Informatique Tronc Commun

Rappels : Preuves des algorithmes

Programme officiel

Notions	Commentaires
Algorithmes dichotomiques.	On met en évidence une accélération entre complexité linéaire d'un algorithme naïf et complexité logarithmique d'un algorithme dichotomique.
Terminaison. Correction partielle. Correction totale. Variant. Invariant.	La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine. On montre sur plusieurs exemples que la terminaison peut se démontrer à l'aide d'un variant de boucle. Sur plusieurs exemples, on explicite, sans insister sur aucun formalisme, des invariants de boucles en vue de montrer la correction des algorithmes.
Complexité.	On aborde la notion de complexité temporelle dans le pire cas en ordre de grandeur. On peut, sur des exemples, aborder la notion de complexité en espace.

Plan du cours

I Complexité

I.1 Définitions

L'exécution d'un programme utilise les ressources de l'ordinateur, que l'on peut caractériser par :

— le temps de calcul pour exécuter les opérations,

— la place mémoire nécessaire pour stocker les données et le programme en cours d'exécution.

Définitions : Complexités

On définit plusieurs types de complexités :

— **complexité en temps** : évaluation du nombre d'opérations nécessaires pour effectuer le calcul.

— **complexité en mémoire** : évaluation de l'espace mémoire nécessaire pour effectuer le calcul.

— On calculera souvent la **complexité dans le pire des cas et dans le meilleur des cas**.

La majorité du temps, on vous demandera de déterminer une complexité en temps (ce qui sera souvent sous entendu)

Pour évaluer le temps d'exécution d'un calcul, il faut évaluer le temps nécessaire pour réaliser chaque instruction et les sommer. La durée d'exécution d'un problème dépend de taille des instances d'entrée, de l'algorithme utilisé, de l'ordinateur utilisé, des autres processus qui tournent sur l'ordinateur en même temps, ...

On se contentera de compter le **nombre d'opérations élémentaires**.

I.2 Ordres de grandeurs

On exprime la complexité d'un calcul comme une fonction des données nécessaires pour décrire le problème à l'ordinateur. Par exemple, si l'entrée est une liste, on exprimera la complexité en fonction du nombre d'éléments de la liste.

Les complexités seront données en utilisant la notation de Landau O (« grand O ») qui sert à comparer le comportement de deux suites (cf cours de maths).

Exemples : $5n + 7 = O(n)$; $3n^2 + 4n + 2 = O(n^2)$

On dira par exemple que « la complexité est en $O(n^2)$ » quand la complexité croit aussi vite que n^2.

On rencontrera des algorithmes de complexité :

— **constante**, quand elle est en $O(1)$, c'est-à-dire indépendante de la taille de l'entrée ;

— **logarithmique**, quand elle est en $O(\log_2(n))$;

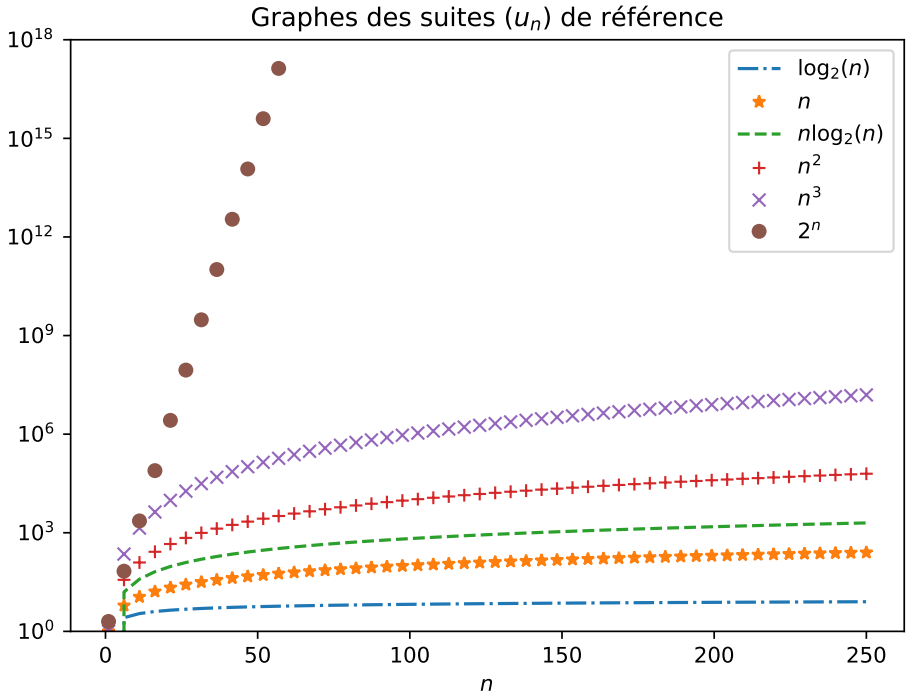
— **linéaire**, quand elle est en $O(n)$;

