UCLouvain

Algorithmique

Cours préparatoires au master en sciences informatiques

Vincent Branders

2022–2023

Bornes supérieures, inférieures et exactes

Rappels

- On s'intéresse au temps calcul (ou à l'espace mémoire) nécessaire pour résoudre un problème
- L'analyse expérimentale présente de nombreuses limites pratiques et théoriques
- Les données du problème sont un facteur important vis-à-vis du temps et de l'espace nécessaire
  - On étudie généralement le(s) pire(s) cas
- La complexité temporelle est une analyse asymptotique ($\lim_{n \to \infty}$) du nombre d'opérations effectuées par un algorithme selon la taille du problème
  - Donc on peut négliger les constantes (variable mais ne dépend pas de la taille du problème)
  - Donc on s'intéresse aux opérations primitives plutôt qu'aux opérations élémentaires

$\Rightarrow$ mais le calcul du nombre d'opérations primitives est long ...

Reprenons le compte des opérations primitives

Algorithm: arrayMax
Input: A un tableau de n entiers ($n > 0$)
Output: La valeur maximale dans A

```
currentMax ← A[0]                                   // 2 opérations
for i ← 1 to n − 1 do                               // 1 + 2.(n − 1) + 2.n op
    if currentMax < A[i] then                       // 3 op., exécutées n − 1 fois
        currentMax ← A[i]                           // 2 op., exécutées au plus n − 1 fois
return currentMax                                   // 1 opération
```

$i++ \equiv i \leftarrow i + 1$: 2 opérations primitives ; $i \leq n - 1$: 2 opérations primitives

Dans le pire cas (p.ex. A en ordre croissant), $9n - 3$ opérations primitives
Dans le meilleur cas (p.ex. A en ordre décroissant), $7n - 1$ opérations primitives

On souhaite une fonction qui indique le nombre d'opérations primitives pour un problème de taille n quelconque

Intuition sur les notions de bornes


```
Algorithm: arrayMax
Input: A un tableau de n entiers (n > 0)
Output: La valeur maximale dans A
currentMax ← A[0]                                          // 2 opérations
for i ← 1 to n − 1 do                                      // 1 + 2.(n − 1) + 2.n op
    if currentMax < A[i] then                              // 3 op., exécutées n − 1 fois
        currentMax ← A[i]                                  // 2 op., exécutées au plus n − 1 fois
return currentMax                                          // 1 opération
```

On souhaite une fonction qui indique le nombre d'opérations primitives pour un problème de taille n quelconque

Candidates :

- ~~$f(n) = 7n - 1$: trop petit dans le pire cas~~

  On s'intéresse généralement au pire cas

- $f(n) = 9n - 3$: trop grand dans le meilleur cas

  Besoin de "compter" le nombre d'opérations primitives

Borne supérieure

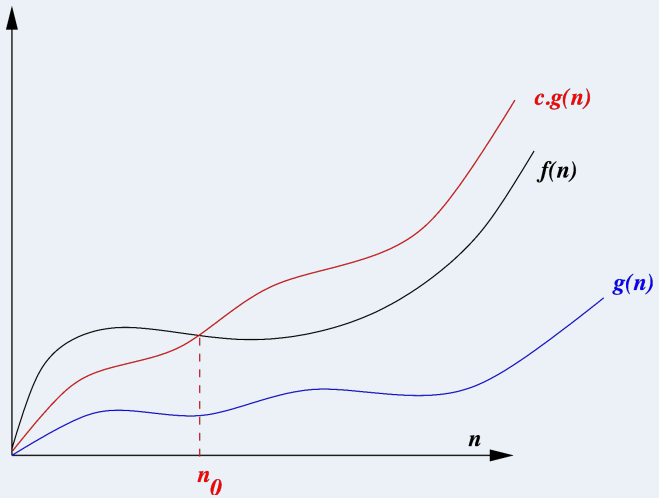
Fonction qui renvoie un nombre toujours supérieur, ou égal, au nombre d'opérations primitives réellement effectuées pour n'importe quelle instance de taille n

Définition : $\mathcal{O}$ (“grand O”)

