Analyse de Complexité

Dr. Mahfoud Houari
mahfoud.houari@gmail.com
Université Abou-Bakr Belkaïd - Tlemcen

2022/2023

Plan du cours :

0. Introduction générale
1. **Analyse de complexité**
2. Diviser pour régner
3. Programmation dynamique
4. Algorithmes gloutons
5. Les classes de problèmes *P*, *NP* et *NPC*
6. Les algorithmes d'approximation

Résumé du cours précédent :

1. Plusieurs catégories de problèmes : (in)décidables, faciles, difficiles, admettant des solutions exactes/approchées.
2. Concevoir un bon algorithme nécessite de passer par plusieurs étapes.
3. Notions d'***Efficacité***, d'***Optimalité***, et du ***Scalability***.

⇒ Comment mesurer l'efficacité d'un algorithme ?

Pourquoi mesurer la complexité d'un algorithme ?

1. Décider théoriquement si l'algorithme est efficace : pourra t-il répondre en consommant une ***quantité raisonnable de ressources*** ? (selon un certain contexte).
2. Étant donnés deux algorithmes répondant au même problème, ***lequel est le plus efficace*** ?
3. Pouvoir ***classer un problème*** à travers le ***coût*** du meilleur ***algorithme*** qui le ***résout***.

Mesurer la complexité à la base de quelle ressource ?

1. **Le temps :** ***Complexité temporelle***.
2. **La mémoire :** ***Complexité spatiale***.
3. **Autre :** nombre de processus, la bande passante,...

Quelle complexité doit-on considérer ?

Temporelle, spatiale ou un compromis entre les deux

(tout dépend du contexte)

Commençons par le commencement

Coût d'un Algorithme

Alors, c'est quoi le coût d'un algorithme ?

1. C'est le nombre total d'opérations élémentaires.

2. ***Pour simplifier***, toutes les opérations suivantes ont un coût égal à ***1*** :
   - Une instruction basique (**:=,+,-,***, **<,>,<=,** $\geq$ **, &&, | |, return,**...).
   - L'accès à la valeur d'un objet (ou élément d'un tableau).
   - La définition d'une variable (type simple ou référence).

Alors, c'est quoi le coût d'un algorithme ?

1. C'est le nombre total d'opérations élémentaires.

2. ***Pour simplifier***, toutes les opérations suivantes ont un coût égal à ***1*** :

- Une instruction basique (**:=,+,-,***, **<,>,<=,** ≥ **, &&, | |, return,**...).
- L'accès à la valeur d'un objet (ou élément d'un tableau).
- La définition d'une variable (type simple ou référence).

3. Le coût d'une boucle c'est le coût du bloc de la boucle multiplié par le nombre d'itérations de cette boucle.

Exemple :

```
int i=1;              2
while(i ≤ 5){         1.5 + 1
    i++ ;             2.5
}
```

Alors, c'est quoi le coût d'un algorithme ?

1. C'est le nombre total d'opérations élémentaires.

2. ***Pour simplifier***, toutes les opérations suivantes ont un coût égal à ***1*** :

- Une instruction basique (**:=,+,-,*, <,>,<=, ≥ , &&, | |, return,**...).
- L'accès à la valeur d'un objet (ou élément d'un tableau).
- La définition d'une variable (type simple ou référence).

3. Le coût d'une boucle c'est le coût du bloc de la boucle multiplié par le nombre d'itérations de cette boucle.

Exemple :

int i=1;	***2***
while(i ≤ 5 && 1==1)){	**3.5 + 1**
i++ ;	**2.5**
}	

Alors, c'est quoi le coût d'un algorithme ?

1. C'est le nombre total d'opérations élémentaires.

2. ***Pour simplifier***, toutes les opérations suivantes ont un coût égal à ***1*** :

- Une instruction basique (**:=,+,-,*, <,>,<=, ≥, &&, ||, return,**...).
- L'accès à la valeur d'un objet (ou élément d'un tableau).
- La définition d'une variable (type simple ou référence).

3. Le coût d'une boucle c'est le coût du bloc de la boucle multiplié par le nombre d'itérations de cette boucle.

4. Le coût d'un branchement conditionnel (***if else***) est celui du bloc le plus gourmand de ce branchement.

Exemple :

```
if(i < 5)                    1
    return 0 ;               1
if(i >= 5 && i < 10)         3
    retrun 1 ;               1
else
    i++ ;                    2
```

Au pire : **06 instructions**
Au meilleur : **02 instructions**

Alors, c'est quoi le coût d'un algorithme ?