— **quasi-linéaire**, quand elle est en $O(n\log_2(n))$;

— **quadratique**, quand elle est en $O(n^2)$;

— **exponentielle**, quand elle est en $O(\rho^n)$ pour un $\rho > 1$.

n	2	2^4	2^6	2^8
$\log_2(n)$	1	4	6	8
n	2	16	64	256
$n\log_2(n)$	2	64	384	2048
n^2	4	256	4096	65536
2^n	4	65536	$1,84 \times 10^{19}$	$1,15 \times 10^{77}$



I.3 Quelques exemples

Exercice 1 Recherche d'un élément dans une liste

On donne le programme Python de recherche d'un élément dans une liste :

```
def recherche(L,x):
    for i in range(len(L)):
        if x==L[i]:
            return True
    return False
```

R1. Quel est le pire des cas en terme de nombres d'opérations ?

Solution: Le pire des cas est le cas où l'élément n'est pas dans la liste ou à la dernière position, car cela nécessite le parcours de l'ensemble de la liste.

R2. Déterminer dans ce cas le nombre d'opérations. En déduire la complexité temporelle.

Solution: En notant n le nombre d'éléments dans la liste, il y a n itérations. Pour chaque itération il y a un test.
Par conséquent, il y a n opérations. La complexité est donc en $O(n)$.

On propose l'algorithme de recherche d'un élément dans une liste triée, par dichotomie :

```
def dichotomie(L,x):
    a=0
    b=len(L)-1
    while b-a>=0:
        c=(a+b)//2
        if x==L[c]:
            return True
        elif x<L[c]:
            b=c
        else:
            a=c
    return False
```

R3. Déterminer la complexité temporelle. Commenter.

Solution: À chaque étape, on recherche x dans la moitié de la liste de la taille précédente. Soit n la longueur de la liste.
À la fin, après m étapes, dans le pire des cas (il a fallu arriver à la fin : $b = a$) il reste une liste d'un élément : $1 = \frac{n}{2^m}$.
Il y a donc au plus $m = \log_2(n)$ opérations.
La complexité est donc logarithmique $O(\log_2(n))$.
La complexité est meilleure que l'algorithme naïf, mais nécessite une liste triée. Le tri d'une liste étant dans le meilleur des cas (tri rapide ou tri fusion) en $O(n\log(n))$.

Exercice 2 Somme de factorielles

On souhaite écrire un programme renvoyant la somme des n premières factorielles. On propose d'écrire ce programme en deux fonctions :

```
def factorielle(n):
    F=1
    for i in range(1,n+1):
        F=F*i
    return F
def somme_fact(n):
    S=0
    for j in range(0,n+1):
        S=S+factorielle(j)
    return S
```

R1. Déterminer le nombre d'opérations élémentaires effectuées dans la fonction `factorielle`. En déduire la complexité temporelle.

Solution:

```
def factorielle(n):
    F=1 #1 affectation
    for i in range(1,n+1): #n itérations
        F=F*i #1 produit et 1 affectation
    return F
```

Ainsi il y a $1+n\times 2$ opérations élémentaires, donc la complexité temporelle de la fonction `factorielle` est en $O(n)$.

R2. Déterminer le nombre d'opérations élémentaires effectuées dans la fonction `somme_fact`. En déduire la complexité temporelle.

