Exemple introductif

On considère l'algorithme suivant :

Algorithme : Recherche simple

Entrées : $a \in \mathbb{N}$ et un tableau d'entiers t de longueur n
Sorties : Booléen indiquant si $a \in t$.

```
1 pour i ← 0 à n − 1 faire
2     si t[i] = a alors
3         return Vrai
4     fin
5 fin
6 return Faux
```

Exemple introductif

On considère l'algorithme suivant :

Algorithme : Recherche simple

Entrées : $a \in \mathbb{N}$ et un tableau d'entiers t de longueur n
Sorties : Booléen indiquant si $a \in t$.

```
1 pour i ← 0 à n − 1 faire
2     si t[i] = a alors
3         return Vrai
4     fin
5 fin
6 return Faux
```

1. Combien de comparaisons effectue cet algorithme dans le meilleur des cas ?

Exemple introductif

On considère l'algorithme suivant :

Algorithme : Recherche simple

Entrées : $a \in \mathbb{N}$ et un tableau d'entiers t de longueur n
Sorties : Booléen indiquant si $a \in t$.

```
1 pour i ← 0 à n − 1 faire
2     si t[i] = a alors
3         return Vrai
4     fin
5 fin
6 return Faux
```

1. Combien de comparaisons effectue cet algorithme dans le meilleur des cas ?
2. Même question dans le pire des cas.

Exemple introductif

On considère l'algorithme suivant :

Algorithme : Recherche simple

Entrées : $a \in \mathbb{N}$ et un tableau d'entiers t de longueur n
Sorties : Booléen indiquant si $a \in t$.

```
1 pour i ← 0 à n − 1 faire
2     si t[i] = a alors
3         return Vrai
4     fin
5 fin
6 return Faux
```

1. Combien de comparaisons effectue cet algorithme dans le meilleur des cas ?
2. Même question dans le pire des cas.
3. Que dire du cas où on recherche un élément a présent en un seul exemplaire dans le tableau t en supposant les positions équiprobables ?

Correction

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X = k) = \frac{1}{n}$. Donc,

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X = k) = \frac{1}{n}$. Donc,

$$E(X) = \sum_{k=1}^{n} k \frac{1}{n}$$

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X = k) = \frac{1}{n}$. Donc,

$$E(X) = \sum_{k=1}^{n} k \frac{1}{n}$$

$$E(X) = \frac{n+1}{2}$$

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X = k) = \frac{1}{n}$. Donc,

$$E(X) = \sum_{k=1}^{n} k \frac{1}{n}$$

$$E(X) = \frac{n+1}{2}$$

Le nombre de comparaisons varie donc avec les données du problème.

Correction

1. Si l'élément cherché est en première position dans le tableau on effectue une seule comparaison.
2. Si l'élément cherché n'est pas dans le tableau (ou qu'il y figure en dernière position) on effectue n comparaison.
3. on note X le nombre de comparaisons avant de trouver a, alors $p(X = k) = \frac{1}{n}$. Donc,

$$\begin{aligned} E(X) &= \sum_{k=1}^{n} k \frac{1}{n} \\ E(X) &= \frac{n+1}{2} \end{aligned}$$

Le nombre de comparaisons varie donc avec les données du problème.
On peut cependant toujours *majorer* le nombre de comparaisons, qui reste inférieur dans tous les cas à Kn où K est une constante et n la taille du tableau.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.
- comme les performances de l'algorithme varient en fonction des données, on s'intéresse généralement à une simple majoration dans le *pire des cas*.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.
- comme les performances de l'algorithme varient en fonction des données, on s'intéresse généralement à une simple majoration dans le *pire des cas*.
  Dans l'exemple bien que l'algorithme puisse parfois répondre avec une seule comparaison c'est le pire des cas qui nous intéresse.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.
- comme les performances de l'algorithme varient en fonction des données, on s'intéresse généralement à une simple majoration dans le *pire des cas*.
  Dans l'exemple bien que l'algorithme puisse parfois répondre avec une seule comparaison c'est le pire des cas qui nous intéresse.
- on fournit une majoration asymptotique c'est à dire à une constante multiplicative près et à partir d'un certain rang.