1. C'est le nombre total d'opérations élémentaires.

2. ***Pour simplifier***, toutes les opérations suivantes ont un coût égal à ***1*** :

- Une instruction basique (**:=,+,-,***, **<,>,<=,** ≥ **, &&, ||, return,**...).
- L'accès à la valeur d'un objet (ou élément d'un tableau).
- La définition d'une variable (type simple ou référence).

3. Le coût d'une boucle c'est le coût du bloc de la boucle multiplié par le nombre d'itérations de cette boucle.

4. Le coût d'un branchement conditionnel (***if else***) est celui du bloc le plus gourmand de ce branchement.

5. Le coût d'un appel d'une fonction c'est le coût du corps de cette fonction.

6. Le coût peut être constant (indépendant de la taille des données).

7. ***Le coût doit être calculé en fonction des entrées.***

Mesurer le coût d'un algorithme – Exemple 1

Code 1 :

```
boolean R = (X == 3) ;                  ..............3
R = (Y.getVal() == 3) ;                 ..............3
R = (Y.getVal() == Z.getVal()) ;        ...4
```

Code 2 :

```
boolean listeVide (LC L){
    return (L.tete == null) ;
}
```

Coût = 3.
Coût constant

Mesurer le coût d'un algorithme – Exemple 2

Code 3 :

```
LC Mystere (LC L1, LC L2){
   Pointeur P = L1.tete ;                 .............3
   While (P.suivant != null) do ......2*(|L1|-1)  + 2
      { P = P.suivant ; }                 ............2*(|L1|-1)

   P.suivant = L2.tete ;                  ............3
   return L1 ;  ............................1
}
```

Coût : $4*(|L1|-1) + 9$.
Coût variable.

Différentes bornes pour calculer le coût

1. <u>**Pire des cas :**</u> ***Worst Case Complexity***

Considérer le cas du problème nécessitant le plus de ressources.

2. <u>**Meilleur des cas :**</u> ***Best Case Complexity***

Considérer le cas du problème nécessitant le moins de ressources.

3. <u>**En moyenne :**</u> ***Average Case Complexity***

Considérer tous les cas du problème et calculer leur coût selon leur probabilité d'apparition. Le coût total est la moyenne de tous ces coûts calculés.

« C'est une complexité probabiliste »

Différentes bornes pour calculer le coût – Exemple

Problème : chercher un élément E dans un tableau T de taille N sachant que le test $E \in T[i]$ prend *1 seconde*.

Au pire des cas : *N.1 secondes*

Au meilleur des cas : *1.1 seconde*.

En moyenne :

[*Temp*(*1^er^ Cas*).*Prob*(*1^er^ Cas*) + ….+ *Temp*(*N^ème^ Cas*).*Prob*(*N^ème^ Cas*)]

En supposant que les cas ont la même probabilité (**1/N**)

$$[Temp(E \in T[0]) + \ldots + Temp(E \in T[N-1])]/[N]$$

$$=$$

$$[1.s + 2.s + \ldots + N.s]/[N] = \frac{[N^2 + N]}{2.N} \text{ secondes}$$

Différentes bornes pour calculer le coût – Exemple

Problème : chercher un élément ***E*** dans un tableau ***T*** de taille ***N*** sachant que le test $E \in T[i]$ prend *1 seconde*.

Pour $N = 10$.

Au pire des cas : *10 secondes*

Au meilleur des cas : *01 seconde*

En moyenne : *5.5 secondes*

⇒ ***La complexité en moyenne des cas diminue l'erreur des deux autres bornes***

Ordres de grandeur asymptotique :

Pour faciliter la comparaison entre deux algorithmes on ne compare pas entre leurs coûts exacts mais plutôt entre leurs ordres de grandeur asymptotiques.