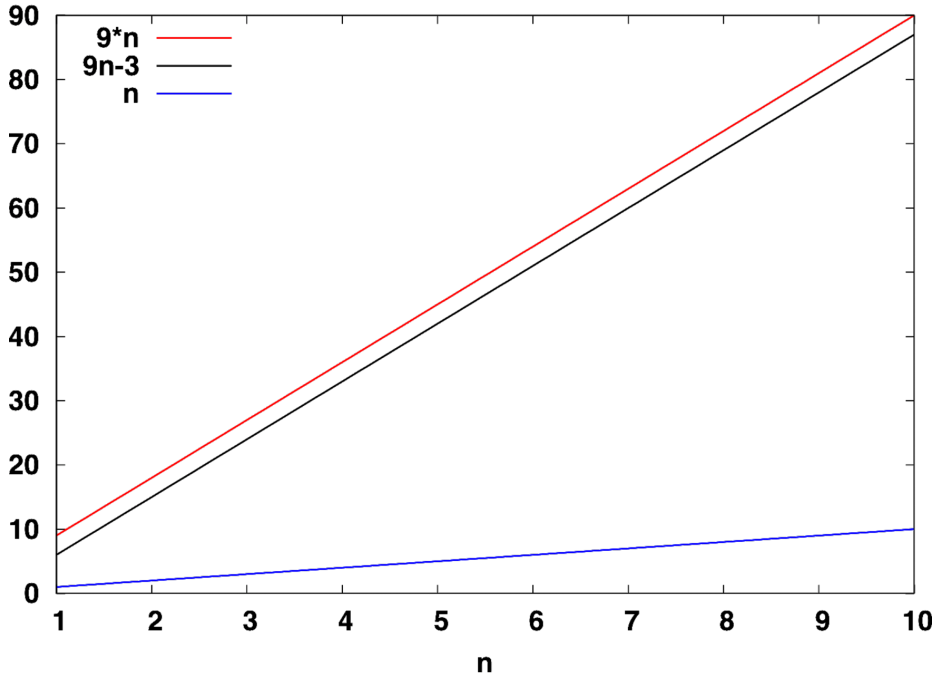
$f(n) \in \mathcal{O}(g(n))$ si $\exists c > 0$, $\exists n_0 \geq 1$ tels que $f(n) \leq c.g(n)$, $\forall n \geq n_0$

La fonction $g(n)$ consitute

- une borne supérieure de $f(n)$
- à une constante multiplicative c près
- et pour autant que n soit suffisament grand

