Sylvain, Raphaël
(111 124 564)

Conception et analyse d'algorithmes
IFT-3001

Travail 2

Travail présenté à
Yanick Ouellet

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

Sylvain, Raphaël
111 124 564

Question 1

Algorithme 1

Le temps d'exécution de l'algorithme dépend de la taille de l'instance n (nombre d'élément dans le vecteur d'entrée A) et de la valeur de k générée aléatoirement.

L'opération de base est le calcul de $i \bmod n$. C'est l'opération qui peut être le plus souvent.

Il n'y a pas de pire ou de meilleur cas, car l'algorithme ne dépend pas du contenu du vecteur (à l'exception de $n = 1$, où la valeur $A[0] = 0$ où la conditionnelle s'exécutera toujours et où $A[0] = 1$ ou elle ne s'éxécutera jamais).

Désignons la valeur de $A[0]$ par a

La complexité de l'algorithme dans tous les cas est donc la suivante :

$$
\begin{aligned}
& C(n) \\
= & \qquad \langle \text{ Définition de l'algorithme } \rangle \\
& E_a\left[E_k\left[C(n,k,a)\right]\right] \\
= & \qquad \langle \text{ Étendre les espérences } \rangle \\
& \sum_a \rho(a) \sum_k \rho(k)\, C(n,k,a) \\
= & \qquad \langle \text{ Valeur de } a \text{ possible est de } [0..1] \ \rangle \\
& \sum_{a=0}^{1} \rho(a) \sum_k \rho(k)\, C(n,k,a) \\
= & \qquad \langle \text{ Valeur de } k \text{ possible est de } [0..n-1] \ \rangle \\
& \sum_{a=0}^{1} \rho(a) \sum_{k=0}^{n-1} \rho(k)\, C(n,k,a) \\
= & \qquad \langle \text{ Toutes les possibilités d'avoir une valeur } a \\
& \qquad\ \ \text{sont équiprobables. Donc } \rho(a) = \tfrac{1}{2} \ \rangle \\
& \sum_{a=0}^{1} \tfrac{1}{2} \sum_{k=0}^{n-1} \rho(k)\, C(n,k,a) \\
= & \qquad \langle \text{ Toutes les possibilités d'avoir une valeur } k \\
& \qquad\ \ \text{sont équiprobables. Donc } \rho(a) = \tfrac{1}{n} \ \rangle
\end{aligned}
$$

$$\sum_{a=0}^{1} \frac{1}{2} \sum_{k=0}^{n-1} \frac{1}{n} C\left(n,k,a\right)$$

$=$ $\langle$ Mettre en évidance les constantes $\rangle$

$$\frac{1}{2}\frac{1}{n} \sum_{a=0}^{1} \sum_{k=0}^{n-1} C\left(n,k,a\right)$$

$=$ $\langle$ Étendre partiellement la sommation de k $\rangle$

$$\frac{1}{2}\frac{1}{n} \sum_{a=0}^{1} \left(C\left(n,0,a\right) + C\left(n,1,a\right) + \sum_{k=2}^{n-1} C\left(n,k,a\right) \right)$$

$=$ $\langle$ Selon le tableau plus bas, $\forall k \geq 2 : C\left(n,k,a\right) = 0$ $\rangle$

$$\frac{1}{2}\frac{1}{n} \sum_{a=0}^{1} \left(C\left(n,0,a\right) + C\left(n,1,a\right) \right)$$

$=$ $\langle$ Étendre la sommation de a $\rangle$

$$\frac{1}{2}\frac{1}{n} \left(C\left(n,0,0\right) + C\left(n,1,0\right) + C\left(n,0,1\right) + C\left(n,1,1\right) \right)$$

$=$ $\langle$ Selon le tableau plus bas $\rangle$

$$\frac{1}{2}\frac{1}{n} \left(100n + 0 + 0 + 100n\right)$$

$=$ $\langle$ Simplification $\rangle$

$$\frac{1}{2}\frac{1}{n} \left(200n\right)$$

$=$ $\langle$ Simplification $\rangle$

$$\frac{1}{2} \left(200\right)$$

$=$ $\langle$ Simplification $\rangle$

$$100$$

$\in$ $\langle$ Définition de $\Theta\left(1\right)$ $\rangle$

$$\Theta\left(1\right)$$

Analysons $C\left(n,k,a\right)$

	$C(n,k,a)$				
a	$C(n,0,a)$	$C(n,1,a)$	$C(n,2,a)$	$C(n,i,a)$	$C(n,n,a)$
$a=0$	$100 \cdot n$	0	0	0	0
$a=1$	0	$100 \cdot n$	0	0	0

Algorithme 2

Le temps d'exécution de l'algorithme dépend de la taille de l'instance n (nombre d'élément dans le vecteur d'entrée A) et de la valeur de k générée aléatoirement.

L'opération de base est le calcul de $i \bmod n$. C'est l'opération qui peut être exécuter le plus souvent.

Il y a un meilleur et un pire cas.

Pire cas

En pire cas, la valeur à la première position du vecteur A est 0. ($A[0] = 0$)
Alors, nous entrons dans la conditionnelle `si - alors`.
La complexité de l'algorithme en pire cas sera donc la suivante :

$$
\begin{aligned}
& C_{worst}(n) \\
= & \qquad \langle \text{ Définition de l'algorithme } \rangle \\
& E_k\left[C(n,k)\right] \\
= & \qquad \langle \text{ Étendre les espérences } \rangle \\
& \sum_k \rho(k)\, C(n,k) \\
= & \qquad \langle \text{ Valeur possible de } k \text{ } \rangle \\
& \sum_{k=0}^{n-1} \rho(k)\, C(n,k) \\
= & \qquad \langle \text{ Toutes les possibilités d'Avoir une valeur de } k \\
& \qquad \text{ sont équiprobables. Donc } \rho(k) = \tfrac{1}{n} \text{ } \rangle \\
& \sum_{k=0}^{n-1} \tfrac{1}{n} C(n,k) \\
= & \qquad \langle \text{ Extraction de valeur constante de la sommation } \rangle \\
& \sum_{k=0}^{n-1} \tfrac{1}{n} C(n,k) \\
= & \qquad \langle \text{ Extraction de valeur constante de la sommation } \rangle \\
& \tfrac{1}{n} \sum_{k=0}^{n-1} C(n,k) \\
= & \qquad \langle \text{ Définition de } C(n,k) \text{ } \rangle
\end{aligned}
$$

