UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

DEVOIR 1

PAR

JÉRÉMY BOUCHARD (BOUJ08019605)

ALEXANDRE LAROUCHE (LARA04119705)

JEAN-PHILIPPE SAVARD (SAVJ04079609)

ALEXIS VALOTAIRE (VALA09129509)

DEVOIR PRÉSENTÉ À

M. FRANÇOIS LEMIEUX

DANS LE CADRE DU COURS D'ALGORITHMIQUE (8INF433)

Note : Le code source LaTeXest disponible à l'URL suivant :

https://github.com/Mukki/algo-devoir-1

Question 1

(a)

$$4^{log_2 n} \in \omega 4^{log_4 n}$$

(b)

$$(n+1)^2 \in \Theta(n^2)$$

(c)

$$2^{n+1} \in \Theta(2^n)$$

(d)

$$n! \in o(n+1)!$$

(e)

$$2^n \in \Theta(4^{n/2})$$

(f)

$$n * log_2(n) \in o(n^2 - n)$$

(g)

$$(log_2(n))^2 \in \omega(log_2(n^2))$$

(h)

$$\frac{1}{2}^n \in o(1)$$

(i)

$$2^{100} \in \Theta(2)$$

(j)

$$log_2(n^2) \in \Theta log_4(n)$$

Question 2

(a)

Il est possible d'exprimer le pseudo-code de l'algorithme décrit dans l'énoncé de la question 2. Il est également possible de constater qu'il s'agit de l'algorithme du tri par selection. Celui-ci ce décrit de cette façon :

Algorithme 1 : Pseudo-code du numéro 2

```
   Données : T[1 ··· n]
   Résultat : T[1 ··· n] (Trié)
1  pour j ← 1 à n − 1 faire
2  |  petit ← j
3  |  pour i ← j + 1 à n faire
4  |  |  si T[i] < T[petit] alors
5  |  |  |  petit ← i
6  |  |  fin
7  |  fin
8  |  T[j] ↔ T[petit]
9  fin
```

(b)

Il est possible de calculer le temps d'exécution de cette algorithme facilement. En effet, pour chaque élément du tableau il est necessaire de parcourir le reste du tableau moins les éléments déjà triés. Donc, pour la première itération, il est nécessaire de réaliser $(n-1)$ comparaisons. Pour la deuxième itération, cela sera $(n-2)$ itérations et ainsi de suite jusqu'à ce qu'il ne reste que une seule valeur à trier. Il est donc possible de formaliser le temps d'exécution de cette manière :

$$(n-1) + (n-2) + (n-3) + ... + 1 = \sum_{i=1}^{n-1} i$$

Il ne reste qu'à trouver l'ordre de $\sum_{i=1}^{n-1} i$.

Temps d'exécution de l'algorithme de tri.

$$
\begin{aligned}
S &= \sum_{i=1}^{n-1} i \\
&= (n-1) + (n-2) + (n-3) + ... + 1 \\
2S &= (n-1) + (n-2) + (n-3) + ... + 1 \\
&\quad\ + 1 \qquad + 2 \qquad + 3 \ + ... + (n-1) \\
&= \underbrace{(n + n + n + ... + n)}_{(n-1) \text{ fois}} \\
&= (n-1)(n) \\
S &= \frac{n(n-1)}{2} \\
&= \frac{1}{2}n^2 - \frac{1}{2}n
\end{aligned}
$$

D'où $\mathcal{O}(n^2)$ □

Question 3

Pour prouver cette égalité, il est nécessaire d'utiliser la definition formelle de la notation grand $\mathcal{O}$.

Pour prouver que $\mathcal{O}(max\{f(n), g(n)\}) = \mathcal{O}(f(n) + g(n))$, on doit donc prouver que n'importe quelle fonction apartenant à un ensemble appartient aussi à l'autre ensemble.

Preuve.
Notons une fonction :

$$S(n) \in \mathcal{O}(f(n) + g(n))$$

Notons aussi les deux relations suivantes :

$$f(n) + g(n) = min\{f(n), g(n)\} + max\{f(n), g(n)\}$$

$$max\{f(n), g(n)\} \leq f(n) + g(n) \leq 2 * max\{f(n), g(n)\}$$

Il est possible d'utiliser la définition formelle pour écrire :

$$\begin{aligned} |S(n)| &\leq c * (f(n) + g(n)), n > n_1 \\ &\leq c * 2 * max\{f(n), g(n)\}, n > n_1 \\ &\leq k * max\{f(n), g(n)\}, n > n_1 \\ n_1 = 1, k &= 2 * c \end{aligned}$$

D'où : $S(n) \in \mathcal{O}(max\{f(n), g(n)\})$ □

Question 4

Temps d'exécution

Pour déterminer le temps d'exécution de l'algorithme tel qu'écrit dans l' énoncé, il faut avant tout analyser ligne par ligne le pseudo-code. Voici donc l'analyse ligne par ligne.

Ligne 1 : $d = 0$
Une affectation en informatique se fait en un temps constant $\mathcal{O}(1)$.