Calcul de complexité

L'exemple précédent est celui du calcul de la complexité d'un algorithme :

- on utilise une mesure de l'efficacité de l'algorithme.
  Dans l'exemple c'est le nombre de tests de comparaisons effectués par l'algorithme. D'autres mesures sont possibles, notamment le nombre d'opérations élémentaires ou encore la quantité de mémoire occupée.
- comme les performances de l'algorithme varient en fonction des données, on s'intéresse généralement à une simple majoration dans le *pire des cas*.
  Dans l'exemple bien que l'algorithme puisse parfois répondre avec une seule comparaison c'est le pire des cas qui nous intéresse.
- on fournit une majoration asymptotique c'est à dire à une constante multiplicative près et à partir d'un certain rang.
  Dans l'exemple précédent la majoration était Kn.

Définition

La complexité d'un algorithme est une mesure de son efficacité.

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations *élémentaires* nécessaire à l'exécution d'un algorithme.

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations *élémentaires* nécessaire à l'exécution d'un algorithme.
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données.

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations *élémentaires* nécessaire à l'exécution d'un algorithme.
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données.

Ces deux éléments varient en fonction de la taille et de la nature des données, on donne donc généralement une majoration dans le pire des cas.

Définition

La complexité d'un algorithme est une mesure de son efficacité. On parle notamment de :

- Complexité en temps : le nombre d'opérations *élémentaires* nécessaire à l'exécution d'un algorithme.
- Complexité en mémoire : l'occupation mémoire en fonction de la taille des données.

Ces deux éléments varient en fonction de la taille et de la nature des données, on donne donc généralement une majoration dans le pire des cas.

Remarque

On peut aussi parler de la complexité en moyenne, qui s'intéresse au nombre moyen d'operations effectuées par un algorithme sur un ensemble d'entrées de taille n.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.

  Par exemple les opérations arithmétiques, les tests, les affectations . . .

  ⚠ En Python, certaines opérations comme par exemple le test d'appartenance à une liste avec `in` ne sont pas des opérations élémentaires.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.
  Par exemple les opérations arithmétiques, les tests, les affectations . . .
  ⚠ En Python, certaines opérations comme par exemple le test d'appartenance à une liste avec `in` ne sont pas des opérations élémentaires.
- On exprime le coût de l'algorithme pour une entrée de taille `n` en nombre d'opérations élémentaires nécessaires à sa réalisation.

Calcul de la complexité temporelle

- On considère certaines opérations comme élémentaires, leur coût est alors majoré par une constante.
  Par exemple les opérations arithmétiques, les tests, les affectations . . .
  ⚠ En Python, certaines opérations comme par exemple le test d'appartenance à une liste avec `in` ne sont pas des opérations élémentaires.
- On exprime le coût de l'algorithme pour une entrée de taille `n` en nombre d'opérations élémentaires nécessaires à sa réalisation.
  Par exemple, la fonction `f` définie par `def f(x) = return x*x + 2*x + 3` demande 5 opérations quelque soit la taille de l'entrée `x`.

Exemple

On considère la fonction suivante :

```
1  def somme(lst):
2      s = 0
3      i = 0
4      while (i<len(lst)):
5          s += lst[i]
6          i += 1
7      return s
```

1. Quelle est la complexité de la fonction suivante en nombre d'opérations élémentaires ?
2. La complexité serait-elle la même si on remplaçait la boucle `while` par un `for` ?

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.
- L'outil mathématique associé est la notion de domination d'une suite :
  Etant donné deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ à valeurs strictement positives.
  On dit que $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier $K > 0$ et un rang $N \in \mathbb{N}$ tel que :
  $\forall n \in \mathbb{N}, n > N$, on a $u_n \leq K v_n$.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.
- L'outil mathématique associé est la notion de domination d'une suite :
  Etant donné deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ à valeurs strictement positives.
  On dit que $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier $K > 0$ et un rang $N \in \mathbb{N}$ tel que :
  $\forall n \in \mathbb{N}, n > N$, on a $u_n \leq K v_n$.
  On note alors $u = \mathcal{O}(v)$ (ou encore $u \in \mathcal{O}(v)$) et on dit que u est un grand $\mathcal{O}$ de v.