Exemple :

- **$n^3 + 2.n^2 + 15$ peut être bornée par n^3.**
- **Un algorithme ayant le coût $n^3 + 2.n^2 + 15$ et moins efficace par rapport à un autre ayant le le coût $n^2 + 3.n + 20$.**
- **Un algorithme ayant le coût $n^3 + 2.n^2 + 15$ et un autre ayant le le coût $n^3 + 20$ ont théoriquement la même complexité.**

Ordres de grandeur asymptotique :

- Soient donnés deux algorithmes ayant respectivement les coûts exacts $X^2 + 2.X + 15$ et $X^2 + 9.X + 10$.
- Ils ont le même ordre de grandeur X^2, vu que pratiquement leurs courbes respectives sont dominées par X^2.

Ordres de grandeur asymptotique – Notation O

Exemple :

$2/3\ n^2 + 1/2\ n \quad = \quad O(n^2)$

$2/3\ n^2 + 1/2\ n \quad = \quad O(n^2)$

$\log(n) \quad = \quad O(\log(n))$

$n + n.\log(n) \quad = \quad O(\mathrm{n}.\log(n))$

$n^2 + n.\log(n) \quad = \quad O(n^2)$

Ordres de grandeur asymptotique – Notation O

Exemple :

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

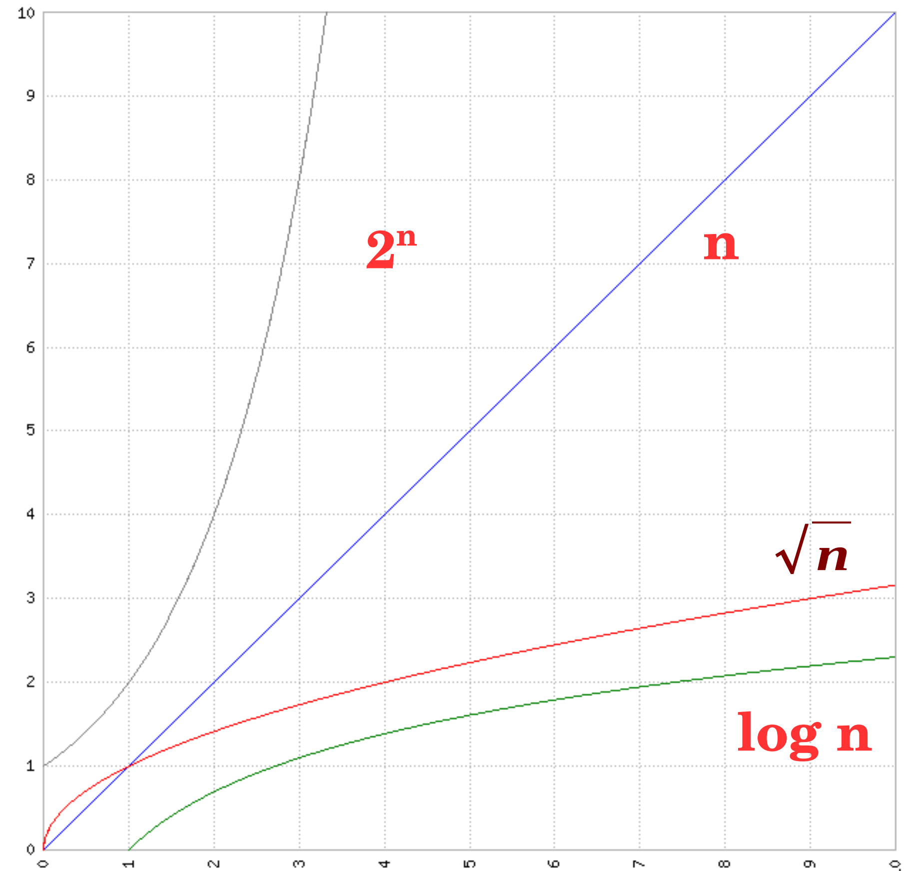
$$n + \log(n) = O(n)$$

$$n + n.\log(n) = O(n.\log(n))$$

$$n + n.\log(n) = O(n^2) \ (\textit{faux})$$

$$n^2 + n.\log(n) = O(n^2)$$

$$n.\sqrt{n} + n.\log(n) = O(n.\sqrt{n})$$



Classes de Complexité :

1. Constante : $O(1)$
2. Linéaire : $O(n)$
3. Logarithmique : $O(log\ n)$
4. Quasi-linéaire : $O(n.log\ (n))$
5. Quadratique : $O(n^2)$
6. Cubique : $O(n^3)$
7. Polynomiale : $O(n^k)$
8. Exponentielle : $O(k^n)$ pour $k > 1$.
9. Quasi-exponentielle : $O(n^{log\ n})$
10. Factorielle : $O(n!)$

