VII. Complexité d'un algorithme

1 Complexité en temps d'un algorithme

Définition 1. *On appelle* complexité temporelle *d'un algorithme la fonction qui associe à la taille entière n de la donnée le temps d'exécution de l'algorithme.*

Exemple 1. *Complexité temporelle de la fonction factorielle.*

Fonction*: factorielle(n)*
Action*: Calcul de la factorielle f d'un entier n*
Début
  $f \leftarrow 1$
  Pour k **allant de** 1 **à** n **faire**
    $f \leftarrow kf$
  FinPour
  Renvoyer f
Fin

L'algorithme précédent effectue $n+1$ affectations et n multiplications.

Remarque 1. *Le temps d'exécution de l'affectation d'une variable entière est négligeable devant celui d'une multiplication de valeurs entières.*

Exercice 1. *On considère les algorithmes suivants :*

Fonction*: pow(x, n)*
Début
  $l \leftarrow [\,]$
  Pour k **allant de** 1 **à** n **faire**
    $p \leftarrow x^k$
    Ajout de p **à la liste** l
  FinPour
  Renvoyer l
Fin

Fonction*: fastpow(x, n)*
Début
  $l \leftarrow [\,]$
  $p \leftarrow 1$
  Pour k **allant de** 1 **à** n **faire**
    $p \leftarrow xp$
    Ajout de p **à la liste** l
  FinPour
  Renvoyer l
Fin

1. *Déterminer la sortie de chacun des algorithmes (justifier au moyen d'un invariant de boucle).*
2. *Déterminer en fonction de n le nombre de multiplications effectuées dans chacun des algorithmes.*

Exemple 2. *Complexité temporelle du calcul de la somme des valeurs d'une liste de nombres.*

Fonction*: somme(l)*
Action*: Calcul de la somme s des valeurs d'une liste de nombres l*
Début
  $s \leftarrow 0$
  Pour k **allant de** 0 **à** longueur$(l) - 1$ **faire**
    $s \leftarrow s + l_k$
  FinPour
  Renvoyer s
Fin

On note n la longueur de la liste l, l'algorithme précédent effectue n additions.

Exercice 2. *On rappelle l'algorithme de recherche de la première occurence d'une valeur dans un tableau :*

```
Fonction: PremierIndice(v, t)
Action: Détermination de la première occurrence d'une valeur v dans un tableau t
Début
    Pour k allant de 0 à longueur(t) − 1 faire
        Si t_k = v alors
            Renvoyer k
        FinSi
    FinPour
    Renvoyer −1
Fin
```

On note n la longueur du tableau t.

1. *Déterminer en fonction de n le nombre de comparaisons effectuées dans le meilleur des cas.*
2. *Déterminer en fonction de n le nombre de comparaisons effectuées dans le pire des cas.*

2 Complexité en mémoire d'un algorithme

Définition 2. *On appelle* complexité spatiale *d'un algorithme la fonction qui associe à la taille entière n de la donnée la quantité d'espace mémoire nécessaire à l'exécution de l'algorithme.*

Exercice 3. *On considère les algorithmes suivants :*

```
Fonction: fib(n)
Début
    l ← [1, 1]
    Pour k allant de 2 à n faire
        Ajout de l_{k−2} + l_{k−1} à la liste l
    FinPour
    Renvoyer l_n
Fin
```

```
Fonction: smartfib(n)
Début
    l ← [1, 1, 2]
    Pour k allant de 3 à n faire
        l_0 ← l_1
        l_1 ← l_2
        l_2 ← l_0 + l_1
    FinPour
    Renvoyer l_2
Fin
```

1. *Déterminer la sortie de chacun des algorithmes (justifier au moyen d'un invariant de boucle).*
2. *Déterminer en fonction de n la longueur maximale de la liste l dans chacun des algorithmes.*

3 Classes de complexité

Définition 3. *On considère deux suites réelles $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ positives.*

On dit que la suite $(u_n)_{n\in\mathbb{N}}$ est dominée *par la suite $(u_n)_{n\in\mathbb{N}}$ et on note $u_n = O(v_n)$ si il existe une constante K telle qu'à partir d'un certain rang $u_n \leqslant Kv_n$.*

Exemple 3. *On a $2n + 1 = O(n)$ car à partir du rang 1, $2n + 1 \leqslant 3n$.*

Définition 4. *On note C la complexité temporelle d'un algorithme dont la taille de la donnée est n, on définit les classes de complexité :*

- *temps constant : $C(n) = O(1)$.*
- *temps logarithmique : $C(n) = O(\ln n)$.*
- *temps linéaire : $C(n) = O(n)$.*
- *temps quadratique : $C(n) = O(n^2)$.*
- *temps polynômial : $C(n) = O(n^\alpha)$.*
- *temps exponentiel : $C(n) = O(\alpha^n)$.*

Exemple 4. *L'algorithme de recherche de la première ocurrence d'une valeur dans un tableau est en temps linéaire.*

Exercice 4. *Déterminer en fonction de n les classes de complexité des fonctions* pow *et* fastpow *de l'exercice 1.*

Exercices supplémentaires

Exercice 5. *On considère un polynôme de degré n, $P(X) = a_0 + a_1X + a_2X^2 + \cdots + a_nX^n$.*

1. *Montrer qu'il faut en principe pour x donné $\frac{n(n+1)}{2}$ multiplications pour calculer $P(x)$.*
2. *Montrer en remarquant que $P(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)))$ qu'on peut calculer $P(x)$ avec n multiplications.*
3. *Écrire les algorithmes précédents permettant de calculer $P(x)$, le polynôme P étant représenté par la liste de ses coefficients.*

Exercice 6.

