Sylvain, Raphaël
(111 124 564)

Conception et analyse d'algorithmes
IFT-3001

Travail 1

Travail présenté à
Yanick Ouellet

Département d'informatique et de génie logiciel
Univesité Laval
Hiver 2019

Sylvain, Raphaël
111 124 564

Question 1

Description textuelle

L'algorithme se base sur ces observations suivantes :

1. Zéro est l'élément absorbant de la multiplication ;
2. Le résultat est un produit d'éléments de vecteur (une multiplication) ;
3. Lorsque deux zéro sont dans le vecteur, le produit sera toujours zéro ;
4. Nous pouvons émuler la non-production d'un élément, en divisant le produit par celui-ci, pour vu qu'il ne soit pas zéro.

L'algorithme débute donc en initialisant une variable pour conserver le produit total (initialiser à 1), ainsi que deux variables "indicatices" (flags). Un pour conserver l'indice d'un élément absorbant et l'autre pour indiquer s'il y a plus d'un élément absorbant.

Puis, l'algorithme parcourt tous les éléments de notre vecteur entrant. S'il trouve pour la première fois un zéro, il affecter l'indice où cet élément se trouve dans le vecteur. S'il en trouve un deuxième il affecte la valeur bouléen indiquant plusieurs zéros à vrai.

Cette boucle terminée, il entre dans une des trois embranchements dépendanments de la valeur des variables indicatrices.

Premièrement, si la variable booléenne indiquant que plusieurs zéro ont été trouvés dans le vecteur entrant, alors nous initialisons le vecteur résultat pour qu'il ne contienne que des zéros, puisque nous sommes garantis qu'il y aura une mutiplication par zéro pour tout ces éléments.

Sinon, si la variable indiquant l'indice d'un zéro a été définie, alors tous les éléments du vecteur résultat seront à zéro, à l'exception de l'élément à cet indice qui aura le résultat total du produit.

Finalement, si aucunne des deux variables indicatrices n'a été définie, alors chaque élément du vecteur résultat sera égale au produit, diviser par l'élément à l'indice correspondant du vecteur entrant.

Analyse de l'algorithme

Puisque l'algorithme est trop complexe pour être analysé à l'aide d'une seule opératoire de base, nous allons séparer l'algorithme en quatres parties

et analyser chacune de ces parties individuellement, puis conclure sur sur l'entièreté de l'algorithme à partir de ces sous-analyses.

La taille de l'instance est n, soit la cardinalité du vecteur entrant A.

Bloc A

Le bloc A est composé de la première boucle `for` et des déclarations lui précédents.

Nous pouvons choisir comme opération de base le `!productIsAlwaysZero`, car c'est la comparaison qui est effectué le plus souvent, c'est-à-dire à chaque itération. De plus, la fonction `A.at(i)` à l'intérieur de la boucle s'effectue en temps constant.

Le nombre de fois que cette opération de base peut s'exécuter dépendant du contenu du vecteur entrant ainsi que de sa cardinalité.

Meilleur cas

En meilleur cas, les deux premiers éléments du vecteur sont 0. Si la boucle découvre deux 0, elle se courcircuite. Nous avons donc que :

$$
\begin{aligned}
& C^A_{best}(n) = 3 \\
\Rightarrow \quad & \langle \text{ Donc } \rangle \\
& 1 \leq C^A_{best}(n) = 3 \leq 5 \\
\in \quad & \langle \text{ Définition de } \Theta \text{ avec } c_1 = 1,\ c_2 = 5 \text{ et } n_0 = 0 \rangle \\
& \Theta(1)
\end{aligned}
$$