$$\frac{1}{n}\sum_{k=0}^{n-1}\sum_{i=1}^{k^2} 1$$

$=$ ⟨ Règle de sommation ⟩

$$\frac{1}{n}\sum_{k=0}^{n-1}(k^2-1+1)\cdot 1$$

$=$ ⟨ Simplification ⟩

$$\frac{1}{n}\sum_{k=0}^{n-1}k^2$$

$=$ ⟨ Règle de sommation ⟩

$$\frac{1}{n}\frac{(n-1)(n-1+1)(2(n-1)+1)}{6}$$

$=$ ⟨ Simplification ⟩

$$\frac{1}{n}\frac{(n-1)(n)(2n-2+1)}{6}$$

$=$ ⟨ Simplification ⟩

$$\frac{(n-1)(n)(2n-1)}{6n}$$

$=$ ⟨ Simplification ⟩

$$\frac{(n-1)(2n-1)}{6}$$

$=$ ⟨ Étendre le polynôme ⟩

$$\frac{2n^2-3n+1}{6}$$

$=$ ⟨ Répartir la fraction ⟩

$$\frac{2n^2}{6}-\frac{3n}{6}+\frac{1}{6}$$

$=$ ⟨ Simplification ⟩

$$\frac{n^2}{3}-\frac{n}{2}+\frac{1}{6}$$

Calcul de $\mathcal{O}$

$$C_{worst}\left(n\right)$$

$=$ ⟨ Définition plus haut ⟩

$$\frac{n^2}{3}-\frac{n}{2}+\frac{1}{6}$$

$\leq$ ⟨ Retirer la soustraction ⟩

$$\frac{n^2}{3}+\frac{1}{6}$$

$\in$ $\langle$ Règle du maximum et définition de $\mathcal{O}$ $\rangle$

$\mathcal{O}\left(n^2\right)$

Calcul de Ω

$C_{worst}\left(n\right)$

$=$ $\langle$ Définition plus haut $\rangle$

$\frac{n^2}{3} - \frac{n}{2} + \frac{1}{6}$

$\geq$ $\langle$ Retirer l'addition d'une constante $\rangle$

$\frac{n^2}{3} - \frac{n}{2}$

$=$ $\langle$ Remettre sur un diviseur commun $\rangle$

$\frac{2n^2}{6} - \frac{3n}{6}$

$=$ $\langle$ Mettre en évidance le diviseur. $\rangle$

$\frac{1}{6}\left(2n^2 - 3n\right)$

$\geq$ $\langle$ $\forall n \geq 3$ $\rangle$

$\frac{1}{6}\left(2n^2 - n^2\right)$

$=$ $\langle$ Simplification $\rangle$

$\frac{1}{6}\left(n^2\right)$

$\in$ $\langle$ Définition de Ω $\rangle$

$\Omega\left(n^2\right)$

Définition de Θ

Puisque $C_{worst}\left(n\right) \in \Omega\left(n^2\right)$ et $C_{worst}\left(n\right) \in \mathcal{O}\left(n^2\right)$, $C_{worst}\left(n\right) \in \Theta\left(n^2\right)$

Meilleur cas

En meilleur cas, la valeur à la première position du vecteur A est 1. ($A[0] = 1$)

Alors, nous sautons complètetement la conditionnelle `si - alors`.

Puisque l'opération de base n'est jamais exécutée, la complexité de l'algorithme en meilleur cas sera donc la suivante : $C_{best}\left(n\right) = 0 \in \Theta\left(1\right)$

Cas moyen

En cas moyen, il faut calculer la moyenne sur toutes les instances de taille n.

Définition $a = A[0]$

La complexité de l'algorithme en cas moyen sera donc la suivante :

$$
\begin{aligned}
& C_{avg}(n) \\
= & \quad \langle \text{ Définition de l'algorithme } \rangle \\
& E_a E_k \left[C(n, k, a) \right] \\
= & \quad \langle \text{ Étendre l'espérence de } a \ \rangle \\
& \sum_a \rho(a)\, E_k \left[C(n, k, a) \right] \\
= & \quad \langle \text{ Valeurs possibles de } a \ \rangle \\
& \sum_{a=0}^{1} \rho(a)\, E_k \left[C(n, k, a) \right] \\
= & \quad \langle \text{ Toutes les possibilités d'avoir un valeur de } a \text{ sont équiprobables. Donc } \rho(a) = \tfrac{1}{2} \ \rangle \\
& \sum_{a=0}^{1} \tfrac{1}{2} E_k \left[C(n, k, a) \right] \\
= & \quad \langle \text{ Extraire multiplication constante } \rangle \\
& \tfrac{1}{2} \sum_{a=0}^{1} E_k \left[C(n, k, a) \right] \\
= & \quad \langle \text{ Étendre la sommation de } a \ \rangle \\
& \tfrac{1}{2} \left(E_k \left[C(n, k, 0) \right] + E_k \left[C(n, k, 1) \right] \right) \\
= & \quad \langle\ C_{worst}(n) = E_k \left[C(n, k, 0) \right] \text{ puisque } a = 0 \ \rangle \\
& \tfrac{1}{2} \left(C_{worst}(n) + E_k \left[C(n, k, 1) \right] \right) \\
= & \quad \langle\ C_{best}(n) = E_k \left[C(n, k, 1) \right] \text{ puisque } a = 1 \ \rangle \\
& \tfrac{1}{2} \left(C_{worst}(n) + C_{best}(n) \right) \\
= & \quad \langle \text{ Définition de } C_{worst}(n) \ \rangle \\
& \tfrac{1}{2} \left(\tfrac{2n^2 - 3n + 1}{6} + C_{best}(n) \right) \\
= & \quad \langle \text{ Définition de } C_{best}(n) \ \rangle \\
& \tfrac{1}{2} \left(\tfrac{2n^2 - 3n + 1}{6} + 0 \right)
\end{aligned}
$$

$=$ $\langle$ Simplification $\rangle$

$\frac{1}{2}\left(\frac{2n^2-3n+1}{6}\right)$

$=$ $\langle$ Simplification $\rangle$

$\frac{2n^2-3n+1}{12}$

Calcul de théta

Montrons que $C_{avg}\left(n\right) \in Theta\left(n^2\right)$.