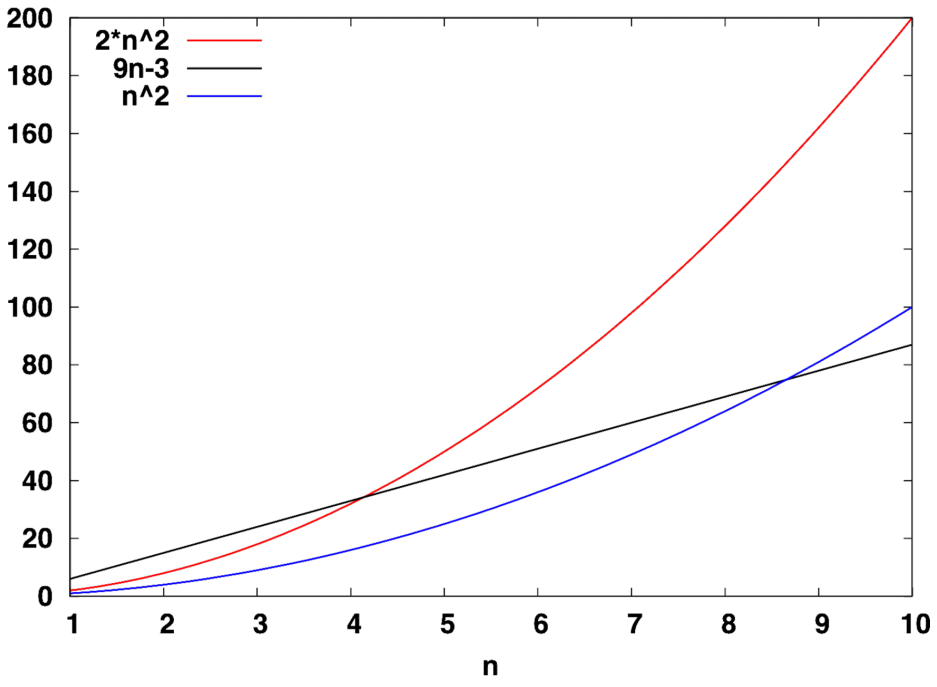


Utilisation de la notation $\mathcal{O}$



$9n - 3 \in \mathcal{O}(n)$ car $9n - 3 \leq 9.n$ pour $n \geq 1$

Utilisation de la notation $\mathcal{O}$



$9n - 3 \in \mathcal{O}\left(n^2\right)$

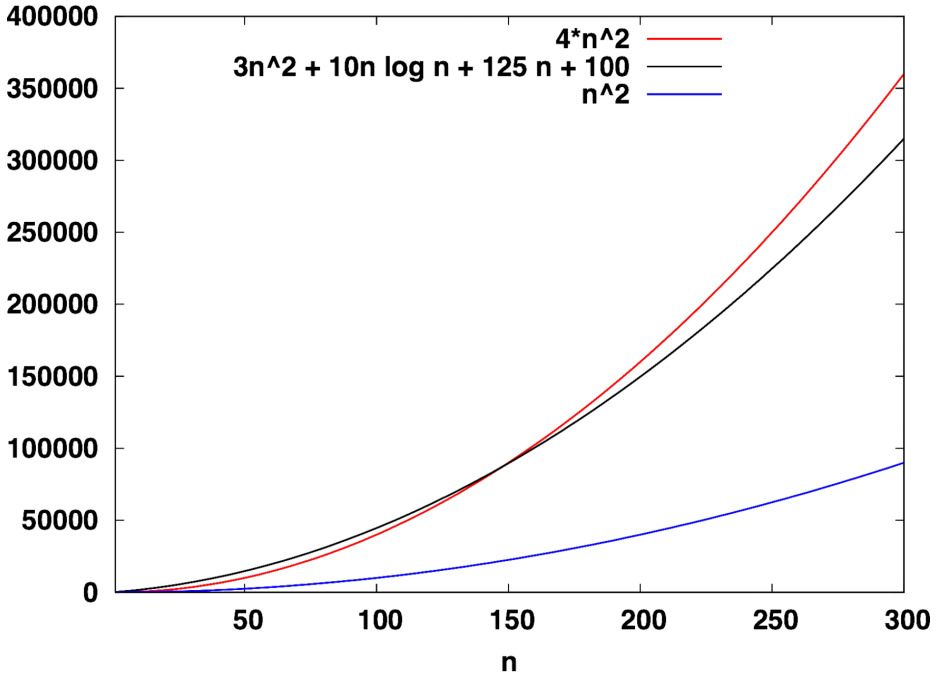
car $9n - 3 \leq 2.n^2$ pour $n \geq 5$
ou $9n - 3 \leq 1.n^2$ pour $n \geq 9$

Utilisation de la notation $\mathcal{O}$

$$
\begin{aligned}
9n-3 &\in \mathcal{O}(n) \\
9n-3 &\in \mathcal{O}(n^2) \\
9n-3 &\in \mathcal{O}(n^3) \\
&\dots
\end{aligned}
$$

- On s'intéresse à la borne la plus simple et la plus stricte possible
  - $9n-3 \in \mathcal{O}(n)$
  - n est une borne supérieure de $9n-3$, à une constante multiplicative près (p.ex. 9)
- Il suffit de garder les termes dominants et supprimer les constantes