Solution:

```
def somme_fact(n):
    S=0 #1 affectation
    for j in range(0,n+1):  # n itérations
        S=S+factorielle(j) # 1 somme + 1 affectat° + j op° élém lors de
    l'appel factorielle(j)
    return S
```

Ainsi, il y a $1+\sum_{j=1}^{n}(2+j)$ opérations.

Ainsi la complexité temporelle vaut : $C(n)=1+2n+\dfrac{n(n+1)}{2}$

La complexité temporelle est donc en $O(n^2)$, elle est quadratique.

R3. Proposer une amélioration du programme complet afin d'obtenir une complexité linéaire.

Solution: Pour obtenir une complexité linéaire, il faut évaluer `factorielle(j)` et l'ajouter aux factorielles précédentes dans la même boucle `for`.

```
def somme_fact(n):
    F=1 #initialisation du calcul de la factorielle
    S=0 #initialisation de la somme
    for j in range(0,n+1):  # n itérations
        F=F*j
        S=S+F
    return S
```

On obtient alors une complexité $C(n)=2+n\times(1+1+1+1)=2+4n=O(n)$

Exercice 3 Fonctions récursives

On considère la fonction récursive ci-dessous :

```
def factorielle(n):
    if n==0 or n==1:
        return 1
    return n*factorielle(n-1)
```

R1. Quelle est la complexité temporelle de cet algorithme ?

Solution:
On note $\mathscr{C}(n)$ la complexité lors de l'appel de `factorielle(n)`.
Il y a 2 tests, suivi d'une multiplication et de l'appel de la fonction `factorielle(n-1)`
Ainsi : $\mathscr{C}(n) = 2 + 1 + \mathscr{C}(n-1)$
$\mathscr{C}(n)$ est donc une suite arithmétique de raison 3, de terme général $\mathscr{C}(n) = 3n + \mathscr{C}(0)$
Ainsi $\boxed{\mathscr{C}(n) = O(n)}$

R2. On souhaite maintenant étudier la suite suivante : $u_0 = 3$ et $u_{n+1} = 5u_n^2 + 2u_n + 1$. On propose :

```
def suite(n):
    if n==0:
        return(3)
    else:
        return(5*suite(n-1)**2+2*suite(n-1)+1)
```

Déterminer la complexité en temps. Commenter.

Solution: En nombre de multiplications : $\mathscr{C}(n) = 7 + 2\mathscr{C}(n-1)$ (1 test + 3 multiplications + 2 soustractions + 1 addition + 2 fois les opérations dans l'appel `suite(n-1)`).
C'est une suite arithmético-géométrique ($a = 2$ et $b = 7$), donc $r = \dfrac{b}{1-a} = \dfrac{7}{1-2} = -7$

$$\begin{aligned} \mathscr{C}(n) &= a^n \times (u_0 - r) + r \\ \mathscr{C}(n) &= 2^n \times (1+7) - 7 \\ \mathscr{C}(n) &= 8 \times 2^n - 7 \end{aligned}$$

Ainsi $\boxed{\mathscr{C}(n) = \mathscr{O}(2^n)$: c'est une complexité exponentielle$}$
Les deux appels de `suite(n-1)` à l'appel de `suite(n)` est responsable de cette complexité exponentielle.
On peut l'améliorer avec :

```
def suite(n):
    if n==0:
        return(3)
    else:
        suite_nmoins1=suite(n-1)
        return(5*suite_nmoins1**2+2*suite_nmoins1+1)
```

La fonction est ici de complexité linéaire.