Nous supposons que $C_{avg}\left(n\right) \in Theta\left(n^2\right)$, car le polynôme ressemble beaucoup au $C_{worst}\left(n\right)$.

$\lim\limits_{n->\infty} \frac{\frac{2n^2-3n+1}{12}}{n^2}$

$=$ $\langle$ Double fraction $\rangle$

$\lim\limits_{n->\infty} \frac{2n^2-3n+1}{12n^2}$

$=$ $\langle$ Règle de l'hôspital $\rangle$

$\lim\limits_{n->\infty} \frac{\left(2n^2-3n+1\right)'}{\left(12n^2\right)'}$

$=$ $\langle$ Distribuer la dérivée $\rangle$

$\lim\limits_{n->\infty} \frac{\left(2n^2\right)'-\left(3n\right)'+\left(1\right)'}{\left(12n^2\right)'}$

$=$ $\langle$ Dériver $\rangle$

$\lim\limits_{n->\infty} \frac{2 \cdot 2n-1 \cdot 3+0 \cdot 1}{2 \cdot 12n}$

$=$ $\langle$ Simplifier $\rangle$

$\lim\limits_{n->\infty} \frac{4n-3}{24n}$

$=$ $\langle$ Règle de l'hôspital $\rangle$

$\lim\limits_{n->\infty} \frac{\left(4n-3\right)'}{\left(24n\right)'}$

$=$ $\langle$ Distribuer la dérivée $\rangle$

$\lim\limits_{n->\infty} \frac{\left(4n\right)'-\left(3\right)'}{\left(24n\right)'}$

$=$ $\langle$ Dériver $\rangle$

$\lim\limits_{n->\infty} \frac{1 \cdot 4-0 \cdot 3}{1 \cdot 24}$

$=$ $\langle$ Simplifier $\rangle$

$$\lim_{n->\infty} \frac{4}{24}$$

$=$ $\langle$ Résoudre la limite $\rangle$

$$\frac{4}{24}$$

$=$ $\langle$ Simplifier $\rangle$

$$\frac{1}{6}$$

Donc $C_{avg}(n) \in \Theta(n^2)$.

Question 2

Description

Soit un menu R où pour un item x, il y a un nombre a_x d'ailes et b_x de pintes de bière pour un coût c_x d'associé.

Définition du tableau

Le tableau M contient le prix minimum.

Définition des dimensions du tableau

La première dimension va de 0 jusqu'au nombre d'item dans le menu.
La deuxième dimension va de 0 jusqu'au nombre de d'ailes commandées.
La troisième dimension va de 0 jusqu'au nombre de bières commandées.

Définition d'une cellule

La cellule $M[i, j, k]$ contient le prix minimum pour une commande de j ailes et k bières. Elle contient l'infini si cette combinaisons de j ailes et k bières est impossible.

Conditions initiales

La cellule $M[0, 0, 0] = 0$
La cellule $M[0, j, k] = \infty$ $(\forall j, k \in \mathbb{N} | j + k > 0)$

Récurrence

$$M\left[i, j, k\right] = \begin{cases} M\left[i-1, j, k\right] & \text{si } k - b_i < 0 \\ M\left[i-1, j, k\right] & \text{si } j - a_i < 0 \\ \min\left(M\left[i, j-a_i, k-b_i\right] + c_i, M\left[i-1, j, k\right]\right) & \text{sinon} \end{cases}$$

Analyse de la fonction `commande`

Le temps d'exécution de l'algorithme dépend du nombre d'item n dans le menu, le nombre d'ailes a à commandées et le nombre de pintes de bières b commandées.

Cette méthode est composée de deux appels à des fonctions. Nous analyserons donc chacune des fonctions et nous pourrons donner notre réponse selon le maximum des deux.

Analyse de la fonction `genererTableau`

Le temps d'exécution de l'algorithme dépend du nombre d'item n dans le menu, le nombre d'ailes a à commandées et le nombre de pintes de bières b commandées.

Nous devons séparer l'analyse en plusieurs blocs, puisqu'il y a des appels de fonction.

Bloc A

Le bloc A est tout ce qui se trouve à dessus des boucles `for`.

Nous avons deux appels de fonction ici. Un appel à `vector::size` et au constructeur de `Tableau::Tableau`

Ils sont tout les deux exécuté une seule fois.

L'appel à `vector::size` ce fait en tant constant $\Theta(1)$.

L'appel à `Tableau::Tableau` ce fait en temps linéaire sur le nombre de cas du tableau. Il n'y a pas de pire cas. Dans notre cas, il se fait donc en tout temps à une complexité de

$$\begin{aligned}
& \Theta\left((n+1) * (a+1) * (b+1)\right) \\
= & \qquad \langle \text{ Étendre polynôme } \rangle \\
& \Theta\left(nab + na + nb + ab + n + a + b + 1\right) \\
= & \qquad \langle \text{ Règle du maximum } \rangle \\
& \Theta\left(nab\right)
\end{aligned}$$

Donc, le bloc A a une complexité de $\Theta\left(\max\left(1 + nab\right)\right) = \Theta\left(nab\right)$

Bloc B

Le bloc B est constitué de la boucle `for` et de ces sous-boucles.