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.
- L'outil mathématique associé est la notion de domination d'une suite :
  Etant donné deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ à valeurs strictement positives.
  On dit que $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier $K > 0$ et un rang $N \in \mathbb{N}$ tel que :
  $\forall n \in \mathbb{N}, n > N$, on a $u_n \leq K v_n$.
  On note alors $u = \mathcal{O}(v)$ (ou encore $u \in \mathcal{O}(v)$) et on dit que u est un grand $\mathcal{O}$ de v.
  « u_n est inférieur à v_n à une constante multiplicative près et pour n assez grand ».

Majoration asymptotique

- En pratique, seul une majoration asymptotique du coût $C(n)$ d'un algorithme nous intéresse et pas sa détermination exacte.
  Par exemple, si le coût de l'algorithme est $C(n) = 3n + 15$ opérations élémentaires, on dira que $C(n)$ est majoré asymptotiquement par n car $C(n) < 4n$ dès que $n > 15$.
- L'outil mathématique associé est la notion de domination d'une suite :
  Etant donné deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ à valeurs strictement positives.
  On dit que $(u_n)_{n\in\mathbb{N}}$ est dominée par $(v_n)_{n\in\mathbb{N}}$ lorsqu'il existe un entier $K > 0$ et un rang $N \in \mathbb{N}$ tel que :
  $\forall n \in \mathbb{N}, n > N$, on a $u_n \leq K v_n$.
  On note alors $u = \mathcal{O}(v)$ (ou encore $u \in \mathcal{O}(v)$) et on dit que u est un grand $\mathcal{O}$ de v.
  « u_n est inférieur à v_n à une constante multiplicative près et pour n assez grand ».
  Dans l'exemple précédent, $C(n) = 3n + 15$ est un $\mathcal{O}(n)$.

Exemples

- Montrer que (a_n) de terme général $10n + 3$ est un $\mathcal{O}(n)$

Exemples

- Montrer que (a_n) de terme général $10n + 3$ est un $\mathcal{O}(n)$
- Montrer que (b_n) de terme général $n^2 + n + 1$ est un $\mathcal{O}(n^2)$

Exemples

- Montrer que (a_n) de terme général $10n+3$ est un $\mathcal{O}(n)$
- Montrer que (b_n) de terme général n^2+n+1 est un $\mathcal{O}(n^2)$
- Déterminer un grand $\mathcal{O}$ de (c_n) de terme général $7n+\ln(n)$

Exemples

- Montrer que (a_n) de terme général $10n+3$ est un $\mathcal{O}(n)$
  $10n+3 < 11n$ pour $n > 3$
- Montrer que (b_n) de terme général n^2+n+1 est un $\mathcal{O}(n^2)$
- Déterminer un grand $\mathcal{O}$ de (c_n) de terme général $7n+\ln(n)$

Exemples

- Montrer que (a_n) de terme général $10n+3$ est un $\mathcal{O}(n)$
  $10n+3 < 11n$ pour $n > 3$
- Montrer que (b_n) de terme général n^2+n+1 est un $\mathcal{O}(n^2)$
  $n^2+n+1 < 2n^2$ pour $n > 2$
- Déterminer un grand $\mathcal{O}$ de (c_n) de terme général $7n+\ln(n)$

Exemples

- Montrer que (a_n) de terme général $10n+3$ est un $\mathcal{O}(n)$
  $10n+3 < 11n$ pour $n > 3$
- Montrer que (b_n) de terme général n^2+n+1 est un $\mathcal{O}(n^2)$
  $n^2+n+1 < 2n^2$ pour $n > 2$
- Déterminer un grand $\mathcal{O}$ de (c_n) de terme général $7n+\ln(n)$
  Comme $\ln(n) < n$, $c_n < 8n$ et donc (c_n) est un $\mathcal{O}(n)$.