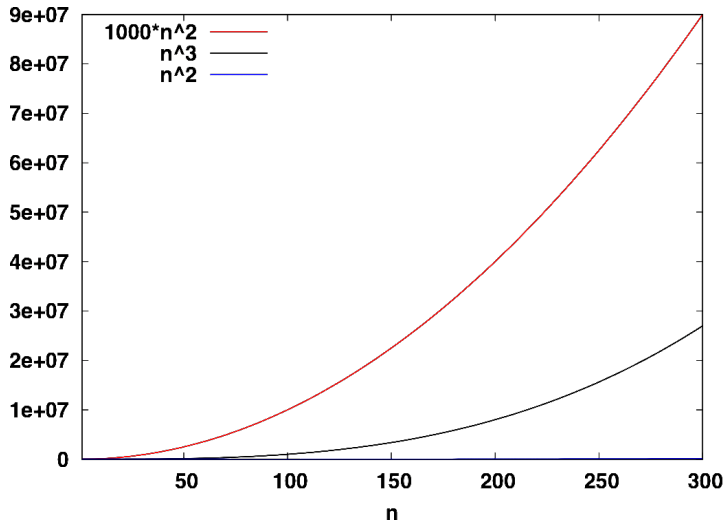
Termes dominants

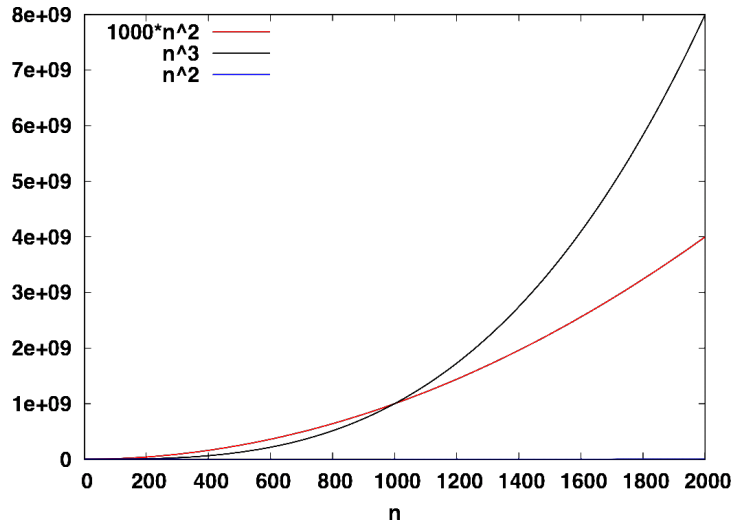


$3n^2 + 10n \log_{10} n + 125n + 100 \in \mathcal{O}\left(n^2\right)$ car
$3n^2 + 10n \log_{10} n + 125n + 100 \leq 4.n^2$ pour $n \geq 148$

Utilisation de la notation $\mathcal{O}$

- $2^{100} \approx 1.26 \times 10^{30} \in \mathcal{O}(1)$ car $2^{100} \leq 2^{100}.1$ pour $n \geq 1$
- $n^3 \notin \mathcal{O}(n^2)$ même si $n^3 \leq 1000.n^2$ pour $n \leq 300$ (voir plus ...)





- $n^3 \notin \mathcal{O}(n^2)$ car $n^3 \not\leq 1000.n^2$ pour $n \geq 1000$

Relations de dominance

Important !

$$n! \gg 3^n \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \sqrt{n} \gg \log n \gg 1$$

- $n!$ croît (finalement) plus vite que toute exponentielle
- a^n croît plus vite qu'une autre exponentielle b^n de plus petite base $(1 \leq b < a)$
- Toute exponentielle (de base > 1) croît plus vite que toute puissance
- n^a croît plus vite qu'une autre puissance n^b de degré moindre $(b < a)$
- Le logarithme croît moins vite que toute puissance positive, même fractionnaire $(\log n \ll \sqrt{n} = n^{\frac{1}{2}})$
- Le logarithme croît néanmoins et finira donc par dépasser n'importe quelle constante $(\log n \gg 1)$

Exercices sur l'identification des bornes supérieures :
https://inginious.org/course/prepa-master/Algo_1_bigO

Retour à notre exemple

Algorithm: arrayMax
Input: A un tableau de n entiers ($n > 0$)
Output: La valeur maximale dans A

```
currentMax ← A[0]                                   // O(1)
for i ← 1 to n − 1 do                               // O(n) itérations
    if currentMax < A[i] then                       // O(1)
        currentMax ← A[i]                           // O(1)
return currentMax                                   // O(1)
```

La complexité temporelle de l'algorithme est globalement en $\mathcal{O}(n)$

- Meilleur cas : $7n - 1 \in \mathcal{O}(n)$
- Pire cas : $9n - 3 \in \mathcal{O}(n)$
- Tous les cas $\in \mathcal{O}(n)$

On ne calcule pas le nombre d'opérations primitives (ni élémentaires) mais on raisonne en ordre de grandeur avec la notation $\mathcal{O}$ (ou Ω ou Θ)

Un exemple déjà vu

Algorithm: surprise
Input: Un entier strictement positif n
Output: Un tableau A de taille $n+1$ vérifiant une propriété à découvrir

```
for i ← 0 to n do
    temp ← i
    for j ← 1 to 3 do
        temp ← temp × i
    A[i] ← temp
return A
```