L'opération de base est la comparaison `i == 0`, car c'est l'opération exécuter le plus souvent et que tout les appels de fonction se font en temps constant, incluant `Tableau::at`.

Il n'y a pas de pire cas.

Le nombre de fois que cette opération peut être exécuter nous est données par la sommation suivante :

$$
\begin{aligned}
& C^{B}_{genererTableau}(n, a, b) \\
= & \qquad \langle \text{ Définition de la sommation selon l'algorithme } \rangle \\
& \sum_{i=0}^{n} \sum_{j=0}^{a} \sum_{k=0}^{b} 1 \\
= & \qquad \langle \text{ Règle de sommation } \rangle \\
& \sum_{i=0}^{n} \sum_{j=0}^{a} ((b - 0 + 1) \cdot 1) \\
= & \qquad \langle \text{ Simplification } \rangle \\
& \sum_{i=0}^{n} \sum_{j=0}^{a} (b + 1) \\
= & \qquad \langle \text{ Règle de sommation } \rangle \\
& \sum_{i=0}^{n} ((a - 0 + 1) \cdot (b + 1)) \\
= & \qquad \langle \text{ Simplification } \rangle \\
& \sum_{i=0}^{n} ((a + 1) \cdot (b + 1)) \\
= & \qquad \langle \text{ Règle de sommation } \rangle \\
& (n - 0 + 1) \cdot ((a + 1) \cdot (b + 1)) \\
= & \qquad \langle \text{ Simplification } \rangle \\
& (n + 1) \cdot (a + 1) \cdot (b + 1) \\
= & \qquad \langle \text{ Simplification } \rangle \\
& nab + na + nb + ab + n + a + b + 1 \\
\in & \qquad \langle \text{ Notation aymptotique } \rangle \\
& \Theta(nab + na + nb + ab + n + a + b + 1) \\
= & \qquad \langle \text{ Règle du maximum } \rangle \\
& \Theta(nab)
\end{aligned}
$$

Conclusion

Puisque le bloc A a une complexité de $\Theta(nab)$ et le bloc B une complexité de $\Theta(nab)$, la fonction genererTableau a une complexité de $\Theta(nab)$, selon la règle du maximum.
$C_{genererTableau} = \Theta(nab)$

Analyse de la fonction `solutionnerTableau`

Le temps d'exécution de l'algorithme dépend du nombre d'item n dans le menu, le nombre d'ailes a à commandées et le nombre de pintes de bières b commandées et du tableau M.

L'opération de base est la comparaison `i > 0` puisque c'est l'opération la plus exécuter et puisque les fonctions appelés sont tous de complexité constante ou constante amortie.

Il y a un meilleur et un pire cas, dépendapement du contenu du tableau M.

Meilleur cas

Le meilleur cas est quand l'élément à la position $M[n, a, b]$ a une valeur infini.

Dans ce cas, l'opération est exécutée 0 fois et la complexité est définie par : $C_{best}^{solutionnerTableau} = 0 \in \Theta(1)$

Pire cas

Le pire cas est quand un élément du menu donne des ailes gratuite, sans pintes de bières et qu'un autre donne des bières gratuite sans ailes de poulet.

Ceci fait en sorte de constuire un tableau tel que l'on ne pourra bouger que d'une case dans une dimensions à la fois.

La valeur de la complexité peut être donnée par la récurrence suivante :

$$M(n, a, b) = \begin{cases} 1 & \text{si } n == 0 \\ 1 + M(n, a, b-1) & \text{si } b > 0 \\ 1 + M(n, a-1, 0) & \text{si } b == 0 \wedge a > 1 \\ 1 + M(n-1, 0, 0) & \text{si } b == 0 \wedge a == 0 \wedge n > 1 \end{cases}$$

On peut la résoudre ainsi :

$$
\begin{aligned}
& M(n, a, b) \\
= & \qquad \langle \text{ Récurrence pour réduire } b \text{ (cas 2) } \rangle \\
& 1 + M(n, a, b-1) \\
= & \ 1 + 1 + M(n, a, b-1-1) \\
= & \ 1 + 1 + 1 + M(n, a, b-1-1-1) \\
= & \ k + M(n, a, b-k) \\
= & \ b + M(n, a, b-b) \\
= & \ b + M(n, a, 0) \\
= & \qquad \langle \text{ Récurrence pour réduire } a \text{ (cas 3) } \rangle \\
& b + 1 + M(n, a-1, 0) \\
= & \ b + 1 + 1 + M(n, a-1-1, 0) \\
= & \ b + 1 + 1 + 1 + M(n, a-1-1-1, 0) \\
= & \ b + j + M(n, a-j, 0) \\
= & \ b + a + M(n, a-a, 0) \\
= & \ b + a + M(n, 0, 0) \\
= & \qquad \langle \text{ Récurrence pour réduire } n \text{ (cas 4) } \rangle \\
& b + a + 1 + M(n-1, 0, 0) \\
= & \ b + a + 1 + 1 + M(n-1-1, 0, 0) \\
= & \ b + a + 1 + 1 + 1 + M(n-1-1-1, 0, 0) \\
= & \ b + a + i + M(n-i, 0, 0) \\
= & \ b + a + n + M(n-n, 0, 0) \\
= & \ b + a + n + M(0, 0, 0) \\
= & \qquad \langle \text{ Cas de base (cas 1) } \rangle \\
& b + a + n + 1 \\
\in & \ \Theta(b + a + n + 1) \\
= & \ \Theta(b + a + n)
\end{aligned}
$$

Donc, en pire cas, l'algorithme de solution s'execute en $\Theta(b + a + n)$

Retour sur la fonction `commande`

Puisque la complexité fonction `solutionnerTableau` est au plus de $\Theta(n + a + b)$, alors que la complexité de la fonction `genererTableau` est en $\Theta(nab)$,

Puisque $\Theta(nab) > \Theta(n + a + b) \, \forall n, a, b | n > 1 \land a > 1 \land b > 1$,

Alors, la complexité de `commande` est $\Theta(nab)$

Question 3

Description

L'algorithme commence par utiliser les vecteurs entrants pour former un vecteur d'objet. Il permettera de manipuler plus facilement les éléments.