Remarques

Ecrire $u_n = \mathcal{O}(v_n)$ traduit une *majoration asymptotique*, c'est à dire que « (u_n) est au plus de l'ordre de (v_n) ».

Par exemple si $u_n = 42n + 2024$, on pourrait écrire $u_n = O(n)$ mais aussi que $u_n = \mathcal{O}(n^2)$ (ou encore $u_n = \mathcal{O}(n^3)$).

On veut généralement donner le « meilleur grand $\mathcal{O}$ ». Afin d'exprimer formellement cette notion, on note :

.

Remarques

Ecrire $u_n = \mathcal{O}(v_n)$ traduit une *majoration asymptotique*, c'est à dire que « (u_n) est au plus de l'ordre de (v_n) ».

Par exemple si $u_n = 42n + 2024$, on pourrait écrire $u_n = O(n)$ mais aussi que $u_n = \mathcal{O}(n^2)$ (ou encore $u_n = \mathcal{O}(n^3)$).

On veut généralement donner le « meilleur grand $\mathcal{O}$ ». Afin d'exprimer formellement cette notion, on note :

- $u_n = \Omega(v_n)$ s'il existe $K \in \mathbb{R}^+$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0, u_n \geqslant K v_n$, c'est à dire que « (u_n) est au moins de l'ordre de (v_n) »

.

Remarques

Ecrire $u_n = \mathcal{O}(v_n)$ traduit une *majoration asymptotique*, c'est à dire que « (u_n) est au plus de l'ordre de (v_n) ».

Par exemple si $u_n = 42n + 2024$, on pourrait écrire $u_n = O(n)$ mais aussi que $u_n = \mathcal{O}(n^2)$ (ou encore $u_n = \mathcal{O}(n^3)$).

On veut généralement donner le « meilleur grand $\mathcal{O}$ ». Afin d'exprimer formellement cette notion, on note :

- $u_n = \Omega(v_n)$ s'il existe $K \in \mathbb{R}^+$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0, u_n \geqslant K v_n$, c'est à dire que « (u_n) est au moins de l'ordre de (v_n) »
- $v_n = \Theta(v_n)$ si $u_n = \mathcal{O}(v_n)$ et $v_n = \mathcal{O}(u_n)$, c'est à dire que « (u_n) est de l'ordre de (v_n) »

.

Remarques

Ecrire $u_n = \mathcal{O}(v_n)$ traduit une *majoration asymptotique*, c'est à dire que « (u_n) est au plus de l'ordre de (v_n) ».
Par exemple si $u_n = 42n + 2024$, on pourrait écrire $u_n = O(n)$ mais aussi que $u_n = \mathcal{O}(n^2)$ (ou encore $u_n = \mathcal{O}(n^3)$).
On veut généralement donner le « meilleur grand $\mathcal{O}$ ». Afin d'exprimer formellement cette notion, on note :

- $u_n = \Omega(v_n)$ s'il existe $K \in \mathbb{R}^+$ et $n_0 \in \mathbb{N}$ tel que pour tout $n \geqslant n_0, u_n \geqslant K v_n$, c'est à dire que « (u_n) est au moins de l'ordre de (v_n) »
- $v_n = \Theta(v_n)$ si $u_n = \mathcal{O}(v_n)$ et $v_n = \mathcal{O}(u_n)$, c'est à dire que « (u_n) est de l'ordre de (v_n) »

On utilisera principalement la notation $\mathcal{O}$, en gardant à l'esprit qu'on essaye toujours d'avoir la meilleure majoration.