Remarques :

- Un algorithme est ***praticable*** si sa complexité est polynomiale.
- Un algorithme est ***optimal*** s'il n'y a pas un autre algorithme qui a une complexité pire cas plus petite.
- Un problème est classé linéaire si tous les algorithmes le résolvant ont une complexité linéaire (pareil pour les autres classes de complexité).
- Une complexité exponentielle dépasse toute autre complexité polynomiale pour des données assez grandes.
- La complexité temporelle dépend généralement de la complexité spatiale.
- Pour un problème qui **n'admet pas un algorithme praticable**, on s'intéresse à trouver une **solution approximative polynomiale**.

Attention : assurer une complexité polynomiale peut être insuffisant dans un contexte de **BIG DATA**.

Remarques :

Temps d'exécution de neuf algorithmes en supposant que chaque instruction prend 0,1 nanoseconde (processeur ayant plus de 6 GHZ).

Complexité \ Taille	20	50	100
$10^7 \cdot n$	0.02 s	0.05 s	0.1 s
$10^7\ n \log_2 n$	0.09 s	0.3 s	0.7 s
$10^6\ n^2$	0.04 s	0.25 s	1 s
$10^5\ n^3$	0.08 s	1.25 s	10 s
$n^{\log_2 n}$	0.04 ms	0.4 s	32 min
$2^{n/3}$	10 ns	0.1 s	1 s
2^n	100 µs	31 h	--
3^n	0.3 s	$2.\ 10^7$ ans	--
$n!$	7.7 ans	--	--

Exemples d'analyse de complexité (1) :

public boolean *Existe*(int *E*, int [] *A*) {	
int *i* = *0* ;	2
while (*i* < *A*.*length*){	2.|*A*| + 2
if (*E* == *A*[*i*])	2.|*A*|
return *true* ;	1
i++ ;	2.|*A*|
}	
return *false* ;	1
}	

WCTC : 5 + 6.|*A*| = O(|*A*|).

BCTC : 5 = O(*1*).

Exemples d'analyse de complexité (2) :

```
//I commençe par 0
public boolean Existe(int E, int [] A, int I) {
    if ( I ≥ A.length)                          2
        return false ;                          1
    else if ( E == A[ I ] )                     2
            return true ;                       1
    else
        return Existe(E, A, I + 1) ;            2 + Coût_App_Rec
}
```

WCTC : 6.(*nombre des appels récursifs*)+3 = 6.|*A*|+3 = O(|*A*|).

BCTC : 3 = O(*1*).

Exemples d'analyse de complexité (4) :

La méthode Java suivante permet de tester si un élément ***E*** appartient à un arbre binaire composé de ***N*** nœuds.

```
public boolean existe(Arbre A, int E) {
    if (A == null)                                              1
        return false ;                                          1
    else if (A.valeur() == E)                                   2
        return true ;                                           1
    else
        return existe(A.fg(), E) || existe(A.fd(), E) ; 4 + C_fg + C_fd
}
```

BCTC : 2 = O(*1*).

WCTC : 7 + C_{fg} + C_{fd} = ????

Exemples d'analyse de complexité (4) :

La méthode Java suivante permet de tester si un élément ***E*** appartient à un arbre binaire composé de ***N*** nœuds.

```
public boolean existe(Arbre A, int E) {
    if (A == null)                                              1
        return false ;                                          1
    else if (A.valeur() == E)                                   2
        return true ;                                           1
    else
        return existe(A.fg(), E) || existe(A.fd(), E) ; 4 + C_fg + C_fd
}
```

- **BCTC : 2 = O(*1*).**
- **WCTC : 7 + C_{fg} + C_{fd} = 7.*N* = O(*N*).**

Exemples d'analyse de complexité (5) :