Ligne 2 : tant que $(X > Y)$ faire $\{...\}$
L'analyse de cette ligne est beaucoup plus difficile à faire, puisque nous ne savons pas comment la condition sortie est modifiée. Nous le saurons au moment de l'analyse du contenu de la boucle.

Avec l'analyse du contenu de la boucle, on découvre qu'elle est exécutée n fois, basée sur la différence de X et Y. Ainsi, on peut conclure qu'elle est exécutée en un temps linéaire $\mathcal{O}(n)$.

Ligne 3 : $Soustraction(X, Y)$
Il est mentionné dans l'énoncé que cette fonction est exécuté en un temps linéaire $\mathcal{O}(n)$.

Ligne 4 : $d = d + 1$
Une affectation et une addition en informatique se font en un temps constant $\mathcal{O}(1)$.

Ligne 5 : return d
Un retour en informatique se fait en un temps constant $\mathcal{O}(1)$.

Ainsi, il est possible de savoir que :

1. Le temps d'exécution du contenu de la boucle est de $n + 1$;
2. Le contenu de la boucle est exécuté 'n' fois, donc $n(n + 1)$;
3. Le reste de l'algorithme est en temps constant, donc $n(n + 1) + 1$.

En distribuant le tout, nous avons $n^2 + n + 1$. Donc le temps d'exécution est de l'ordre de $\mathcal{O}(n^2)$.

Efficacité de l'algorithme

Est-ce que l'algorithme est efficace? La réponse est non. En effet, avec l'analyse du temps d'exécution fait précédement, nous avons déterminé que dans le pire des cas, le temps d'exécution est de l'ordre de $\mathcal{O}(n^2)$.

Bien qu'un algorithme de cet ordre est généralement acceptable, il est mentionné dans l'énoncé que les variables X et Y sont de grands entiers. On peut conclure que le pire des cas se produira relativement souvent à l'exécution de l'algorithme, ce qui n'est pas vraiment acceptable.

Pour améliorer l'algorithme, il faudrait que la fonction $Soustraction(X, Y)$ soit exectuable dans un temps constant au lieu d'un temps linéaire, ce qui réduirait le temps d'exécution dans l'ordre de $\mathcal{O}(n)$, ce qui serait beaucoup plus acceptable avec de grandes valeurs d'entiers. Il serait possible de le faire en remplacant la ligne avec $X - Y$.

Question 5

a)

Temps d'exécution de la fonction fib2 dans le pire des cas :

Ligne 1 : $int\, i = 1$
Une affectation en informatique se fait en un temps constant $\mathcal{O}(1)$.
S'exécute 1 fois.

Ligne 2 : $int\, j = 0$
Une affectation en informatique se fait en un temps constant $\mathcal{O}(1)$.
S'exécute 1 fois.

Ligne 3 : $for\ (\ int\, k = 1; k <= n; k++)\{\cdots\}$
Une boucle *for* de cette forme se fait en temps linéaire $\mathcal{O}(n)$.
S'exécute n fois.

Ligne 4 : $j = j + i$
Une affectation et une addition en informatique se font en un temps constant, cependant le fait qu'elle se trouve dans une boucle *for* la rend linéaire $\mathcal{O}(n)$.
S'exécute n-1 fois.

Ligne 5 : $i = j - i$
Une affectation et une addition en informatique se font en un temps constant, cependant le fait qu'elle se trouve dans une boucle *for* la rend linéaire $\mathcal{O}(n)$.
S'exécute n-1 fois.

Ligne 6 : $return\, j$
Un retour en informatique se fait en un temps constant $\mathcal{O}(1)$.
S'exécute 1 fois.

Ainsi, il est possible de savoir que :

$$\begin{aligned} T(n) &= \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(1) \\ &= \mathcal{O}(1 + 1 + n + n + n + 1) \\ &= \mathcal{O}(n) \end{aligned}$$

Temps d'exécution de la fonction fib3 dans le pire des cas :

Ligne 1 : $int\, i = 1$
Une affectation en informatique se fait en un temps constant $\mathcal{O}(1)$.
S'exécute 1 fois.

Ligne 2 : $int\, j = 0$
Une affectation en informatique se fait en un temps constant $\mathcal{O}(1)$.
S'exécute 1 fois.

Ligne 3 : $int\, k = 0$
Une affectation en informatique se fait en un temps constant $\mathcal{O}(1)$.
S'exécute 1 fois.

Ligne 4 : $int\, h = 1$
Une affectation en informatique se fait en un temps constant $\mathcal{O}(1)$.
S'exécute 1 fois.

Ligne 5 : $while\ (n > 0)\{\cdots\}$
Dans le pire des cas, n vaudra $2^k - 1$ où $k \in \mathbb{N}$, donc de cette manière nous savons qu'il est question d'un logarithmique. $\mathcal{O}(log\, n)$.
S'exécute log(n) fois.