I.4 Complexités des algorithmes de tris

Exercice 4 Tri bulle

On considère une liste `L` à n éléments. Le tri à bulles consiste à faire remonter les éléments les plus grands en permutant successivement les éléments du tableau : on parcourt le tableau, et à chaque fois que l'élément de gauche est strictement supérieur à l'élément de droite, on les permute. À la fin de ce parcours, le plus grand élément du tableau est en dernière position. On recommence le parcours du tableau pour trier les $n-1$ éléments, puis les $n-2$ éléments, etc. Le nom « tri à bulles » vient du fait que les éléments les plus grands remontent plus vite, comme les bulles dans l'eau. On propose les deux algorithmes suivants :

```
def tri_bulles(L):
    for i in range(len(L)):#n=len(L) étapes
        for j in range(len(L)-i-1): #on parcourt L du rang 0 au rang n-i-2 ;
    on trie L[:n-i-1], les i+1 derniers éléments sont déjà triés
```

```
            if L[j+1]<L[j]:
                L[j],L[j+1]=L[j+1],L[j]
    return L  # inutile : fonction avec effet de bord, L est modifiée
def tri_bulles_rec(L):
    if len(L)<=1: #condition d'arrêt
        return L
    for j in range(len(L)-1):
        if L[j+1]<L[j]: #on fait remonter l'élément le plus grand
            L[j],L[j+1]=L[j+1],L[j]
    return tri_bulles_rec(L[:len(L)-1])+[L[len(L)-1]] #appel récursif : on
   applique la fonction sur la liste L sans le dernier élément, et on
   concatène avec le dernier élément
```

R1. Déterminer la complexité en temps de l'algorithme itératif, puis de l'algorithme récursif.

Solution:

Pour la fonction itérative :

— Dans le meilleur des cas, la liste est déjà triée, il n'y a aucune réaffectation et uniquement des comparaisons.

Soit n la longueur de la liste à trier, il y a n étapes (pour $i \in [\![0, n-1]\!]$) au sein desquelles il y a $n-i-1$ comparaisons (pour $j \in [\![0, n-i-2]\!]$), donc il y a $\sum_{i=0}^{n-1}(n-i-1)$ comparaisons.

Avec $\sum_{i=0}^{n-1}(n-i-1) = n \times (n-1) - \sum_{i=0}^{n-1} i = n^2 - n - \frac{n(n-1)}{2} = \frac{n^2}{2} + \frac{3n}{2} = \mathcal{O}(n^2)$

— Dans le pire des cas, la liste est triée en sens inverse, il y a donc deux réaffectations (1 permutation) à chaque étape.

Il y a autant de comparaisons que précédemment, il faut à chacune des n étapes ajouter les réaffectations : il y a $2(n-i-1)$ réaffectations (il y a deux réaffectations par comparaison).

Ainsi $\mathscr{C}(n) = \sum_{i=0}^{n-1}(3(n-i-1)) = 3\left(\frac{n^2}{2} + \frac{3n}{2}\right) = \mathcal{O}(n^2)$

Pour la fonction récursive :

$$\begin{aligned}
\mathscr{C}(n) &= 1 + 3n + \mathscr{C}(n-1) \\
\mathscr{C}(n) - \mathscr{C}(n) &= 1 + 3n \\
\sum_{k=1}^{n} \mathscr{C}(k) - \mathscr{C}(k-1) &= \sum_{k=1}^{n} 1 + 3k \\
\mathscr{C}(n) - \mathscr{C}(0) &= 1 + 3 \times \frac{n(n+1)}{2}
\end{aligned}$$

Soit $\boxed{\mathscr{C}(n) = \mathcal{O}(n^2)}$

R2. Que peut-on dire de la complexité en mémoire ?

Complexité	En temps (nb de comparaisons)		
	Pire des cas	Meilleur des cas	En moyenne
Tri bulle	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Tri par insertion	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Tri fusion	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n\ln(n))$
Tri rapide	$\mathcal{O}(n^2)$	$\mathcal{O}(n\ln(n))$	$\mathcal{O}(n\ln(n))$

II Terminaison et correction d'un algorithme

II.1 Terminaison d'un algorithme

Définition : Variant de boucle

Un variant de boucle est une suite qui :

- dépend des variables de la boucle,
- à valeurs entières positives,
- strictement décroissante à chaque itération de la boucle.

Méthode : Comment montrer la terminaison d'un algorithme ?