Il tri ensuite ces objets selon leur jour d'annonce.

Un boucle s'effectue par jour. À chaque tour de boucle, on ajoute les éléments de cette journée dans un tas. Ce tas est trié par date d'échéance.

Avant de terminer la boucle, on prend l'élément avec l'échéance le plus court du tas. On défini cet objet comme étant le travail à effectué cette journée.

Analyse

Analyse de la fonction `Travail::GetJourAnnonce()`

Cette fonction n'a qu'une seule opération, qui est l'accès à la valeur de variable. Il s'effectue donc en $\Theta(1)$

Analyse de la fonction `ordonnancement::f2`

Cette fonction n'a qu'un seule opération, qui est l'appel à la fonction `Travail::GetJourAnnonce()`. Il s'effectue donc en $\Theta(1)$ comme cette fonction.

Analyse de la fonction `triParDenombrement`

Cette fonction prend un vecteur A, deux entiers délimiteurs (*lbound* et *ubound*) ainsi qu'une fonction de traduction f. La taille de l'instance est défini par $|A|$ la cadinalité du vecteur entrant et $n = ubound - lbound$ la différence entre les deux entiers délimiteurs.

Par assertion, nous avons $|A| = n$.

Nous devrons effectué une analyse en plusieurs blocs.

Bloc A

Le bloc A est composé de l'appel à `resize` sur le nouveau vecteur D.
Selon la documentation, cet appel a une complexité de $\Theta(n)$.

Bloc B

Le bloc B est composé de la boucle d'initialisation.
La complexité de cette boucle peut être donnée, en tout temps, par la sommation suivante :
$C^B(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$

Bloc C

Le bloc C est composé de la boucle de fréquence.
La complexité de cette algorithme dépend de la complexité de la fonction f.
La complexité de cette boucle peut être donnée, en tout temps, par la sommation suivante :

$$\begin{aligned}
& C^C(n) \\
= & \sum_{i=0}^{n-1} C_f(n) \\
= & C_f(n) \sum_{i=0}^{n-1} 1 \\
= & C_f(n)\, n \\
\in & \Theta(C_f(n)\, n)
\end{aligned}$$

Bloc D

Le bloc D est composé de la boucle de distribution.
La complexité de cette boucle peut être donnée, en tout temps, par la sommation suivante :
$C^D(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$

Bloc E

Le bloc E est composé de l'appel à `resize` sur le vecteur de solution S.

Selon la documentation, cet appel a une complexité de $\Theta(n)$.

Bloc F

Le bloc F est composé de la boucle de solution.

La complexité de cette algorithme dépend de la complexité de la fonction f.

La complexité de cette boucle peut être donnée, en tout temps, par la sommation suivante :

$$\begin{aligned} & C^F(n) \\ = & \sum_{i=0}^{n-1} C_f(n) \\ = & C_f(n) \sum_{i=0}^{n-1} 1 \\ = & C_f(n)\, n \\ \in & \Theta(C_f(n)\, n) \end{aligned}$$

Conclusion de l'analyse de la fonction `triParDenombrement`

La complexité de la fonction `triParDenombrement` peut être donné par le maximum des complexités des blocs précédents.

$$\begin{aligned} & C(n) \\ \in & \max\left(C^A, C^B, C^C, C^D, C^E, C^F\right) \\ = & \max\left(\Theta(n), \Theta(n), \Theta(C_f(n)\, n), \Theta(n), \Theta(n), \Theta(C_f(n)\, n)\right) \\ = & \Theta(C_f(n)\, n) \end{aligned}$$

En particulier, considérons le cas où $C_f(n) \in \Theta(n)$, la fonction `triParDenombrement` s'éxecutera donc en $\Theta(n)$.

Analyse de la fonction `ordonnancement`

Cette fonction prendre deux vecteurs entrants A et B et produit un vecteur sortant C. Par assertion, nous avons que $|A| = |B| = |C|$.

La taille de l'instance est donc $n = |A|$.

Analysons en pire cas cette fonction.

Il faudra séparer l'analyse en plusieurs blocs.

Bloc A

Le bloc A est composé de la boucle d'initialisation du vecteur d'objet de type `Travail`.

Nous pouvons prendre comme opération baromètre l'appel à la fonction `std::vector::emplace_back`.

Cette opération est appelé la plus souvent à une constante près. Toutes les autres opérations s'exécute en temps constant.

Selon la documentation, un appel à cette méthode a une complexité constante amortie.

Le nombre de fois que cette opération est exécuté peut donc être donnée par la sommation suivante :

$C^A(n) = \sum_{i=0}^{n-1} 1 = n \in \Theta(n)$

Bloc B

Le bloc B est composé de l'appel à `triParDenombrement`.

Selon l'analyse précédentes, cette fonction s'exécute en $\Theta(n)$ lors que la complexité de la fonction en paramètre est constante. Ce qui est le cas, puis que nous lui envoyons `ordonnancement::f2` qui est en $\Theta(1)$ selon$<$ l'analyse précédentes.

Donc $C^B \in \Theta(n)$

Bloc C

Le bloc C est composé de la boucle `for` contant une boucle `while`.

Il est important de noter que dans l'exécution de la méthode `ordonnancement`, le corps du `while` se fera au maximum un nombre n de fois. On peut conclure ceci, puis que nous avons une condition dans le `while` que `positionJourAnnonce < nb_travaux` et dans le corps nous avons un `positionJourAnnonce++`, alors que l'initialisation de cette variable se fait à l'extérieur de toutes les boucles.