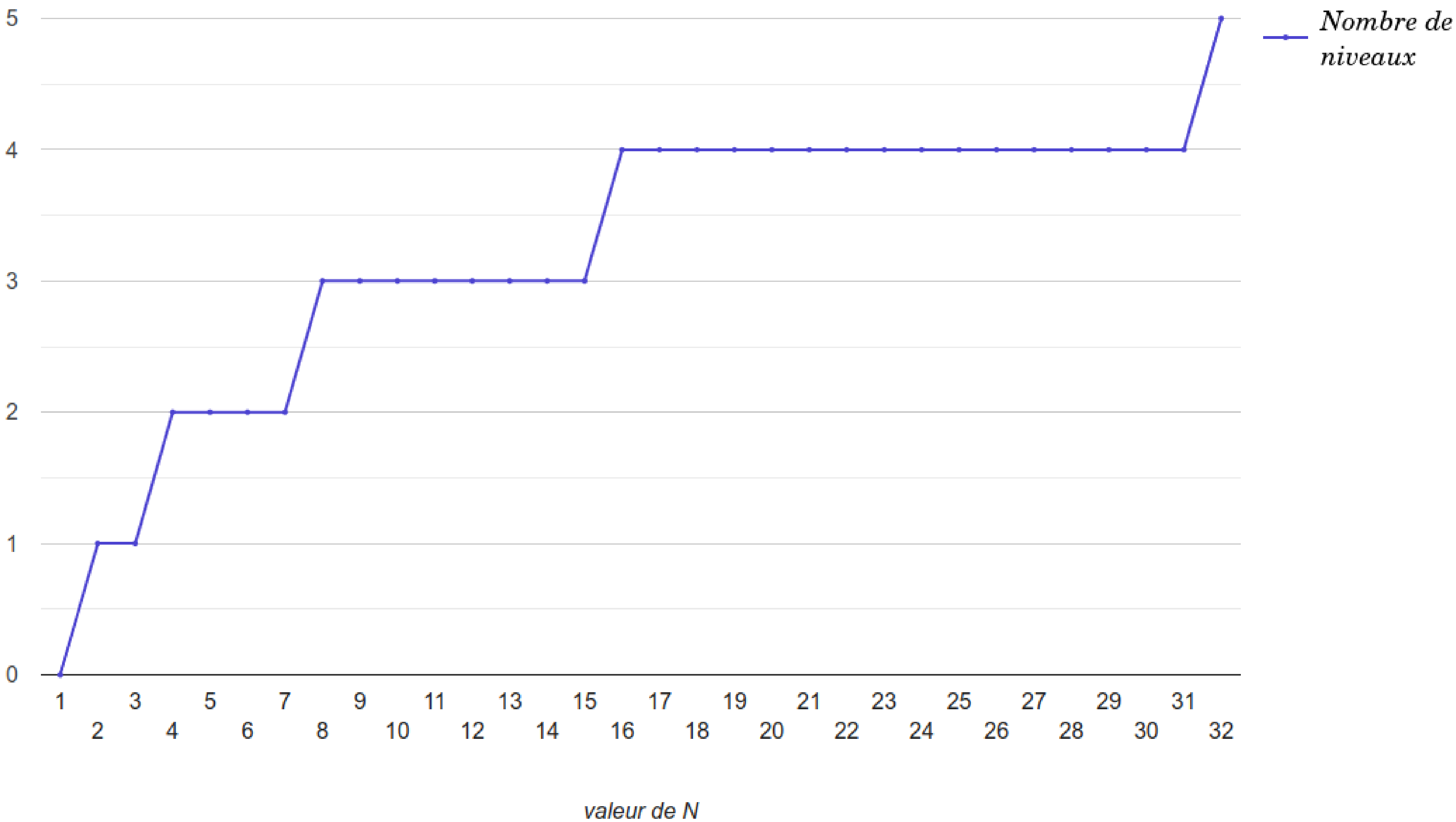
```
public int puissance(int X, int N) :
    if (X == 0)                              1
        return 1 ;                           1
    else if (N == 1)                         1
        return X ;                           1
    else {
        int demi = puissance(X, N / 2) ;     3 + C_N/2
        if (N % 2 == 0)                      2
                return demi * demi ;         2
        else
                return demi * demi * X ;     3
    }
}
```

BCTC : 2 = O(*1*)

