Rapport du projet d'Algorithmique INFOB237

Esteban Bernagou
Martin Devolder
Virgile Devolder

Mai 2023

1 Exercice 1

1.1 Analyse de l'énoncé

Le 2*n lignes du fichier input est ambigu dans la consigne. Il ne nous sert pas à résoudre le problème.

1.2 Specs JML

/*
@requires n > 0; // le nombre de rangées est positif
@requires \nonnullelements(rows); // les rangées ne sont pas nulles
@requires (\forall int i; 0 ≤ i ∧ i < rows.length; rows[i] > 0); // le nombre de plantes est strictement positif pour chaque rangée
@ensures \result == null || (\exists int i; 0 ≤ i ∧ i < rows.length;\result.equals(invasive(rows[i])));
// la fonction renvoie null ou le nom d'une plante envahissante pour chaque rangée
@*/

1.3 Algorithme naïf

On utilise une liste pour stocker les fleurs ainsi qu'une Hashmap pour stocker le nombre d'occurences de ces fleurs dans notre liste.
Voir code pour plus de détails

1.4 Correction d'invariant et terminaison de clôture

1.4.1 Première boucle

$P \equiv a = a0 \land N = listFlower.size() \land counts\ vide$
$I \equiv a = a0 \land counts \mathrel{+}= plant.occurences() \mid plant \in listFlowers$

Correction de l'invariant :

Au début counts est vide à chaque itération. A chaque itération on rajoute le nombre d'occurences de la plante courante dans counts. A la fin counts contient le nombre d'occurences de plant dans listFlowers.
En vu de tout cela on obtient bien :
$P \Rightarrow I \wedge sp(S, I \wedge B) \Rightarrow I$
S : Instruction du corps de la boucle
B : Condition de terminaison de boucle

Terminaison de clôture :

$V = N \mid N$: nombre d'éléments restants à parcourir dans listFlowers
on a :
$sp(S, I \wedge B \wedge V = n) \Rightarrow V < n \mid n$: taille de listFlowers

Terminaison si :
$V = 0 \Rightarrow \neg B$
or V = V - 1 à chaque itération,
On obtient finalement : $V = 0 \equiv N = 0$

1.4.2 Deuxième boucle

$P \equiv a = a0 \wedge N =$ counts.keyset().size() $\wedge$ counts non vide
$I \equiv a = a0 \wedge$ plant.occurences() < listFlowers.size() / 2 | plant $\in$ listFlowers

Correction de l'invariant :

Au début de la boucle, tooMuch est null. Si le nombre d'occurrences de l'élément courant dans la table de hachage counts est supérieur à la moitié de la taille de la liste listFlowers, alors tooMuch prend la valeur de l'élément courant. À la fin de la boucle, tooMuch contient l'élément qui apparaît plus de la moitié du temps dans la liste listFlowers, s'il en existe un.
En vu de tout cela on obtient bien :
$P \Rightarrow I \wedge sp(S, I \wedge B) \Rightarrow I$
S : Instruction du corps de la boucle
B : Condition de terminaison de boucle

Terminaison de clôture :

$V = N \mid N$: nombre d'éléments à parcourir dans counts on a :
$sp(S, I \wedge B \wedge V = n) \Rightarrow V < n \mid n$: taille de counts

Terminaison si :
$V = 0 \Rightarrow \neg B$
or V = V - 1 à chaque itération,
On obtient finalement : $V = 0 \equiv N = 0$

1.5 Complexité de l'algorithme naïf

1.5.1 Première boucle

Complexité O(n) car on itère une fois sur chaque élément

1.5.2 Deuxième boucle

Complexité O(n) car on itère une fois sur chaque élément

1.5.3 Conclusion

O(n + n) = O(2n) = O(n) ⇒ complexité linéaire

1.6 Solution plus efficace

Cette fois nous avons choisi d'implémenter un algorithme récursif.
Voir code pour plus de détails