On représente un vecteur de $\mathbb{R}^n$ par la liste U de ses coordonnées et une matrice de $\mathcal{M}_n(\mathbb{R})$ par la liste M de ses lignes.

1. *Évaluer le nombre d'additions et de multiplications nécessaires pour calculer le produit MU.*
2. *Écrire un algorithme permettant de calculer le produit MU.*

Exercice 7.

1. *Écrire sous forme d'algorithme une fonction* Pascal *de la variable l permettant d'obtenir à partir d'une ligne du triangle de Pascal la ligne suivante.*
2. *Utiliser cette fonction pour écrire une fonction* TriangleDePascal *de la variable n qui retourne la liste des $n+1$ premières lignes du triangle de Pascal.*
3. *Déterminer la classe de complexité temporelle de la fonction* TriangleDePascal.

Exercice 8. *On rappelle l'algorithme de recherche d'une valeur dans un tableau de nombres trié en ordre croissant :*

```
Fonction: Indice(v, t)
Action: Détermination d'un indice d'occurrence d'une valeur v dans un tableau de nombres t trié en
        ordre croissant
Début
    a ← 0
    b ← longueur(t) − 1
    TantQue a ⩽ b faire
        c ← ⌊(a+b)/2⌋
        Si t_c = v alors
            Renvoyer c
        sinon
            Si t_c < v alors
                a ← c + 1
            sinon
                b ← c − 1
            FinSi
        FinSi
    FinTantQue
    Renvoyer −1
Fin
```

1. *Montrer que $\delta = b - a$ diminue strictement à chaque itération, en déduire que l'algorithme se termine.*
2. *Montrer que la propriété « la valeur v est présente parmi les éléments de t d'indices compris entre a et b » est un invariant de boucle, en déduire que l'algorithme produit le résultat attendu.*
3. *Déterminer en fonction de la longueur n du tableau t le nombre d'itérations effectuées dans le meilleur des cas.*
4. *Montrer que $\delta = b - a$ est au moins divisé par 2 à chaque itération, en déduire que le nombre d'itérations effectuées dans le pire des cas est dominé par $\ln n$.*

Réponses

1) • On montre que la propriété « En fin d'itération, p vaut x^k et l vaut $[x, x^2, \dots, x^k]$ » est un invariant de boucle, elle est vraie pour $k = 1$ et le restera pour $k = n$, la sortie des algorithmes est donc $[x, x^2, \dots, x^n]$:

valeur de k	valeur de p	valeur de l
i	x^i	$[x, x^2, \dots, x^i]$
$i+1$	x^{i+1}	$[x, x^2, \dots, x^i, x^{i+1}]$

• Le premier algorithme effectue $\displaystyle\sum_{k=1}^{k=n}(k-1) = \frac{n(n-1)}{2}$ multiplications et le second $\displaystyle\sum_{k=1}^{k=n} 1 = n$.

2) • Dans le meilleur des cas la valeur v est présente au début du tableau et l'algorithme effectue une comparaison.

• Dans le pire des cas la valeur v n'est pas présente dans le tableau et l'algorithme effectue n comparaisons.

3) • La sortie de chacun des algorithmes est le terme u_n de la suite de Fibonacci définie par :

$$\begin{cases} u_0 &= 1 \\ u_1 &= 1 \\ u_{n+2} &= u_{n+1} + u_n \ , \ n \in \mathbb{N} \end{cases}$$

Pour le premier algorithme, dans le cas $n \geqslant 2$, on considère l'invariant de boucle « En fin d'itération, l vaut $[u_0, u_1, \dots, u_k]$ » :

valeur de k	valeur de l
i	$[u_0, u_1, \dots, u_i]$
$i+1$	$[u_0, u_1, \dots, u_i, u_{i+1-2} + u_{i+1-1}] = [u_0, u_1, \dots, u_i, u_{i+1}]$

Pour le second algorithme, dans le cas $n \geqslant 3$, on considère l'invariant de boucle « En fin d'itération, l vaut $[u_{k-2}, u_{k-1}, u_k]$ » :

valeur de k	valeur de l
i	$[u_{i-2}, u_{i-1}, u_i]$
$i+1$	$[u_{i-1}, u_i, u_{i-1} + u_i] = [u_{i-1}, u_i, u_{i+1}]$

• Dans le premier algorithme la longueur finale de la liste l est $n+1$, dans le second algorithme la longueur de la liste l est constante égale à 3.