WCTC : $10+C_{N/2} = 10+10+C_{N/4} = 10 *$ *nombre des appels récursifs*

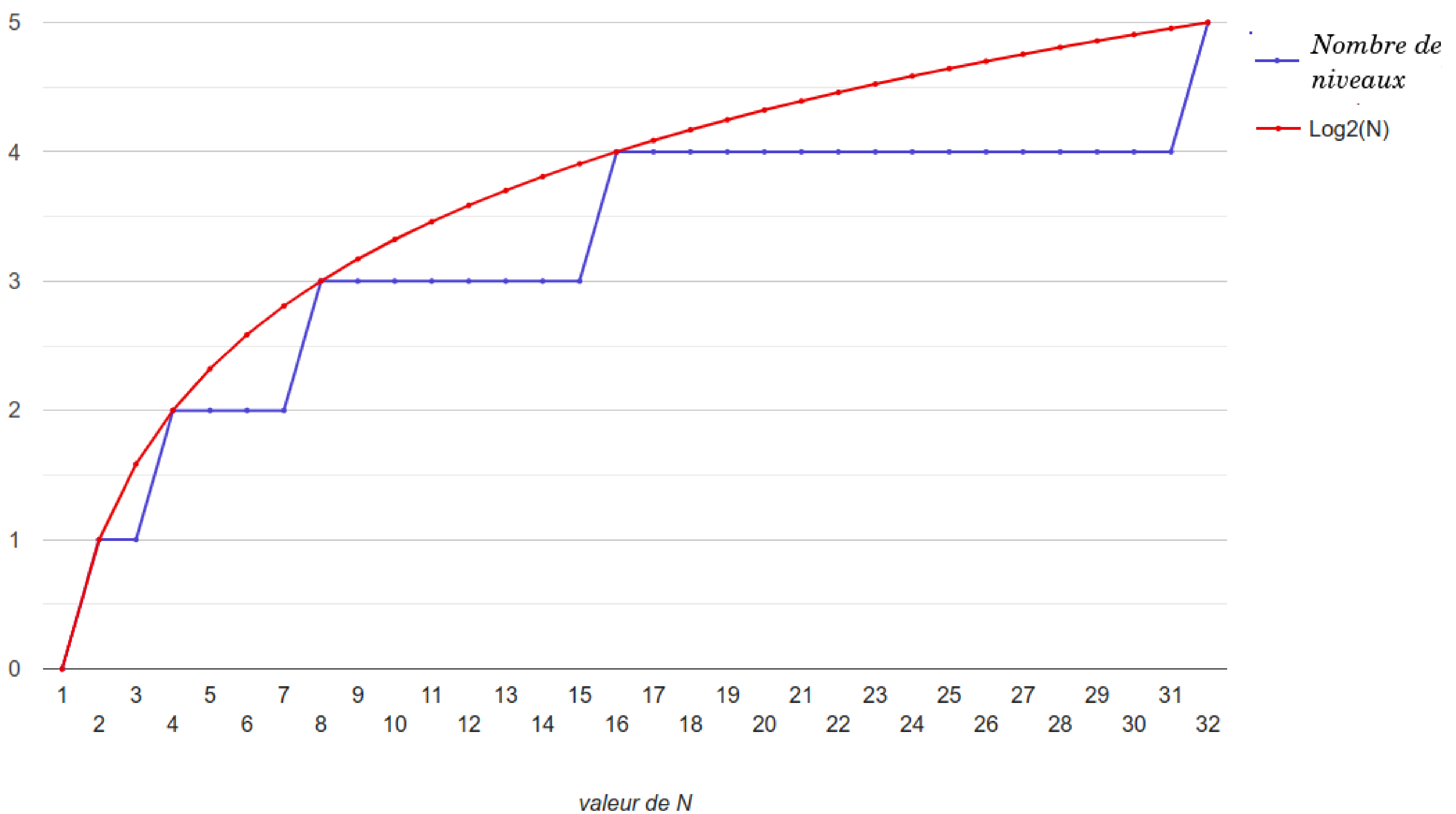
Exemples d'analyse de complexité (5) :

Nombre des appels récursifs de la méthode ***puissance*** :



Exemples d'analyse de complexité (5) :

Nombre des appels récursifs de la méthode ***puissance*** :



Exemples d'analyse de complexité (5) :

public int *puissance*(int *X*, int *N*) :	
if (*X* == 0)	1
return *1* ;	1
else if (*N* == *1*)	1
return *X* ;	1
else {	
int *demi* = *puissance*(*X*, *N* / *2*) ;	$3 + C_{N/2}$
if (*N* % 2 == 0)	2
return *demi* * *demi* ;	2
else	
return *demi* * *demi* * *X* ;	3
}	
}	

- BCTC : $2 = O(1)$
- WCTC : $10.\log_2(N) = O(\log_2(N))$