Chapitre n°4

Algorithmique

Exercice n°4.1 ★

Calcul du PGCD de deux entiers

On considère l'algorithme suivant permettant de calculer le PGCD de deux entiers naturels (non-nuls) n et m :

Algorithme : PGCD

```
Entrée : n, m
Sortie : p
p ⟵ n
q ⟵ m
Tant que p ≠ q faire :
    Si q < p alors :
        p ⟵ p − q
    Sinon :
        q ⟵ q − p
Retourner p
```

Nota : cet algorithme repose sur la propriété suivante : si $m \leq n$ sont deux entiers de $\mathbb{N}^*$:

$$PGCD(m, n) = PGCD\,(m, n - m)$$

Q1 Vérifier que l'algorithme `PGCD` produit bien le résultat attendu pour l'instance $n = 42$ et $m = 15$.

Q2 Montrer que la quantité $\max(\mathtt{p}, \mathtt{q})$ est un variant de boucle pour la boucle itérative conditionnelle « tant que » de l'algorithme `PGCD`.

Q3 Conclure quant à la terminaison de l'algorithme `PGCD`.

Exercice n°4.2 ★

Calcul du PGCD de deux entiers – correction

On considère l'algorithme `PGCD` défini dans l'exercice précédent. On note $\mathtt{x}_k$ le contenu de la variable `x` à l'issue de la $k^{\text{ième}}$ itération de la boucle « tant que ». Par convention, $\mathtt{x}_0$ désigne la valeur affectée à la variable `x` avant d'entrer dans la boucle « tant que ».

Q1 Montrer que la proposition :

$$\mathcal{P}_k \ : \qquad PGCD(\mathtt{p}_k, \mathtt{q}_k) = PGCD(n, m)$$

constitue un invariant de boucle pour la boucle itérative conditionnelle « tant que » de l'algorithme `PGCD`.

Q2 Conclure quant à la correction de l'algorithme `PGCD`.

Exercice n°4.3 ★★

Terminaison d'un algorithme récursif

Démonstration par récurrence faible

Dans le cas d'algorithmes récursifs suffisamment simples et ne présentant qu'un seul argument [1], il est possible d'adapter directement la preuve par récurrence (faible) rencontrée en cours de Mathématiques.

Afin d'illustrer cette démarche, on considère l'algorithme récursif suivant permettant de calculer $n!$:

Algorithme : `factorielle`

Entrée : n
Sortie : f

```
Si n = 0 alors :
    Retourner 1
Sinon :
    Retourner n × factorielle(n − 1)
```

Q1 Proposer une implémentation de cet algorithme en Python sous la forme d'une fonction `factorielle`.

Q2 Vérifier le bon fonctionnement de la fonction `factorielle` sur les instances suivantes :

- $n = 3$
- $n = 0$
- $n = 1\,500$
- $n = -2$

Commenter.

Intuitivement, on comprend que l'algorithme `factorielle` termine pour tout $n \in \mathbb{N}$.

Q3 Démontrer que pour tout $n \in \mathbb{N}$, `factorielle`(n) termine.

1. De préférence, un entier positif, mais on peut se ramener à ce cas par changement de variable.

Application au calcul d'approximations successives du nombre d'or

On considère la suite u définie par :

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

Cette suite est intimement liée au nombre d'or φ, solution positive de l'équation :

$$x^2 - x - 1 = 0$$

Q4 Déterminer l'expression analytique du nombre d'or φ puis en calculer une valeur approchée à 3 chiffres significatifs. Donner également l'expression analytique de la racine négative de l'équation proposée.

Q5 Démontrer la proposition suivante :

$$\forall n \in \mathbb{N}\,,\ 1 \leqslant u_n < \varphi$$

Q6 Démontrer que la suite u est strictement croissante.

Q7 Montrer que la suite u converge vers le nombre d'or φ.

On propose de retrouver ces résultats numériquement en calculant les termes successifs de la suite u sous Python.

Q8 Écrire une fonction récursive `phi` prenant comme argument un entier naturel n, et retournant le terme u_n de la suite u.

Q9 Proposer une suite d'instructions permettant d'afficher le nuage de points (n, u_n) pour $n \in [\![0, 10]\!]$. On affichera la limite φ de la suite u sur le même graphe. Les résultats sont-ils en accord avec l'étude faite précédemment ?

Q10 Démontrer la terminaison de la fonction `phi` par récurrence faible.