4) La fonction *fastpow* est en temps linéaire et la fonction *pow* en temps quadratique car $\dfrac{n(n-1)}{2} \underset{+\infty}{\sim} n^2$.

5) La première méthode utilise $\displaystyle\sum_{k=1}^{k=n} k = \frac{n(n+1)}{2}$ multiplications :

```
Fonction: eval(P, x)
Action: Évaluation du polynôme P au point x
Début
    s ← 0
    Pour k allant de 0 à longueur(P) − 1 faire
        s ← s + P_k x^k
    FinPour
    Renvoyer s
Fin
```

La seconde méthode utilise $\sum_{k=1}^{k=n} 1 = n$ multiplications :

```
Fonction: fasteval(P, x)
Action: Évaluation du polynôme P au point x
Début
    s ← 0
    Pour k allant de 0 à longueur(P) − 1 faire
        s ← P_{longueur(P)−1−k} + xs
    FinPour
    Renvoyer s
Fin
```

6) Si on note $V = MU$ on a $V_i = \sum_{k=1}^{k=n} M_{ik}U_k$ pour $i \in [\![1; n]\!]$, il faut donc effectuer $n \times (n-1)$ additions et $n \times n$ multiplications.

```
Fonction: prod(M, U)
Action: Calcul du produit de la matrice carrée M et du vecteur colonne U
Début
    V ← [ ]
    Pour i allant de 0 à longueur(M) − 1 faire
        s ← 0
        Pour k allant de 0 à longueur(U) − 1 faire
            s ← s + M_{ik}U_k
        FinPour
        Ajout de s à la liste V
    FinPour
    Renvoyer V
Fin
```

7)

```
Fonction: Pascal(l)
Action: Calcul de la ligne consécutive à la ligne l dans le triangle de Pascal
Début
    L ← [ ]
    Ajout de 1 à la liste L
    Pour k allant de 0 à longueur(l) − 2 faire
        Ajout de l_k + l_{k+1} à la liste L
    FinPour
    Ajout de 1 à la liste L
    Renvoyer L
Fin
```

```
Fonction: TriangleDePascal(n)
Action: Calcul des n + 1 premières lignes du triangle de Pascal
Début
    T ← [[1]]
    Pour k allant de 0 à n − 1 faire
        Ajout de Pascal(T_k) à la liste T
    FinPour
    Renvoyer T
Fin
```

La fonction *Pascal* effectue longueur$(l) - 1$ additions donc la fonction *TriangleDePascal* effectue $\sum_{k=0}^{k=n-1} k = \frac{n(n-1)}{2}$ additions et est donc en temps quadratique.

8) • En remarquant que pour x, y entiers avec $x \leqslant y$ on a $x \leqslant \lfloor \frac{x+y}{2} \rfloor \leqslant y$, on montre que le variant de boucle entier $\delta = b - a$ diminue strictement à chaque itération, il finit donc par devenir strictement négatif et la boucle *TantQue* se termine :

valeur de a	valeur de b	valeur de c	valeur de $b-a$
x	y	$\lfloor \frac{x+y}{2} \rfloor$	$y-x$
$\lfloor \frac{x+y}{2} \rfloor + 1$	y	$\lfloor \frac{x+y}{2} \rfloor$	$y - \lfloor \frac{x+y}{2} \rfloor - 1 \leqslant y-x-1 < y-x$

valeur de a	valeur de b	valeur de c	valeur de $b-a$
x	y	$\lfloor \frac{x+y}{2} \rfloor$	$y-x$
x	$\lfloor \frac{x+y}{2} \rfloor - 1$	$\lfloor \frac{x+y}{2} \rfloor$	$\lfloor \frac{x+y}{2} \rfloor - 1 - x \leqslant y-1-x < y-x$

- Si la boucle *TantQue* se termine, deux cas se présentent lors de la dernière étape :
  - ◇ $c+1 > b$ soit $c \geqslant b$ d'où a vaut b et donc v n'était pas présente dans le tableau.
  - ◇ $c-1 < a$ soit $c \leqslant a$ d'où a vaut b et donc v n'était pas présente dans le tableau.
- Dans le meilleur des cas on obtient une occurence de la valeur v en une seule itération.
- On remarque que :
  - ◇ $y - \lfloor \frac{x+y}{2} \rfloor - 1 \leqslant y - \frac{x+y}{2} + 1 - 1 \leqslant \frac{y-x}{2}$
  - ◇ $\lfloor \frac{x+y}{2} \rfloor - 1 - x \leqslant \frac{x+y}{2} - 1 - x \leqslant \frac{y-x}{2}$

  Dans le pire des cas la valeur v n'est pas trouvée, le nombre d'itérations k vérifie $1 \leqslant \dfrac{n-1}{2^k}$ soit $k \leqslant \dfrac{\ln(n-1)}{\ln 2} \leqslant \dfrac{1}{\ln 2} \ln n$.