Pire cas

En pire cas, nous devons parcourir l'ensemble des éléments du vecteur entrant. À chacune des boucles, nous devons effectuer une seule comparaison, puis nous effectuons une dernière compairaison avant de sortir. Nous avons donc

$$
\begin{aligned}
& C^A_{worst}(n) \\
= \quad & \langle \text{ Définition mathématique } \rangle \\
& \textstyle\sum_0^{n-1} 1 + 1
\end{aligned}
$$

$$
\begin{aligned}
&= && \langle \text{ Première règle des sommations } \rangle \\
&(((n-1)-0+1)\times 1)+1 \\
&= && \langle \text{ Simplification } \rangle \\
&n+1 \\
&\Rightarrow && \langle \text{ Donc } \rangle \\
&\tfrac{1}{2}n \leq C^A_{worst}(n) = n+1 \leq 2n \ \forall n \geq 1 \\
&\in && \langle \text{ Définition de } \Theta \text{ avec } c_1 = \tfrac{1}{2},\ c_2 = 2 \text{ et } n_0 = 1 \ \rangle \\
&\Theta(n)
\end{aligned}
$$

Bloc B

Le bloc B est composé uniquement du premier bloc du `if`, soit :

```
B.assign(VECTOR_SIZE, 0);
```

Ce bloc est exécuté uniquement si deux zéros sont trouvés dans le vecteur entrant.

L'opération à utiliser est donc l'appel à la fonction `std::vector::assign`.

La complexité de la fonction `std::vector::assign` est linéaire en tout temps sur la valeur du premier paramètre. Dans notre cas, ce paramètre est équivalent à la cardinalité du vecteur entrant. Donc

$$
\begin{aligned}
&C^B(n) \\
&= && \langle \text{ Définition de la complexité de } C^B \ \rangle \\
&C_{assign}(n)\cdot 1 \\
&= && \langle \text{ Simplification } \rangle \\
&C_{assign}(n) \\
&\in && \langle \text{ Documentation de } \texttt{std::vector::assign} \ \rangle \\
&\Theta(n)
\end{aligned}
$$

Bloc C

Le bloc C est composé uniquement du bloc `else if`, soit :

```
B.assign(VECTOR_SIZE, 0);
B.at(indexOfAbsorbingElement) = totalProductExceptZeroes;
```

Ce bloc est exécuté uniquement si un seul zéro a été trouvé dans le vecteur entrant.

L'opération à utiliser est donc l'appel à la fonction `std::vector::assign`, puisque celle-ci à une complexité linéaire sur la valeur du premier paramètre en tout temps, alors que la fonction `std::vector::at` est constante en tout temps. Donc

$$
\begin{aligned}
& C^C(n) \\
= & \qquad \langle \text{ Définition de la complexité de } C^C \rangle \\
& C_{assign}(n) \cdot 1 \\
= & \qquad \langle \text{ Simplification } \rangle \\
& C_{assign}(n) \\
\in & \qquad \langle \text{ Documentation de } \texttt{std::vector::assign} \rangle \\
& \Theta(n)
\end{aligned}
$$

Bloc D

Le bloc D est composé de la boucle `for` situé dans le `else`.

Ce bloc est exéctuer que lorsque aucun zéro n'a été trouvé dans le vecteur entrant.

L'opération à utiliser est l'appel à la fonction `std::vector::push_back`, car elle a une complexité amortie de $\Theta(1)$ et qu'elle est appelé n fois, soit à une constante près de l'élément appeler le plus souvent. De plus, l'appel à `std::vector::at` à une complexité constante qui est moindre ou égale à la complexité de `std::vector::push_back`.