Determinez la complexité temporelle de l'algorithme `surprise`

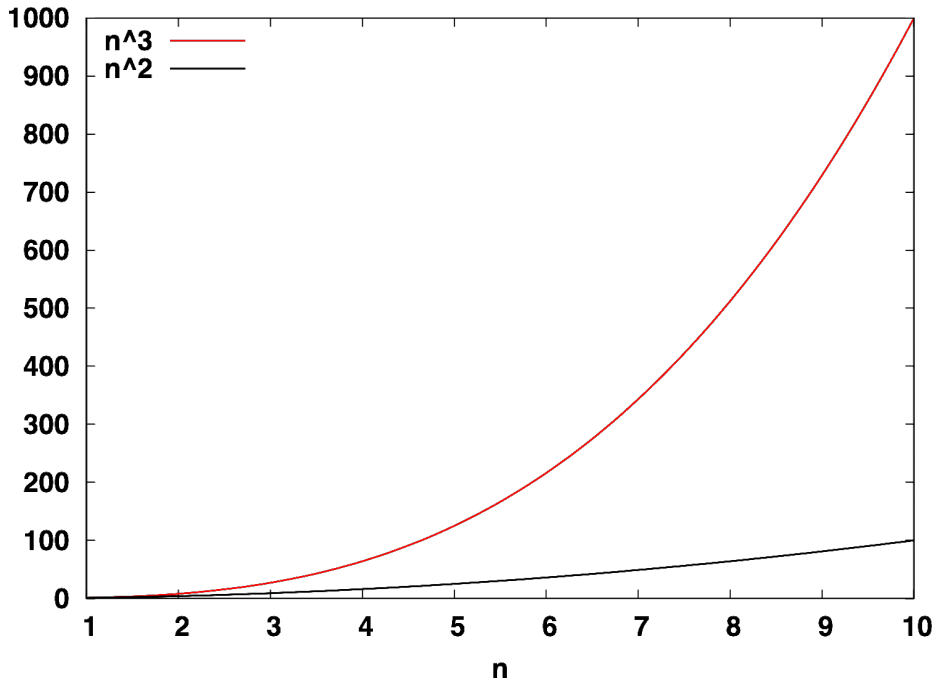
`surprise` $\in \mathcal{O}(n)$ (en réalité $\Theta(n)$)

Notation Ω

Oméga

Ω désigne une borne inférieure : $f(n) \in \Omega(g(n))$ si $g(n) \in \mathcal{O}(f(n))$