Complexités usuelles

Complexité	Nom	Exemple

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$\mathcal{O}(n)$	Linéaire	Recherche simple dans une liste

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$\mathcal{O}(n)$	Linéaire	Recherche simple dans une liste
$\mathcal{O}(n\log(n))$	Linéaritmique	Tri fusion

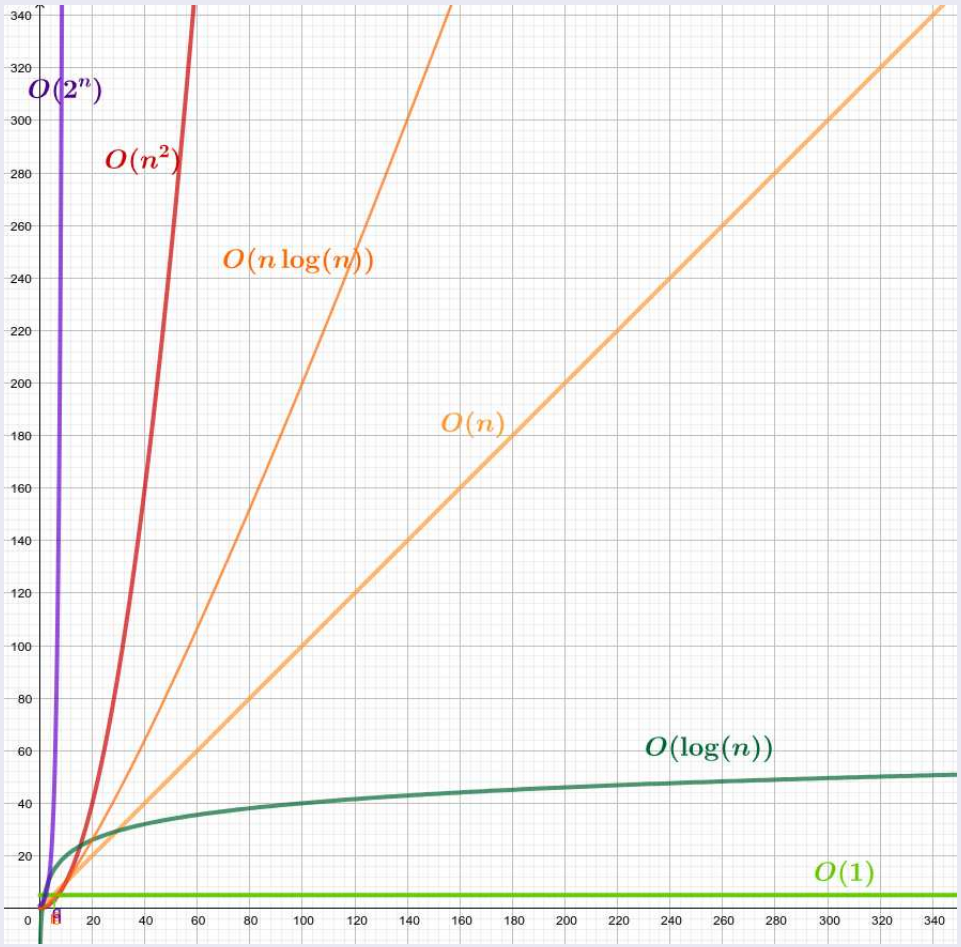
Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$\mathcal{O}(n)$	Linéaire	Recherche simple dans une liste
$\mathcal{O}(n\log(n))$	Linéaritmique	Tri fusion
$\mathcal{O}(n^2)$	Quadratique	Tri par insertion d'une liste

Complexités usuelles

Complexité	Nom	Exemple
$\mathcal{O}(1)$	Constant	Accéder à un élément d'un tableau
$\mathcal{O}(\log(n))$	Logarithmique	Recherche dichotomique dans une liste
$\mathcal{O}(n)$	Linéaire	Recherche simple dans une liste
$\mathcal{O}(n\log(n))$	Linéaritmique	Tri fusion
$\mathcal{O}(n^2)$	Quadratique	Tri par insertion d'une liste
$\mathcal{O}(2^n)$	Exponentielle	Algorithme par force brute pour le sac à dos

Représentation graphique



Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n=10$	$n=100$	$n=1000$	$n=10^6$	$n=10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$					
$\mathcal{O}(n\log(n))$					
$\mathcal{O}(n^2)$					
$\mathcal{O}(2^n)$					

Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$	✔	✔	✔	✔	$\simeq 10$s
$\mathcal{O}(n\log(n))$					
$\mathcal{O}(n^2)$					
$\mathcal{O}(2^n)$					

Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$	✔	✔	✔	✔	$\simeq 10$s
$\mathcal{O}(n\log(n))$	✔	✔	✔	✔	$\simeq 1,5$ mn
$\mathcal{O}(n^2)$					
$\mathcal{O}(2^n)$					

Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$	✔	✔	✔	✔	$\simeq 10$s
$\mathcal{O}(n\log(n))$	✔	✔	✔	✔	$\simeq 1,5$ mn
$\mathcal{O}(n^2)$	✔	✔	✔	$\simeq 3$ h	$\simeq 300$ ans
$\mathcal{O}(2^n)$					

Temps de calcul effectif

Sur un ordinateur réalisant 100 million d'opérations par seconde, en notant ✔ un temps de calcul quasi instantané et ✘ un temps de calcul inaccessible :

	$n = 10$	$n = 100$	$n = 1000$	$n = 10^6$	$n = 10^9$
$\mathcal{O}(\log(n))$	✔	✔	✔	✔	✔
$\mathcal{O}(n)$	✔	✔	✔	✔	$\simeq 10$s
$\mathcal{O}(n\log(n))$	✔	✔	✔	✔	$\simeq 1,5$ mn
$\mathcal{O}(n^2)$	✔	✔	✔	$\simeq 3$ h	$\simeq 300$ ans
$\mathcal{O}(2^n)$	✔	✘	✘	✘	✘

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.
  $0.015 \times 250 = 3.75$, on peut donc prévoir un temps de calcul d'environ 3,75 secondes

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.
  $0.015 \times 250 = 3.75$, on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.
  $0.015 \times 250 = 3.75$, on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.
  La taille des données a été multiplié par 250, la complexité étant quadratique le temps de calcul sera approximativement multiplié par $250^2 = 62500$

Exemples

- On suppose qu'on dispose d'un algorithme de complexité linéaire travaillant sur une liste, il traite une liste de 1 000 éléments en 0,015 secondes. Donner une estimation du temps de calcul pour une liste de 250 000 éléments.
  La taille des données a été multiplié par 250, la complexité étant lineaire le temps de calcul sera aussi approximativement multiplié par 250.
  $0.015 \times 250 = 3.75$, on peut donc prévoir un temps de calcul d'environ 3,75 secondes
- Même question pour un algorithme de complexité quadratique qui traite une liste de 1 000 éléments en 0,07 secondes.
  La taille des données a été multiplié par 250, la complexité étant quadratique le temps de calcul sera approximativement multiplié par $250^2 = 62500$
  $0.07 \times 62\,500 = 4375$, on peut donc prévoir un temps de calcul d'environ $4\,375$ secondes, c'est-à-dire près d'une heure et 15 minutes !

Equation de complexité

Dans le cas des fonction récursives, la complexité pour une entrée de taille n s'exprime à partir de complexité pour des tailles inférieures. On est donc amené à résoudre une *équation de complexité*.

Equation de complexité

Dans le cas des fonction récursives, la complexité pour une entrée de taille n s'exprime à partir de complexité pour des tailles inférieures. On est donc amené à résoudre une *équation de complexité*.

Exemple

Par exemple si on considère la version récursive du calcul de la somme d'une liste d'entiers en OCaml :
Alors on a $C(n) = C(n-1) + a$, et donc $C(n)$ est arithmétique de raison a et $C(n)$ est un $O(n)$.

Les tours de Hanoï

On rappelle que le jeu des tours de Hanoï peut être résolu de façon élégante par récursion. On note $T(n)$ le nombre de mouvement minimal nécessaire afin de résoudre Hanoï avec n disques en utilisant l'algorithme récursif.

1. Déterminer $T(1)$
2. Exprimer $T(n)$ en fonction $T(n-1)$
3. En déduire la complexité de l'algorithme.