Il est nécessaire de prouver la terminaison des boucles while et des fonctions récursives. Pour prouver la terminaison d'un algorithme, si cela est possible, il suffit souvent de prouver que les boucles se terminent et donc de :

1. trouver un variant de boucle (suite d'entiers, positifs, strictement décroissante),
2. montrer que le variant est minoré, qu'il franchit nécessairement une valeur limite liée à la condition d'arrêt.

II.2 Correction d'un algorithme

Définition : Correction d'un algorithme

La correction d'un algorithme est sa capacité à :

1. se terminer,
2. produire un résultat correct, conforme à ce qu'on attend de lui, quelles que soient les entrées.

On dit que la correction est :

- **partielle** si le résultat est correct lorsque l'algorithme se termine,
- **totale** si elle est partielle et que l'algorithme se termine.

Méthode : Comment montrer la correction d'un algorithme ?

La démarche générale de démonstration de la correction des boucles est une forme de **démonstration par récurrence**. La propriété à démontrer est nommée invariant de boucle. On procède donc en trois phases : l'initialisation de l'invariant, l'hérédité et la conclusion.

Définition : Invariant de boucle

Un invariant de boucle est une **propriété** liée aux variables d'un algorithme qui :

1. est vraie avant la boucle,
2. est invariante par les instructions de la boucle à chaque itération,
3. donne le résultat escompté si la condition de boucle est invalidée.

Méthode : Cas des algorithmes récursifs

La terminaison et la correction d'un algorithme récursif se montrent simultanément **par récurrence**. $\mathscr{P}(n)$: « La fonction `fonction(n)` termine et renvoie .... »

II.3 Quelques exemples

Exercice 5 Terminaison et correction d'un programme itératif

```
def fonction(n):
    r=2
    i=0
    while i<n:
        r=r*r
        i=i+1
    return r
```

R1. Donner une preuve formelle de terminaison de cette fonction.

Solution:
Pour prouver la terminaison de cette fonction, il est nécessaire de montrer que la boucle while se termine. Pour cela, il faut exhiber une suite à valeurs entières strictement décroissante.
Posons $u_i = n - i$. Initialement, $i = 0 < n$, donc on entre dans la boucle while et $u_0 = n > 0$.
À chaque passage dans la boucle, i augmente de 1 et u_i diminue de 1.
La suite (u_i) est donc une suite d'entiers positifs strictement décroissante, elle ne contient qu'un nombre fini de termes, et par conséquent l'algorithme se termine.

R2. On note r_k la valeur de r à la fin de la k$^{\text{e}}$ itération, avec $r_0 = 2$.
Démontrer que la propriété « $r_k = 2^{2^k}$ » est un invariant de boucle pour cette fonction.

Solution:
Pour tout $k \in [\![0, n]\!]$, on note $\mathscr{P}(k)$: « $r_k = 2^{2^k}$ est un invariant de boucle pour cette fonction ».

— **Initialisation** :
Pour $k = 0$, `fonction(0)` renvoie 2.
Or $r_0 = 2^{2^0} = 2^1 = 2$,
Donc $\mathscr{P}(0)$ est vraie

— **Hérédité**
Soit $k \in [\![1, n]\!]$, supposons que la propriété est vraie avant la k$^{\text{e}}$ itération (c'est-à-dire à la fin de la $(k-1)^{\text{e}}$ itération), c'est-à-dire $r_{k-1} = 2^{2^{k-1}}$
À la fin de la k$^{\text{e}}$ itération : $r_k = r_{k-1} \times r_{k-1} = 2^{2^{k-1}} \times 2^{2^{k-1}} = \left(2^{2^{k-1}}\right)^2 = 2^{2^{k-1} \times 2}$, donc $r_k = 2^{2^k}$
Si la propriété est vraie avant la k$^{\text{e}}$ itération, alors est est vraie à la fin de la k$^{\text{e}}$ itération.

— **Conclusion** On a donc démontré que $r_k = 2^{2^k}$ est un invariant de boucle de la fonction.

Or l'algorithme se termine à la n$^{\text{e}}$ itération, donc la fonction renvoie r_n, soit 2^{2^n}.

R3. Conclure sur ce que calcule cette fonction.

Solution: La fonction retourne 2^{2^n}.