Ligne 6 : $if\ (n\%2)\{\cdots\}$
Une condition se fait en un temps constant, par contre le fait qu'elle se situe dans un *while* fait en sorte que le temps constant va être multiplié par le temps $\mathcal{O}(log\, n)$ de la boucle. En effet, en étant en pire cas, cette condition se doit d'être vrai en tout temps. $\mathcal{O}(1) * \mathcal{O}(log\, n)$.
S'exécute log(n)-1 fois.

Ligne 7 : $int\, t = j * h$
Une affectation et une addition en informatique se font en un temps constant, cependant le fait qu'elle se trouve dans une boucle *while* fait en sorte qu'elle se multiplie avec le temps d'exécution de cette dernière. $\mathcal{O}(1) * \mathcal{O}(log\, n)$.
S'exécute log(n)-1 fois.

Ligne 8 : $j = i * h + j * k + t$
Une affectation et une addition en informatique se font en un temps constant, cepen-

dant le fait qu'elle se trouve dans une boucle *while* fait en sorte qu'elle se multiplie avec le temps d'exécution de cette dernière. $\mathcal{O}(1) * \mathcal{O}(log\, n)$.
S'exécute log(n)-1 fois.

Ligne 9 : $i = i * k + t;$
Une affectation et une addition en informatique se font en un temps constant, cependant le fait qu'elle se trouve dans une boucle *while* fait en sorte qu'elle se multiplie avec le temps d'exécution de cette dernière. $\mathcal{O}(1) * \mathcal{O}(log\, n)$.
S'exécute log(n)-1 fois.

Ligne 10 : $int\, t = h * h$
Une affectation et une addition en informatique se font en un temps constant, cependant le fait qu'elle se trouve dans une boucle *while* fait en sorte qu'elle se multiplie avec le temps d'exécution de cette dernière. $\mathcal{O}(1) * \mathcal{O}(log\, n)$.
S'exécute log(n)-1 fois.

Ligne 11 : $h = 2 * k * h + t$
Une affectation et une addition en informatique se font en un temps constant, cependant le fait qu'elle se trouve dans une boucle *while* fait en sorte qu'elle se multiplie avec le temps d'exécution de cette dernière. $\mathcal{O}(1) * \mathcal{O}(log\, n)$.
S'exécute log(n)-1 fois.

Ligne 12 : $k = k * k + t$
Une affectation et une addition en informatique se font en un temps constant, cependant le fait qu'elle se trouve dans une boucle *while* fait en sorte qu'elle se multiplie avec le temps d'exécution de cette dernière. $\mathcal{O}(1) * \mathcal{O}(log\, n)$.
S'exécute log(n)-1 fois.

Ligne 13 : $n = n/2$
Une affectation et une addition en informatique se font en un temps constant, cependant le fait qu'elle se trouve dans une boucle *while* fait en sorte qu'elle se multiplie avec le temps d'exécution de cette dernière. $\mathcal{O}(1) * \mathcal{O}(log\, n)$.
S'exécute log(n)-1 fois.

Ligne 14 : $return\, j$
Un retour en informatique se fait en un temps constant $\mathcal{O}(1)$.
S'exécute 1 fois.

Ainsi, il est possible de savoir que :

$$\begin{aligned} T(n) &= 4 * (\mathcal{O}(1)) + \mathcal{O}(log\ n) + 8 * (\mathcal{O}(1) * \mathcal{O}(log\ n)) + \mathcal{O}(1) \\ &= \mathcal{O}(13 * (1) + 9 * (log\ n)) \\ &= \mathcal{O}(log\ n) \end{aligned}$$

b)

Nous observons que *fib1* est la fonction dont la temps d'exécution est le plus long et que la fonction *fib3* possède le temps d'exécution le plus court. Cette différence s'explique par le fait que $\phi^n \geq n \geq log\, n \;\; \forall n \in \mathbb{R}$

c)

Fonction fib1 :

Cette fonction n'utilise pas beaucoup d'espace mémoire puisqu'elle ne fait qu'une comparaison et une addition, cependant le fait qu'elle soit récursive et qu'elle possède un temps d'exécution exponentiel font en sorte qu'elle a besoin de beaucoup plus d'espace mémoire que les 2 autres fonctions. Elle doit garder en mémoire la fonction à chaque fois qu'elle fait appelle à elle-même.

Fonction fib2 :

Cette fonction utilise moins de ressource que *fib3* puisqu'elle possède moins de déclaration de variable que *fib3*.

Fonction fib3 :

Cette fonction utilise moins d'espace mémoire que *fib1* mais plus que *fib2* car cette dernière utilise déclare moins de variable que *fib3*.

d)

Voici le temps processeur nécessaire pour les valeurs de n suivantes :

	fib1 (sec)	**fib2(sec)**	**fib3 (sec)**
n=10	0.000003	0.000005	0.000016
n=100	N/A	0.000006	0.000017
n=1000	N/A	0.000014	0.000019
n=10000	N/A	0.000197	0.000019
n=100000	N/A	0.001976	0.000020
n=1000000	N/A	0.020925	0.000017*(Devrait être plus élevée)

Note : N/A = Beaucoup trop long.