Donc

$$
\begin{aligned}
& C^D(n) \\
= & \qquad \langle \text{ Définition de la complexité de } C^D \rangle \\
& \textstyle\sum_{i=0}^{n} C_{push\_back}(n) \\
= & \qquad \langle \text{ Première règle de somation } \rangle \\
& ((n-1) - 0 + 1) \cdot C_{push\_back}(n) \\
= & \qquad \langle \text{ Simplification } \rangle \\
& n \cdot C_{push\_back}(n)
\end{aligned}
$$

$\approx$ ⟨ Abus de notation.
Définition de la complexité de `std::vector::push_back` ⟩

$n \cdot 1$

$=$ ⟨ Simplification ⟩

n

$\Rightarrow$ ⟨ Donc ⟩

$n \leq C^D(n) = n \leq n \ \forall n \geq 0$

$\in$ ⟨ Définition de Θ avec $c_1 = c_2 = 1$ et $n_0 = 0$ ⟩

$\Theta(n)$

Résultat final

L'algorithme est composé de 4 blocs. Dont un et un seul bloc entre B, C et D ne peut-être exécuter. Notons $C(n)$ le temps d'exécution de l'algorithme complet. Le bloc A est exécuté dans tous les cas.

En meilleur cas, le bloc B est executé après le bloc A. Nous avons donc :

$C_{best}(n)$

$=$ ⟨ Définition de la complexité de C_best ⟩

$C^A_{best}(n) + C^B(n)$

$=$ ⟨ Définition de $C^A_{best}(n)$ et $C^B(n)$ ⟩

$3 + C_{assign}(n)$

$\approx$ ⟨ Abus de notation.
Définition de la complexité de `std::vector::assign` ⟩

$n + 3$

$=$ ⟨ Par la règle du maximum ⟩

n

$\Rightarrow$ ⟨ Donc ⟩

$n \leq C_{best}(n) = n \leq n \ \forall n \geq 0$

$\in$ ⟨ Définition de Θ avec $c_1 = c_2 = 1$ et $n_0 = 0$ ⟩

$\Theta(n)$

Dans le pire cas, le bloc C ou D est exécuté après le bloc A. Nous avons

donc :

$$
\begin{aligned}
& C_{worst_C}(n) \\
= & \quad \langle \text{ Définition de la complexité de } C_{worst_C} \rangle \\
& C^A_{worst}(n) + C^C(n) \\
= & \quad \langle \text{ Définition de } C^A_{best}(n) \text{ et } C^C(n) \rangle \\
& n + 1 + C_{assign}(n) \\
\approx & \quad \langle \text{ Abus de notation.} \\
& \quad \text{Définition de la complexité de } \texttt{std::vector::assign} \rangle \\
& n + 1 + n \\
= & \quad \langle \text{ Simplification} \rangle \\
& 2n + 1 \\
= & \quad \langle \text{ Par la règle du maximum} \rangle \\
& 2n \\
\Rightarrow & \quad \langle \text{ Donc} \rangle \\
& n \leq C_{worst_C}(n) = 2n \leq 2n \ \forall n \geq 0 \\
\in & \quad \langle \text{ Définition de } \Theta \text{ avec } c_1 = 1, c_2 = 2 \text{ et } n_0 = 0 \rangle \\
& \Theta(n)
\end{aligned}
$$

et aussi que :

$$
\begin{aligned}
& C_{worst_D}(n) \\
= & \quad \langle \text{ Définition de la complexité de } C_{worst_D} \rangle \\
& C^A_{worst}(n) + C^D(n) \\
= & \quad \langle \text{ Définition de } C^A_{best}(n) \text{ et } C^D(n) \rangle \\
& n + 1 + n \\
= & \quad \langle \text{ Simplification} \rangle \\
& 2n + 1 \\
= & \quad \langle \text{ Par la règle du maximum} \rangle \\
& 2n \\
\Rightarrow & \quad \langle \text{ Donc} \rangle \\
& n \leq C_{worst_D}(n) = 2n \leq 2n \ \forall n \geq 0
\end{aligned}
$$

$$\in \qquad \langle \text{ Définition de } \Theta \text{ avec } c_1 = 1,\ c_2 = 2 \text{ et } n_0 = 0 \ \rangle$$
$$\Theta(n)$$

Donc, nous avons que cet algorithme est linéaire sur la cardinalité du vecteur entrant pour tous les cas.