Exercice 6 Terminaison et correction d'un programme récursif

Montrer la terminaison et la correction de l'algorithme suivant qui renvoie $n!$.

```
def factorielle(n):
    if n==0:
        return 1
    else :
        return n*factorielle(n-1)
```

Solution: Preuve de la terminaison et la correction, que l'on fait en même temps pour un algorithme récursif, en utilisant une démonstration par récurrence.

Notons, pour tout $n \in \mathbb{N}$, $\mathscr{P}(n)$ la propriété « `factorielle(n)` termine et renvoie $n!$ ».

— **Initialisation**.

Pour $n = 0$, `factorielle(0)` renvoie 1, et $0! = 1$. Donc $\mathscr{P}(0)$ vraie.

— **Hérédité**

Soit, $n \in \mathbb{N}$, supposons $\mathscr{P}(n)$ vraie.

$n + 1 \geq 1$, donc `factorielle(n+1)` renvoie $(n + 1) \times$ `factorielle(n)`.

`factorielle(n)` termine, donc `factorielle(n+1)` termine.

`factorielle(n)` renvoie $n!$, donc `factorielle(n+1)` renvoie $(n + 1) \times n! = (n + 1)!$

Donc $\mathscr{P}(n + 1)$ est vraie.

— **Conclusion.** Pour tout $n \in \mathbb{N}$, `factorielle(n)` termine et renvoie $n!$.

Exercice 7 Puissance rapide

L'algorithme de puissance rapide repose sur la propriété suivante :

pour a réel et n entier naturel non nul, $a^n = \begin{cases} a^{\frac{n}{2}} \times a^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ a^{\frac{n-1}{2}} \times a^{\frac{n-1}{2}} \times a & \text{si } n \text{ est impair} \end{cases}$

R1. Écrire une fonction récursive `puissance_rapide` de paramètres a et n qui renvoie la valeur de a^n en utilisant la propriété précédente.

Solution:

```
def puissance_rapide(a,n):
    """prend en entrée un réel a et un entier n, et renvoie a^n selon un algo rapide"""
    if n==0:
        return 1
    else:
        if n%2==0: #teste si n pair
            P=puissance_rapide(a,n//2)
            return P*P #appel récursif
        else:
            P=puissance_rapide(a,(n-1)//2)
            return a*P*P
```

R2. Montrer la terminaison et la correction de `puissance_rapide` (par récurrence forte).

Solution:

Pour prouver la terminaison, il faut trouver une suite d'entiers naturels strictement décroissante.

Posons (u_p) la suite des arguments successifs de l'appel de la fonction `puissance_rapide(a,n)`.

Lors du premier appel $u_0 = n$. Puis, pour tout $n \geq 1$, tel que u_{p-1} est non nul, $u_p = u_{p-1}//2$ est le quotient de la division euclidienne de u_{p-1} par 2, et ainsi $u_{p-1} < u_p$.

La suite (u_p) est une suite strictement décroissante d'entiers naturels, donc elle ne contient qu'un nombre fini de termes, donc l'algorithme se termine.

Correction (et terminaison)

Pour tout $n \in \mathbb{N}$, notons $\mathscr{P}(n)$ la proposition : « $\forall k < n$, la fonction `puissance_rapide(a,k)` termine et renvoie a^k renvoie ».

— **Initialisation**
`puissance_rapide(a,0)` renvoie 1, or $a^0 = 1$, donc $\mathscr{P}(0)$ est vraie

— **Hérédité** (récurrence forte)
Soit $n \in \mathbb{N}^*$, supposons $\mathscr{P}(n)$ vraie.
Exprimons `puissance_rapide(a,n+1)`
Comme $n+1 > 0$:

— si $n+1$ est pair : `puissance_rapide(a,n+1)=puissance_rapide(a,(n+1)//2)**2`.
Or le quotient de la division de $n+1$ par 2 est inférieur à n, donc `puissance_rapide(a,(n+1)//2)` termine, donc `puissance_rapide(a,n+1)` termine.

