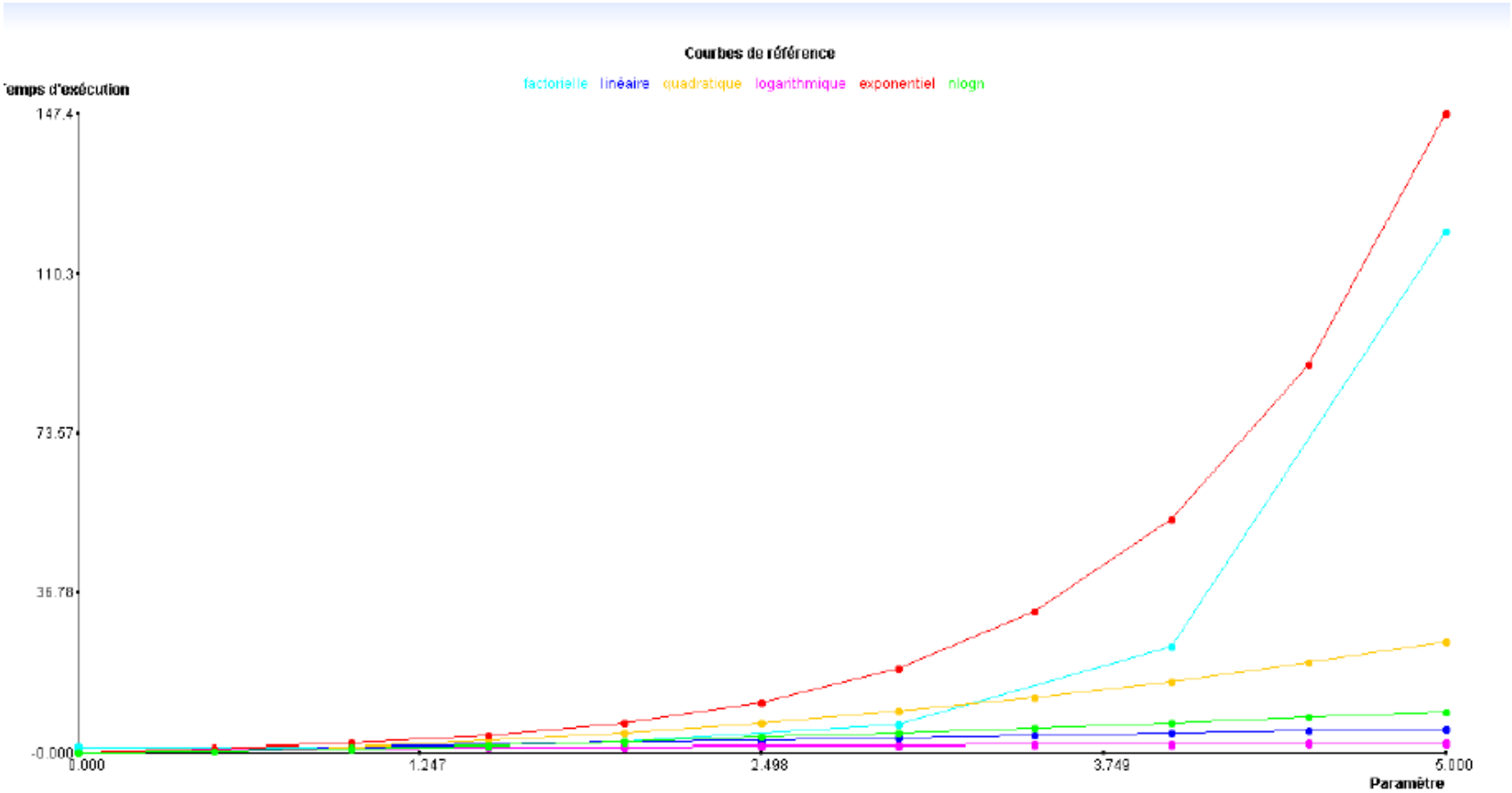
Complexités connues

TABLE – Ordre de grandeur pour n vers le million

FLOPS	$\log(n)$	n	n^2	2^n
10^6	$0,013ms$	$1s$	278heures	10000 ans
10^9	$0,013\mu s$	$1ms$	$\frac{1}{4}$ heure	10 ans
10^{12}	$0,013ns$	$1\mu s$	$1s$	1 semaine

Nos ordinateurs sont dans l'ordre de plusieurs gigaflops et les serveurs de calculs ont au moins des teraflops.

Complexités connues



Un peu d'exercice

Exercice d'application

Soit une suite définie par :

$$\begin{cases} u_1 = 1 \\ u_2 = 2 \\ u_n = u_{n-1} + u_{n-2} \forall n \geq 3 \end{cases}$$

1. Quelle est la complexité de la fonction récursive qui calcule u_n pour $n \geq 1$?
2. Proposez une fonction itérative équivalente à la fonction récursive.
3. Quelle est la complexité de la fonction itérative ?
4. Soit f la fonction itérative et g celle récursive. Déterminez si :
   1. $f = O(g)$;
   2. $f = \Omega(g)$;
   3. $f = \Theta(g)$;
5. Que peut-on en déduire ?