Par exemple, $n^3 \in \Omega(n^2)$ car $n^2 \in \mathcal{O}(n^3)$



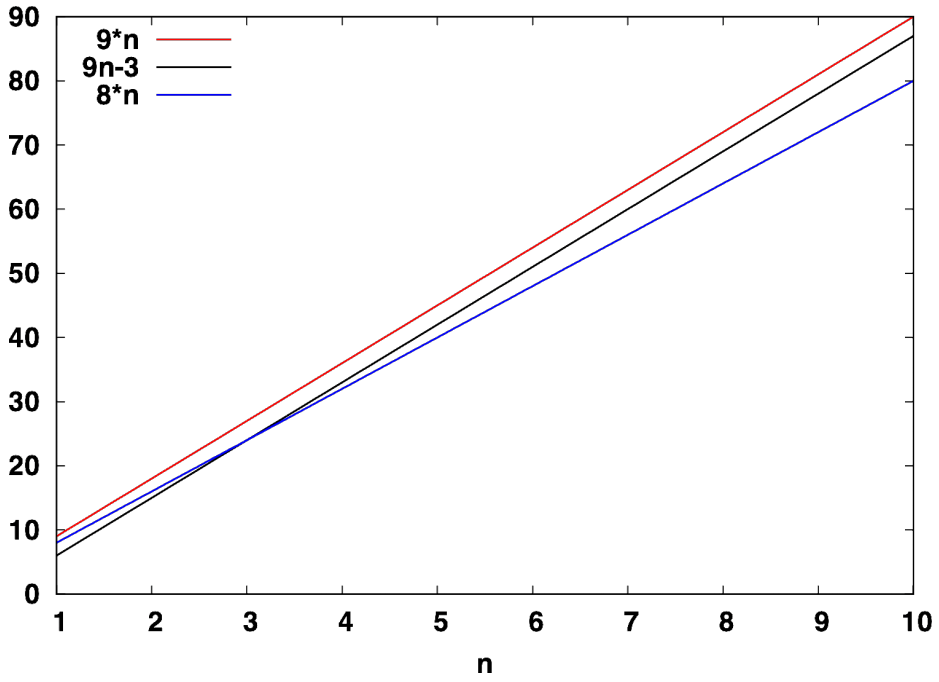
Notation Θ

Théta

Θ désigne une fonction asymptotiquement équivalente :

$$f(n) \in \Theta(g(n)) \text{ si } f(n) \in \mathcal{O}(g(n)) \text{ et } f(n) \in \Omega(g(n))$$

Par exemple, $9n - 3 \in \Theta(n)$ car $8.n \leq 9n - 3 \leq 9.n$ pour $n \geq 4$



Pourquoi distinguer $\mathcal{O}$ et Θ ?

Algorithm: arrayFirstNegative
Input: A, un tableau de n entiers ($n > 0$)
Output: L'indice du premier entier negatif dans A

```
for i ← 0 to n − 1 do
    if A[i] < 0 then
        return i                    // Indice du premier entier négatif
return −1                           // Aucun entier négatif dans A
```

La complexité temporelle

- dans le meilleur cas est $\mathcal{O}(1)$ mais aussi $\Theta(1)$
- dans le pire cas est $\mathcal{O}(n)$ mais aussi $\Theta(n)$

La complexité temporelle de `firstNegative` est globalement en $\mathcal{O}(n)$, *mais **pas** en* $\Theta(n)$

Pourquoi distinguer $\mathcal{O}$ et Θ ?

Algorithm: arrayMax
Input: A un tableau de n entiers ($n > 0$)
Output: La valeur maximale dans A

```
currentMax ← A[0]                                          // Θ(1)
for i ← 1 to n − 1 do                  // O(n) et Ω(n) ⇒ Θ(n) itérations
    if currentMax < A[i] then                              // Θ(1)
        currentMax ← A[i]                                  // Θ(1)
return currentMax                                          // Θ(1)
```

Note

La complexité dans le *meilleur cas* n'est **pas toujours**, *en ordre de grandeur*, **inférieure** à la complexité dans le *pire cas* Par exemple, la complexité dans tous les cas de `arrayMax` est $\Theta(n)$

- Meilleur cas : $7n - 1 \in \mathcal{O}(n)$ **mais aussi** $\Theta(n)$
- Pire cas : $9n - 3 \in \mathcal{O}(n)$ **mais aussi** $\Theta(n)$
- Tous les cas $\in \Theta(n)$

Exercices sur la relation entre fonctions :

https://inginious.org/course/prepa-master/Algo_2_Relations

Exercices sur les notations $\mathcal{O}, \Omega, \Theta$:

https://inginious.org/course/prepa-master/Algo_3_Notations_vrai_faux

Exercices sur les notions de bornes :

https://inginious.org/course/prepa-master/Algo_4_Summary

Exercices sur l'analyse de complexité d'algorithmes :

https://inginious.org/course/prepa-master/Algo_5_mystere

https://inginious.org/course/prepa-master/Algo_6_secret

https://inginious.org/course/prepa-master/Algo_7_paire_max_brute

Résumé

- La complexité temporelle d'un algorithme mesure comment évolue le temps calcul du programme associé
  - lorsque la taille du problème augmente
  - en négligeant des constantes multiplicatives et des termes non-dominants
- Ces complexités s'expriment en $\mathcal{O}$ (borne supérieure), Ω (borne inférieure) ou Θ (borne exacte)
- La complexité dans le pire cas **peut être différente** de la complexité dans le meilleur cas
  - Exemple : $\Theta(1)$ meilleure cas, $\Theta(n^2)$ pire cas $\Rightarrow$ globalement $\mathcal{O}(n^2)$
- La complexité dans le pire cas est **parfois égale** à la complexité dans le meilleur cas
  - Exemple : $\Theta(n)$ meilleure cas, $\Theta(n)$ pire cas $\Rightarrow$ globalement $\Theta(n)$