Bloc C.while

Dans la boucle `while`, il y a plusieurs appels de fonctions :

1. `std::vector::push_back`
   $\Rightarrow \Theta(1)$

2. `std::make_tuple`
   $\Rightarrow \Theta\left(1\right)$
3. `std::vector::at`
   $\Rightarrow \Theta\left(1\right)$
4. `Travail::GetJourDu`
   $\Rightarrow \Theta\left(1\right)$
5. `std::push_heap`
   $\Rightarrow \log(n)$
6. `std::vector::begin`
   $\Rightarrow \Theta\left(1\right)$
7. `std::vector::end`
   $\Rightarrow \Theta\left(1\right)$

La complexité du bloc C.while peut donc être donnée par la sommation suivante :

$$\begin{aligned} & C^{C.while}\left(n\right) \\ = & \sum_{i=0}^{n-1}(\log(n)) \\ = & \log(n)\sum_{i=0}^{n-1}1 \\ = & n\cdot\log(n) \\ \in & \Theta\left(n\log(n)\right) \end{aligned}$$

Bloc C.for

Pour le reste de la boucle `for`, il y a plusieurs appels de fonctions :

1. `std::pop_head`
   $\Rightarrow 2\log(n)$
2. `std::vector::begin`
   $\Rightarrow \Theta\left(1\right)$
3. `std::vector::end`
   $\Rightarrow \Theta\left(1\right)$
4. `std::vector::at`
   $\Rightarrow \Theta\left(1\right)$
5. `std::get<1>`
   $\Rightarrow \Theta\left(1\right)$

6. `std::vector::back`
   $\Rightarrow \Theta(1)$
7. `std::vector::pop_back`
   $\Rightarrow \Theta(1)$
8. `Travail::GetNumero`
   $\Rightarrow \Theta(1)$

La complexité du bloc C.for peut donc être donnée par la sommation suivante :

$$\begin{aligned} & C^{C.for}(n) \\ = & \sum_{i=0}^{n-1} (2\log(n)) \\ = & 2\log(n) \sum_{i=0}^{n-1} 1 \\ = & 2n\log(n) \\ \in & \Theta(n\log(n)) \end{aligned}$$

Conclusion Bloc C

Le bloc C peut donc s'exécuter en $\Theta(\max(n\log(n), n\log(n))) = \Theta(n\log(n))$

Conclusion de l'analyse de `ordonnancement`

En prenant le maximum de chaque analyse, nous avons que
$C(n) \in \Theta(\max(n, n, n\log n)) = \Theta(n\log n)$

Question 4

Le problème est-il dans NP ?

Définition du certificat

Le certificat à la forme $((G_1, G_2), (V_1', V_2'))$ où :

- G_1 est un graphe $\langle V_1, E_1 \rangle$ où V_1 est un vecteur de sommets et où $v_i \in V_1$ ne sont pas numérotés.
- G_2 est un graphe $\langle V_2, E_2 \rangle$ où V_2 est un vecteur de sommets et où $v_i \in V_2$ ne sont pas numérotés.
- V_1' est un vecteur d'entier de longueur $|V_1|$.
- V_2' est un vecteur d'entier de longueur $|V_2|$.
- V_1' contient des valeurs entières différentes.
- V_2' contient des valeurs entières différentes.
- $|V_1'| \leq |V_2'|$
- Toutes les valeurs entières de V_1' sont aussi dans V_2'.
- Les indices d'une valeur i présente à la fois dans V_1' et V_2', représente les indices d'un noeud dans V_1 et un noeud dans V_2 ayant les mêmes arrêtes dans E_1 et E_2 respectivement.

Pseudo-code de vérification

```
Verification(g1, g2: graphe, vp1, vp2: vector<int>)
{
  // bloc A
  if vp1.count != g1.vertices.count
  {
    return false
  }

  if vp2.count != g2.vertices.count
  {
    return false
  }

  if vp1.count > vp2.count
  {
    return false
```

```
}

valeurG1 := new vector<int>
valeurG2 := new vector<int>
for i := 0 to vp1.count -1
{
  for(j = 0 to valeurG1.count - 1)
  {
    if valeurG1[j] == vp1[i]
    {
      return false /* 2 fois meme nombre dans g1*/
    }
  }
  valeurG1.add(vp1[i])
}
// Bloc B
for i := 0 to vp2.count -1
{
  for(j := 0 to valeurG1.count - 1)
  {
    if valeurG2[j] == vp2[i]
    {
      return false /* 2 fois meme nombre dans g2*/
    }
  }
  valeurG2.add(vp2[i])
}

vectorIntersect := new vector<int>
// Bloc C
for(i := 0 to valeurG1.count -1)
{
  for(j := 0 to valeurG2.count -1)
  {
    if (valeurG1[i] == valeurG2[j])
    {
      vectorIntersect.add(valeurG1[i])
    }
  }
}

if vectorIntersect.count != vp1.count)
{
  return false; /* Tout les element du plus petit graphe
                ne sont pas present dans le plus gros */
```

```
}

// Bloc D
for (i := 0 to vp1.count -1)
{
  // Bloc D.A
  label := vp1[i];
  index2 := vp2.find(label)
  vertex1 := g1.vertices[i]
  vertex2 := g2.vertices[index2]
  // Bloc D.B
  otherVertices1 = new vecor<vertex>
  for (j := 0 to vp1.edges.count -1)
  {
    if vp1.edges[j].first == vertex1
    {
      otherVertices1.add(vp1.edges[j].second)
    }
    else if vp1.edges[j].second == vertex1
    {
      otherVertices1.add(vp1.edges[j].first)
    }
  }

  //Bloc D.C
  otherVertices2 = new vector<vertex>
  for (j := 0 to vp2.edges.count -1)
  {
    if vp2.edges[j].first == vertex1
    {
      otherVertices2.add(vp2.edges[j].second)
    }
    else if vp2.edges[j].second == vertex1
    {
      otherVertices2.add(vp2.edges[j].first)
    }
  }
  if (otherVertices1.count != otherVertices2.count)
  {
    return false;
  }

  intersectCounter := 0
  // Bloc D.D
  for(j := 0 to otherVertices1.count -1)
```

```
    {
      for(k := 0 to otherVertices2.count -1)
      {
        if (otherVertices1[k] == otherVertices2[k])
        {
          intersectCounter := intersectCounter + 1
        }
      }
    }

    if intersectCounter != otherVertices1.count
    {
      return false;
    }
  }

  return true;
}
```

Analyse du pseudo-code de vérification

La fonction de vérification prend deux graphes (un vecteur de noeud et un vecteur de pair de noeud(arrêt)) et deux vecteurs d'entiers.