1.7 Specs JML algorithme efficace

Voir code

1.8 Solution "diviser pour mieux régner

Voir code

1.9 Complexité de "Diviser pour mieux régner"

Utilisons la formule récurente : $T(n) = c * T(n/d) + b * n^k$
Nombre de sous-problèmes à chaque récursion : 2
Taille de chaque sous problème : $n/2$

Donc $T(n) = 2 * T(n/2) + O(n)$ avec O(n) le coût de recherche du nombre d'occurences d'une plante dans une sous-liste

Conclusion :
k = 1, c = 2 et d = 2 et $c = d^k \iff 2 = 2^1$
Complexité de l'algorithme : $O(n^k * log(n) = O(n * log(n)$
⇒ Complexité linéarithmique

2 Exercice 2

2.1 Analyse de l'énoncé

L'énoncé est clair

2.2 Specs JML

/*
@requires $K > 0$; //Le nombre de bières maximal est strictement positif
@requires N, $M > 0$; // Le nombre de cases est strictement positif
@requires tab a[i][j], \forall i, $j > 0 \mid i < N \wedge j < M$
@ensures nBeers $\leq$ K; //Le nombre de bières bues par Franck est inférieur au nombre de bières maximum
/

2.3 Solution naïve

Le programme calcule tous les chemins qui sont possibles à partir de la position haut gauche de la matrice (0 0). On utlise alors la méthode "allPaths" qui renvoie une liste de tous les chemins possibles sous forme de liste de "int[]" contenant les coordonéees de chaque point du chemin. Pour chaque chemin possible, le programme calcule le nombre total de bières (via la méthode caclulatePaths), si le nombre total de bières collectées dépasse le nombre max de bières autorisées, la méthode renvoie "-1". Le programme garde dans une variable le meilleur nombre de bières (sans dépasser le nombre autorisé). Si un chemin est trouvé avec un nombre de bières supérieur au précédent, le meilleur nombre de bières est mis à jour. Après avoir regardé tous les chemins possibles, le programme affiche le meilleur nombre de bières possible. Si aucun chemin n'existe, le programme affiche "-1".

2.4 Correction d'invariant et terminaison de clôture

2.4.1 Boucle de calculatePaths

$P \equiv a = a0 \wedge nBeer \leq nMaxBeers \wedge$ path non null $\wedge\ nBeer = 0$
$I \equiv a = a0 \wedge \forall$ point, $nBeer \leq nMaxBeers \mid point \in path$

Correction de l'invariant :

La correction de l'invariant est vérifiée par le fait que nous incrémentons le nombre total de bières à chaque étape du parcours du chemin. De plus le nombre de bières consommé est toujours inférieur au nombre maximum et dans le cas limite nous retournous -1.
En vu de tout cela on obtient bien :
$P \Rightarrow I \wedge sp(S, I \wedge B) \Rightarrow I$
S : Instruction du corps de la boucle
B : Condition de terminaison de boucle

Terminaison de clôture :

V = le nombre de points restants à parcourir dans le chemin. La clotûre est garantie vu que ce nombre diminue à chaque itération jusqu'à atteindre 0.

$V = N \mid N$: nombre de points à parcourir dans path on a :
$sp(S, I \wedge B \wedge V = n) \Rightarrow V < n \mid n$: taille de path

Terminaison si :
$V = 0 \Rightarrow \neg B$
or $V = V - 1$ à chaque itération,
On obtient finalement : $V = 0 \equiv N = 0$

2.4.2 Première boucle de main

Ici il n'y a pas d'invariants on se contente simplment d'itérer sur un nombre de test, un entier $P \equiv a = a0 \wedge$ data not null
$I \equiv \emptyset$

Correction de l'invariant :

Pas d'invariant donc pas de correction

Terminaison de clôture :

Par contre il y a bel et bien une terminaison de clôture car la liste data décroit jusqu'à atteindre 0.
$V = N \mid N$: nombre de tests restants dans data on a :
$sp(S, I \wedge B \wedge V = n) \Rightarrow V < n \mid n$: taille de data