Exercice n°4.4 ★

Recherche du maximum dans une liste

On souhaite étudier l'algorithme naïf de recherche d'un maximum dans une liste `L` d'entiers et/ou de flottants :

Algorithme : recherche_max

Entrée : `L`
Sortie : `max`

```
n ⟵ longueur de la liste L
max ⟵ L(0)
Pour k dans [[0, n − 1]] faire :
    Si L(k) > max alors :
        max ⟵ L(k)
Retourner max
```

Q1 Proposer une implémentation de cet algorithme en Python.

Q2 Montrer que cet algorithme termine.

Q3 Montrer que cet algorithme est correct.

Q4 Déterminer la complexité de cet algorithme. On donnera le résultat en notation de Landau. Cette complexité dépend-elle de la position du maximum dans la liste `L` ?

Exercice n°4.5 ★

Tri stupide

Le tri stupide (« *bozo sort* » en Anglais) est un algorithme de tri pouvant être exprimé de la manière suivante en pseudo-code :

Algorithme : `tri_stupide`

Entrée : L
Sortie : T

```
n ⟵ nombre d'éléments dans la liste L
T ⟵ copie de la liste L
k ⟵ 0

Tant que k < n − 1 faire :
    Si T(k) ⩽ T(k + 1) alors :
        k ⟵ k + 1
    Sinon :
        k ⟵ 0
        mélanger aléatoirement la liste T

Retourner T
```

Q1 Cet algorithme termine-t-il ?

Q2 Déterminer la complexité de l'algorithme `tri_stupide` (exprimée en notation de Landau) dans le meilleur des cas.

Q3 Déterminer la complexité de l'algorithme `tri_stupide` dans le pire des cas.

Q4 Évaluer la complexité moyenne de l'algorithme `tri_stupide` (exprimée en notation de Landau).

Exercice n°4.6 ★★

Tri par insertion

Le tri par insertion est l'algorithme de tri utilisé par les joueurs de cartes : chaque nouvelle valeur est correctement insérée dans une liste déjà triée.

En pseudo-code, cet algorithme peut s'écrire de la manière suivante (pour une liste `L` de taille n, indicée de 0 à $n-1$) :

Algorithme : `tri_insertion`

Entrée : `L`
Sortie : `T`

```
n ⟵ nombre d'éléments dans la liste L
T ⟵ copie de la liste L
Pour i dans ⟦0, n−1⟧ faire :
    mem ⟵ T(i)
    j ⟵ i
    Tant que j > 0 et T(j−1) > mem faire :
        T(j) ⟵ T(j−1)
        j ⟵ j − 1
    T(j) ⟵ mem
Retourner T
```

Q1 Proposer une implémentation de cet algorithme en Python.

Q2 Démontrer que l'algorithme `tri_insertion` termine. Par convention, on notera $\mathtt{x}_k$ le contenu de la variable `x` à la fin de la $k^{\text{ième}}$ itération.

Q3 Démontrer que la boucle « tant que » de l'algorithme `tri_insertion` est correcte. On utilisera l'invariant de boucle $\mathcal{P}$ suivant :

$$\mathcal{P}_k \; : \qquad \forall m \in [\![\mathtt{j}_k, \mathtt{i}]\!], \mathtt{T}(m) \geqslant \mathtt{mem}$$

Q4 Démontrer que la boucle « pour » de l'algorithme `tri_insertion` est correcte. On utilisera l'invariant de boucle $\mathcal{Q}$ suivant :

$\mathcal{Q}_k$: les positions 0 à $\mathtt{i}_k$ de la liste `T` sont rangées par ordre croissant

Q5 Déterminer la complexité de l'algorithme `tri_insertion` (exprimée en notation de Landau) dans le meilleur des cas.

Q6 Déterminer la complexité de l'algorithme `tri_insertion` (exprimée en notation de Landau) dans le pire des cas.

Exercice n°4.7 ★★

Tri par sélection

Le tri par sélection est un algorithme de tri simple qui peut être décrit de la manière suivante pour une liste de taille n :

1. identifier le plus petit élément de la liste et l'échanger avec l'élément en première position ;
2. identifier le second plus petit élément de la liste et l'échanger avec l'élément en seconde position ;
3. continuer jusqu'à ce que la liste soit triée.