La taille de l'instance est donc déterminer par :

— n : la cardinalité du vecteur d'arrêt du premier graphe ;
— m : la cardinalité du vecteur d'arrêt du deuxième graphe
— x : la cardinalité du premier vecteur d'entier ;
— y : la cardinalité du deuxième vecteur d'entier.

Nous ne nous interressons qu'à la borne maximale ($\mathcal{O}$) du comportement asymptotic du pire cas.

Nous devrons faire une analyse en plusieurs blocs.

Analyse du bloc A

Remarquons que toutes les opérations du bloc A ont une complexité constante ou constante amortie.

— L'ajout dans un vecteur se fait en temps constant amorti.
— L'accès à une valeur du vecteur se fait en temps constant.
— L'accès au nombre d'élément du vecteur se fait en temps constant.
— La création d'un vecteur vide se fait en temps constant.

Remarquons aussi que si le vecteur du noeud du premier graphe ne contient pas le même nombre d'élément que le nombre d'élément dans le premier vecteur d'entier, nous arrêtons le l'éxécution de la fonction.

Nous pourons donc prendre comme opération de base la comparaison `valeurG1[j] == vp1[i]` qui est l'opération exécuté le plus souvent.

La complexité peut être donnée par la sommation suivante :

$$
\begin{aligned}
& C^A(n, m, x, y) \\
= \; & \sum_{i=0}^{x-1} \sum_{j=0}^{x-1} 1 \\
= \; & \sum_{i=0}^{x-1} x \\
= \; & x \cdot \sum_{i=0}^{x-1} 1 \\
= \; & x \cdot x \\
= \; & x^2 \\
\in \; & \mathcal{O}\left(x^2\right)
\end{aligned}
$$

Analyse du bloc B

Remarquons que toutes les opérations du bloc B ont une complexité constante ou constante amortie.

— L'ajout dans un vecteur se fait en temps constant amorti.
— L'accès à une valeur du vecteur se fait en temps constant.
— L'accès au nombre d'élément du vecteur se fait en temps constant.
— La création d'un vecteur vide se fait en temps constant.

Remarquons aussi que si le vecteur du noeud du deuxième graphe ne contient pas le même nombre d'élément que le nombre d'élément dans le deuxième vecteur d'entier, nous arrêtons le l'éxécution de la fonction.

Nous pourons donc prendre comme opération de base la comparaison `valeurG2[j] == vp2[i]` qui est l'opération exécuté le plus souvent..

La complexité peut être donnée par la sommation suivante :

$$
\begin{aligned}
& C^B(n, m, x, y) \\
= \; & \sum_{i=0}^{y-1} \sum_{j=0}^{y-1} 1 \\
= \; & \sum_{i=0}^{y-1} y \\
= \; & y \cdot \sum_{i=0}^{y-1} 1
\end{aligned}
$$

$$
\begin{aligned}
&= y \cdot y \\
&= y^2 \\
&\in \mathcal{O}\left(y^2\right)
\end{aligned}
$$

Analyse du bloc C

Remarquons que toutes les opérations du bloc C ont une complexité constante ou constante amortie.

- L'ajout dans un vecteur se fait en temps constant amorti.
- L'accès à une valeur du vecteur se fait en temps constant.
- L'accès au nombre d'élément du vecteur se fait en temps constant.
- La création d'un vecteur vide se fait en temps constant.

Nous pourons donc prendre comme opération de base la comparaison `valeurG1[i] == valeurG2[j]` qui est l'opération exécuté le plus souvent.

La complexité peut être donnée par la sommation suivante :

$$
\begin{aligned}
& C^{B}\left(n, m, x, y\right) \\
&= \sum_{i=0}^{x-1} \sum_{j=0}^{y-1} 1 \\
&= \sum_{i=0}^{y-1} x \\
&= x \cdot \sum_{i=0}^{y-1} 1 \\
&= x \cdot y \\
&\in \mathcal{O}\left(x \cdot y\right)
\end{aligned}
$$

Analyse du sous-bloc D.A

Remarquons que toutes les opérations du sous-bloc D.A ont une complexité constante, sauf l'appel à `find`.

`find` à une complexité linéaire en longueur du vecteur, en pire cas.

La complexité peut donc être donné comme suit :

$C^{D.A}\left(n, m, x, y\right) = y \in \mathcal{O}\left(y\right)$

Analyse du sous-bloc D.B

Remarquons que toutes les opérations du sous-bloc D.B ont une complexité constante ou constante ammortie.

— L'ajout dans un vecteur se fait en temps constant amorti.
— L'accès à une valeur du vecteur se fait en temps constant.
— L'accès au nombre d'élément du vecteur se fait en temps constant.
— La création d'un vecteur vide se fait en temps constant.

Nous pourons donc prendre comme opération de base la comparaison `vp1.edges[j].first == vertex1` qui est l'opération exécuté le plus souvent.

La complexité peut être donnée par la sommation suivante :

$$\begin{aligned} & C^{D.B}(n,m,x,y) \\ = & \sum_{j=0}^{n-1} 1 \\ = & n \\ \in & \mathcal{O}(n) \end{aligned}$$

Analyse du sous-bloc D.C

Remarquons que toutes les opérations du sous-bloc D.C ont une complexité constante ou constante ammortie.

— L'ajout dans un vecteur se fait en temps constant amorti.
— L'accès à une valeur du vecteur se fait en temps constant.
— L'accès au nombre d'élément du vecteur se fait en temps constant.
— La création d'un vecteur vide se fait en temps constant.