Terminaison si :
$V = 0 \Rightarrow \neg B$
or $V = V - 1$ à chaque itération,
On obtient finalement : $V = 0 \equiv N = 0$

2.4.3 Deuxième boucle de main

$P \equiv a = a0 \wedge BestNBeer = -1 \wedge \exists$ nBeer $\wedge$ allPaths not null $I \equiv a = a0 \wedge BestNBeer = -1$ jusqu'à l'itération qui rentre dans le if

Correction de l'invariant :

A chaque itération on vérifie si nBeer est supérieur au maximum d'avant on fait cela pour toutes les bières jusqu'à trouver la max (BestNBeer). Dans le cas limite où il n'y aurait aucun bière BestNBeer gardera sa valeur de -1.
En vu de tout cela on obtient bien :
$P \Rightarrow I \wedge sp(S, I \wedge B) \Rightarrow I$
S : Instruction du corps de la boucle
B : Condition de terminaison de boucle

Terminaison de clôture :

V = N | N : paths restants à parcourir dans allPaths on a :
sp(S, I $\wedge$ B $\wedge$ V = n) $\Rightarrow$ V < n | n : taille de allPaths

Terminaison si :
V = 0 $\Rightarrow$ $\neg$B
or V = V - 1 à chaque itération,
On obtient finalement : V = 0 $\equiv$ N = 0

2.5 Complexité de l'algorithme naif

<u>Méthode allPaths :</u>

Dans le pire des cas nous parcourons toute la grille et le nombre de possibilités dans une grille de n*m est $2^(n*m)$. Donc la complexité est : $O(2^(n*m))$ donc complexité exponentielle

<u>Méthode calculatePaths :</u>

Le nombre de point dans le plus long chemin est n + m - 1. n étant le nombre de lignes, m le nombre de colonnes et -1 car on ne compte pas notre position initiale. La complexité est donc de O(n+m). Donc une complexité linéaire.

<u>Méthode calculatePaths :</u>

La méthode main itère sur les chemins possibles donc sa complexité est de $O(2^(n+m))$ comme dit au point 1.
La complexité totale de l'algorithme est donc $2*O(2^(n+m))+O(n+m)$.
La complexité exponentielle l'emporte sur la linéaire. Donc algo de complexité exponentielle simplifiée en $O(2^(n+m))$.

2.6 Sous-structure optimale

On connait la solution pour atteindre une certaine position dans la matrice (i, j) donc on peut trouver pour (i+1, j) ou pour (i, j+1) et utiliser les résultats précédents (memoïsation) afin de résoudre le problème de manière efficace.

2.7 Caractérisez (par une équation récursive) cette sous-structure optimale

allPaths(n, i, j) =
max(allPpaths(n-nBeer, i+1,j), //descendre
allPaths(n-nBeer, j+1,i), //aller à droite
allPaths(n-nBeer, i+1, j+1) //aller en diagonale en bas à droite

Avec :
n : Le nombre maximum de bières que Frank peut boire
i : la ligne où Frank se trouve
j : la colonne où Frank se trouve
nBeer : le nombre de bières se trouvant à la position i, j

2.8 Spécifiez ce problème en tuilisant JML

Voir code

2.9 Ecrivez un algorithme basé sur le principe de programmation dynamique pour trouver le nombre maximum de bières que Frank peut boire

Voir code

2.10 Donnez la complexité de votre algorithme basé sur l'approche "diviser pour régner"

La complexité est une complexité quadratique de la forme $O(n^2)$.
En effet on peut exprimer la complexité de l'algorithme diviser pour régner sous cette forme : $T(n) = c * T(n/d) + bn^k$ avec c = 3 car on passe 3 fois sur chaque point du chemin, n/d = n/2 ce qui correspond environ à la moitié de la taille de la matrice.
Donc on a c < $d^k \iff 3 < 2^2$.
On applique alors la formule $n^k \Rightarrow O(n^2)$

3 Exercice 3