— Si $n+1$ est pair et comme `(n+1)//2)` est inférieur n, alors `puissance_rapide(a,(n+1)//2)` renvoie $a^{\frac{n+1}{2}}$
Alors `puissance_rapide(a,n+1)`$=\left(a^{\frac{n+1}{2}}\right)^2 = a^{n+1}$.

— Si $n+1$ est impair et comme `(n+1)//2)` est inférieur n, alors `puissance_rapide(a,(n+1)//2)` renvoie $a^{\frac{n+1-1}{2}} \times a^{\frac{n+1-1}{2}} \times a$
Alors `puissance_rapide(a,n+1)`$=\left(a^{\frac{n}{2}}\right)^2 \times a = a^{n+1}$.

Donc $\mathscr{P}(n+1)$ est vraie

— **Conclusion**

R3. Déterminer la complexité temporelle.

Solution:
Calculons la complexité en temps de `puissance_rapide` et montrons qu'elle est inférieure à la complexité de l'algorithme `puissance`.
Notons $\mathscr{C}(n)$ la complexité de l'appel à `puissance_rapide(a,n)` pour a quelconque.

— Pour $n = 0$, il n'y a pas de multiplications et un test , donc $\mathscr{C}(0) = 1$.

— Pour $n = 1$, il y a deux multiplications et deux tests : $\mathscr{C}(1) = 4$.

— Pour $n \geq 2$:

— Si n est pair, il y a 2 tests (`n==0` et `n%2==0`) + 1 multiplication (calcul de `**2`) + l'appel à `puissance_rapide(a,n//2)`, donc : $\mathscr{C}(n) = 3 + \mathscr{C}\left(\frac{n}{2}\right)$

— Si n est impair, il y a 2 tests (`n==0` et `n%2==0`) + 2 multiplications (calcul de `**2` et `*a`) + 1 soustraction (`n-1`) + l'appel à `puissance_rapide(a,(n-1)//2)`, donc : $\mathscr{C}(n) = 5 + \mathscr{C}\left(\frac{n-1}{2}\right)$

Plaçons-nous dans le meilleur puis dans le pire des cas :

— Dans le meilleur des cas, à chaque étape on calcule une puissance paire, ce qui se produit si n est de la forme $n = 2^p$, avec $p \in \mathbb{N}^*$.
Alors $\mathscr{C}(2^p) = 3 + \mathscr{C}(2^{p-1})$.
En posant $x_p = \mathscr{C}(2^p)$, on a $x_p = 3 + x_{p-1}$: (x_p) est donc une suite arithmétique de raison 3, donc $x_p = x_1 + 3(p-1)$ avec $x_1 = \mathscr{C}(2) = 2 + \mathscr{C}(1) = 6$
Ainsi $\mathscr{C}(2^p) = 3p + 3 = 3\log_2(2^p) + 3$
Soit, dans le meilleur des cas $\boxed{\mathscr{C}(n) = \mathscr{O}(\log_2(n))}$

— Dans le pire des cas, à chaque étape on calcule une puissance impaire, ce qui se produit si n est de la forme $n = 2^p - 1$, avec $p \in \mathbb{N}^*$.
Alors $\mathscr{C}(2^p - 1) = 5 + \mathscr{C}\left(\frac{2^p - 1 - 1}{2}\right) = 5 + \mathscr{C}\left(2^{p-1} - 1\right)$
On introduit $y_p = \mathscr{C}(2^p - 1)$ pour tout $p \in \mathbb{N}^*$, alors $y_p = 5 + y_{p-1}$

On en déduit $y_p = 5(p-1) + y_1$, avec $y_1 = \mathscr{C}(1) = 5$, soit $y_p = 5p$
Ainsi $\mathscr{C}(2^p - 1) = 5p = 5\log_2(2^p)$
Soit, dans le pire des cas $\boxed{\mathscr{C}(n) = \mathscr{O}(\log_2(n))}$

En conclusion, on trouve $\mathscr{C}(n) = \mathscr{O}(\log_2(n))$: la complexité de `puissance_rapide` est logarithmique est donc bien meilleure que la complexité linéaire de la fonction `puissance`.