En pseudo-code, cet algorithme peut s'écrire de la manière suivante (pour une liste `L` de taille n, indicée de 0 à $n-1$) :

Algorithme : `tri_selection`

Entrée : `L`
Sortie : `T`

```
1 n ⟵ nombre d'éléments dans la liste L
2 T ⟵ copie de la liste L
3 Pour i dans ⟦0, n−2⟧ faire :
4     min ⟵ i
5     Pour j dans ⟦i+1, n−1⟧ faire :
6         Si T(j) < T(min) alors :
7             min ⟵ j
8     échanger le contenu des positions i et min de la liste T
9 Retourner T
```

Q1 Proposer une implémentation de cet algorithme en Python.

Q2 Démontrer que l'algorithme `tri_selection` termine. Par convention, on notera $\mathtt{x}_k$ le contenu de la variable `x` à la fin de la $k^{\text{ième}}$ itération.

Q3 Démontrer que la boucle « pour » la plus profonde (l. 5 à 7) de l'algorithme `tri_selection` est correcte. On utilisera l'invariant de boucle $\mathcal{P}$ suivant :

$$\mathcal{P}_k : \quad \forall \ell \in [\![\mathtt{i}, \mathtt{i}+k]\!], \mathtt{T}(\mathtt{j}_\ell) \geqslant \mathtt{T}(\mathtt{min}_k)$$

Q4 Démontrer que la boucle « pour » la moins profonde (l. 3 à 8) de l'algorithme `tri_selection` est correcte. On utilisera l'invariant de boucle $\mathcal{Q}$ suivant :

$\mathcal{Q}_k$: la sous-liste $\mathtt{T}(0:\mathtt{i}_k)$ est constituée par les k plus petits éléments de la liste `T` triés par valeurs croissantes

Q5 Déterminer la complexité de l'algorithme `tri_selection` (exprimée en notation de Landau) dans le meilleur des cas.

Q6 Déterminer la complexité de l'algorithme `tri_selection` dans le cas général. On ne cherchera pas à exprimer analytiquement cette complexité, mais uniquement à en déterminer une expression en notation de Landau.

Exercice n°4.8 ★★

Tri à bulles

Le tri à bulles est un algorithme de tri consistant à comparer un élément avec l'élément directement situé après lui dans la liste à trier. Si un élément est plus grand que son voisin direct, les deux éléments sont échangés et l'algorithme se poursuit – ainsi, les plus grands éléments

 Julien Lejeune (lejeune.enseignement@free.fr) – site : http://lejeune.enseignement.free.fr

migrent rapidement vers la fin de la liste, comme des bulles remontant à la surface d'un liquide (d'où son nom).

En pseudo-code, cet algorithme peut s'écrire de la manière suivante (pour une liste `L` de taille n, indicée de 0 à $n-1$) :

ALGORITHME : `tri_bulles`

Entrée : L
Sortie : T

```
1 n ⟵ nombre d'éléments dans la liste L
2 T ⟵ copie de la liste L
3 Pour i dans ⟦n−1, 1⟧ faire :
4 |   Pour j dans ⟦0, i−1⟧ faire :
5 |   |   Si T(j+1) < T(j) alors :
6 |   |   |   échanger les éléments T(j) et T(j+1)
7 Retourner T
```

Q1 Proposer une implémentation de cet algorithme en Python.

Q2 Démontrer que l'algorithme `tri_bulles` termine. Par convention, on notera $\mathtt{x}_k$ le contenu de la variable `x` à la fin de la $k^{\text{ième}}$ itération.

Q3 Démontrer que la boucle « pour » la plus profonde (l. 4 à 6) de l'algorithme `tri_selection` est correcte.. On utilisera l'invariant de boucle $\mathcal{P}$ suivant :

$$\mathcal{P}_k : \qquad \text{la position } \mathtt{T}(k) \text{ contient la plus grande valeur de la sous-liste } \mathtt{T}(0,k)$$

Q4 Démontrer que la boucle « pour » la moins profonde (l. 3 à 6) de l'algorithme `tri_bulles` est correcte. On utilisera l'invariant de boucle $\mathcal{Q}$ suivant :

$\mathcal{Q}_k$: la sous-liste $\mathtt{T}(\mathtt{n}-k : \mathtt{n}-1)$ est constituée par les k plus grands éléments de la liste `T` triés par valeurs croissantes

Q5 Déterminer la complexité de l'algorithme `tri_bulles` (exprimée en notation de LANDAU) dans le meilleur des cas.

Q6 Déterminer la complexité de l'algorithme `tri_bulles` (exprimée en notation de LANDAU) dans le pire des cas.

Q7 En déduire la complexité moyenne de l'algorithme `tri_bulles` (exprimée en notation de LANDAU).