Nous pourons donc prendre comme opération de base la comparaison `vp2.edges[j].first == vertex1` qui est l'opération exécuté le plus souvent.

La complexité peut être donnée par la sommation suivante :

$$\begin{aligned} & C^{D.C}(n,m,x,y) \\ = & \sum_{j=0}^{m-1} 1 \\ = & m \\ \in & \mathcal{O}(m) \end{aligned}$$

Analyse du sous-bloc D.D

Remarquons que toutes les opérations du sous-bloc D.D ont une complexité constante.

— L'accès à une valeur du vecteur se fait en temps constant.
— L'accès au nombre d'élément du vecteur se fait en temps constant.

Nous pourons donc prendre comme opération de base la comparaison `otherVertices1[i] == otherVertices2[j]` qui est l'opération exécuté le plus souvent.

La complexité peut être donnée par la sommation suivante :

$$\begin{aligned} & C^{D.D}(n,m,x,y) \\ = & \sum_{j=0}^{n-1}\sum_{k=0}^{m-1} 1 \\ = & \sum_{j=0}^{n-1} m \\ = & m \cdot \sum_{j=0}^{n-1} 1 \\ = & m \cdot n \\ \in & \mathcal{O}(m \cdot n) \end{aligned}$$

Analyse du bloc D

Le bloc D fait appel a chacun des sous-Bloc D. ? de façon séquentielle.

En pire cas, il fera appel a chacun d'entre eux au complet.

Mon l'analyse asymptotic, nous pouvons donc prendre le sous-bloc qui a la complexité la plus élevé.

Dans notre cas, ce serait donc le sous-bloc D.D, car $\max y, n, m, m \cdot n = m \cdot m \cdot n$ si $m > 2^n > 2$.

On peut conclure que $m \cdot n >= y$ si $m > 2^n > 2$, car il y a au moins $nbNoeud - 1$ arrêt dans un graphe et y représente le nombre de noeud d'un graphe.

La complexité du bloc D peut donc être donnée par la sommation suivante :

$$\begin{aligned} & C^{D}(n,m,x,y) \\ = & \sum_{i=0}^{x-1} (m \cdot n) \\ = & (m \cdot n) \cdot \sum_{i=0}^{x-1} 1 \\ = & (m \cdot n) \cdot x \\ = & m \cdot n \cdot x \\ \in & \mathcal{O}(m \cdot n \cdot x) \end{aligned}$$

Conclusion de l'analyse

La fonction fait appel a chacun des bloc de façon séquentielle.
En pire cas, elle fera appel à chacun.
Nous aurons donc :

$$\begin{aligned}
& C\left(n, m, x, y\right) \\
= \; & \max(x^2, y^2, x \cdot y, m \cdot n \cdot x) \\
\leq \; & \qquad \langle\ \forall n, m, x, y | x > 2 \wedge y > 2 \wedge n \geq x - 1 \wedge n \geq y - 1\ \rangle \\
& m \cdot n \cdot x \\
\in \; & \mathcal{O}\left(m \cdot n \cdot x\right)
\end{aligned}$$

Puisque $\mathcal{O}\left(m \cdot n \cdot x\right)$ est polynomial, alors la fonction de vérification appartient à P.

Analyse en pire cas de l'algorithme de réduction

La fonction de réduction prend un graphe et modifie deux autres graphes pour la sortie.

La taille de l'instance est par :
— n : le nombre de sommet du graphe entrant ;
— m : la cardinalité du vecteur d'arrêt du graphe entrant.

Nous ne nous interressons qu'à la borne maximale ($\mathcal{O}$) du comportement asymptotic du pire cas.

Nous devrons faire une analyse en plusieurs blocs.

Bloc A

Le bloc A est composé de la boucle `for` et de la ligne qui suit (grapheG1).

Comptons le nombre de fois que `std::vector::emplace_back` est exécuté.

Nous pouvons prendre cette opération, car c'est elle qui a la plus grande complexité (constante amortie), en plus d'être appelé le plus souvent.

La complexité peut donc être donnée par la sommation suivante :

$$\begin{aligned}
& C_{worst}^{D}\left(n, m\right) \\
= \; & \sum_{i=0}^{n} 1 \\
= \; & n \\
\in \; & \Theta\left(n\right)
\end{aligned}$$

Bloc B

Le bloc A est composé de la copie dans le grapheG2.

La copie de la structure graphe inclue une copie d'un entier et une copie d'un vecteur.

La copie de l'entier se fait en $\Theta(1)$.

Une copie d'un vecteur est linéaire sur le nombre d'élément. Puisqu'il y a m éléments dans le vecteur d'arrêtes, la copie du vecteur aura une complexité de $\Theta(m)$.

La copie du graphe est donc en $C^{B}_{worst}(n,m)\,\Theta(m+1) = \Theta(m)$.

Concluse de l'analyse de la réduction

La réduction d'exécute donc en =

$$\begin{aligned} & C^{A}_{worst}(n,m) \\ = \; & C^{A}_{worst}(n,m) + C^{B}_{worst}(n,m) \\ \in \; & \Theta(m+n) \end{aligned}$$

CycleHamiltonien = Oui ⇒ Sous-Graphe = Oui

Pour qu'un graphe contienne un cycle hamiltonien, il faut qu'il y ait un cycle reliant tous les noeuds du graphe en ne les visitant qu'une fois.

La représentation de ce chemin représente un graphe cyclique.

Donc la compairaison avec un graphe cylique de la réduction retournera vrai, si ce graphe contient ce cycle hamiltonien.

CycleHamiltonien = Non ⇒ Sous-Graphe = Non

Pour qu'un graphe ne contienne pas un cycle hamiltonien, il faut qu'il soit impossible de relier tous les noeuds de ce graphe en ne les visiant qu'une fois.

La représentation d'un tel chemin serait un graphe cyclique.

Puisque ce cycle n'existe pas, la comparaison avec un graphe cyclique de la réduction retournera faux, si ce graphe ne contient pas de cycle hamiltonien.