POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

LOG1810
STRUCTURES DISCRÈTES

TD 6 : ALGORITHMES ET ANALYSE DE COMPLEXITÉ

E2025

SOLUTIONNAIRE

Exercice 1 :

Partie A : Vrai ou Faux

Pour chaque affirmation suivante, déterminez si elle est VRAIE ou FAUSSE. Justifiez rigoureusement votre réponse.

a) Si un algorithme A a une complexité dans le pire des cas de $O(n^2)$ et un algorithme B a une complexité dans le pire des cas de $O(n \log n)$ pour le même problème, alors l'algorithme B sera toujours plus rapide que l'algorithme A pour des entrées de taille $n > 1$.

FAUX. Les notations asymptotiques décrivent le comportement pour des n suffisamment grands. Pour des petites valeurs de n, ou à cause de constantes multiplicatives importantes cachées dans la notation O, l'algorithme A ($O(n^2)$) pourrait être plus rapide. Par exemple, si $T_A(n) = n^2$ et $T_B(n) = 1000n \log n$. Pour $n = 10$, $T_A(10) = 100$ et $T_B(10) \approx 1000 \cdot 10 \cdot 3.32 \approx 33200$. De plus, $O(n \log n)$ indique une borne supérieure, la performance réelle pourrait être meilleure, mais la comparaison "toujours plus rapide" est trop forte.

b) $5n^3 + 20n^2 \log n + 100 \in \Omega(n^3 \log n)$.

FAUX. On cherche une constante $c > 0$ telle que :

$$5n^3 + 20n^2 \log n + 100 \geq c \cdot n^3 \log n \quad \text{pour tout } n \geq k.$$

Divisons par $n^3 \log n$:

$$\frac{5}{\log n} + \frac{20}{n} + \frac{100}{n^3 \log n} \geq c.$$

Mais plus n devient grand, plus chaque terme à gauche approche 0. Donc le membre gauche devient de plus en plus petit, alors que c est constant.

Il est donc impossible de trouver un $c > 0$ satisfaisant l'inégalité. Donc $5n^3 + 20n^2 \log n + 100 \notin \Omega(n^3 \log n)$.

c) Si $f(n) \in O(g(n))$ et $g(n) \in O(h(n))$, alors $f(n) \in O(h(n))$. Cette propriété est-elle également vraie pour la notation Ω et Θ ?

VRAI pour les trois notations (O, Ω, Θ). C'est la propriété de transitivité. Pour O : Si $f(n) \leq c_1 g(n)$ pour $n \geq n_1$ et $g(n) \leq c_2 h(n)$ pour $n \geq n_2$. Alors $f(n) \leq c_1(c_2 h(n)) = (c_1 c_2) h(n)$ pour $n \geq \max(n_1, n_2)$. Soit $c = c_1 c_2$ et $n_0 = \max(n_1, n_2)$. Pour Ω : Si $f(n) \geq c_1 g(n)$ et $g(n) \geq c_2 h(n)$. Alors $f(n) \geq c_1(c_2 h(n)) = (c_1 c_2) h(n)$. Pour Θ : Si $f(n) \in \Theta(g(n))$ et $g(n) \in \Theta(h(n))$, alors $f(n) \in O(g(n))$, $f(n) \in \Omega(g(n))$, $g(n) \in O(h(n))$, $g(n) \in \Omega(h(n))$. Par transitivité de O, $f(n) \in O(h(n))$. Par transitivité de Ω, $f(n) \in \Omega(h(n))$. Donc $f(n) \in \Theta(h(n))$.

d) Un algorithme dont le temps d'exécution est $T(n) = 1000n^2 + 0.001 \cdot 2^n$ a une complexité asymptotique polynomiale.

FAUX. La complexité asymptotique est déterminée par le terme qui croît le plus rapidement. Ici, 2^n est un terme exponentiel, tandis que $1000n^2$ est polynomial. Les fonctions exponentielles (avec base > 1) croissent plus vite que n'importe quel polynôme. Donc, $T(n) \in \Theta(2^n)$, ce qui est une complexité exponentielle, non polynomiale.

e) Si $f(n) \in o(g(n))$, alors $f(n) \in O(g(n))$ mais $f(n) \notin \Theta(g(n))$.

VRAI. Dire que $f(n) \in o(g(n))$ signifie que $f(n)$ devient négligeable comparé à $g(n)$ quand n devient grand. Donc, pour tout $c > 0$, il existe n_0 tel que $f(n) < c \cdot g(n)$ pour tout $n \geq k$.
En particulier, pour un certain c, on a $f(n) \leq c \cdot g(n)$ à partir d'un certain rang, donc $f(n) \in O(g(n))$.
Mais pour être dans $\Theta(g(n))$, il faudrait aussi que $f(n) \geq c' \cdot g(n)$ pour un $c' > 0$, ce qui n'est pas possible car plus n devient grand, plus $f(n)$ est petit par rapport à $g(n)$.
Donc $f(n) \notin \Theta(g(n))$.

f) La complexité dans le meilleur des cas d'un algorithme de tri par insertion sur un tableau de n éléments est $\Theta(n)$.

VRAI. Le meilleur cas pour le tri par insertion se produit lorsque le tableau est déjà trié. Dans ce cas, la boucle interne (qui décale les éléments) ne s'exécute jamais. La boucle externe parcourt les $n - 1$ éléments (du deuxième au dernier). Pour chaque élément, seule une comparaison est effectuée pour vérifier s'il doit être inséré plus tôt. Le nombre d'opérations est donc proportionnel à n. Ainsi, la complexité est $\Theta(n)$.

Partie B : Analyse de Fonctions Simples

Déterminez la complexité O (Grand-O) la plus stricte possible pour les fonctions suivantes. Justifiez brièvement.

1) $f_a(n) = (n^3 + 3n)(2n + n^2)$

   Le premier facteur, $n^3 + 3n$, est $O(n^3)$. Le deuxième facteur, $2n + n^2$, est $O(n^2)$. Par la règle du produit, $f_a(n) \in O(n^3 \cdot n^2) = O(n^5)$.

2) $f_b(n) = (50n + \log^2 n)(n \log n + 300)$

   Le premier facteur, $50n + \log^2 n$. Puisque n domine $\log^2 n$, ce facteur est $O(n)$. Le deuxième facteur, $n \log n + 300$. Le terme $n \log n$ domine 300. Ce facteur est $O(n \log n)$. Par la règle du produit, $f_b(n) \in O(n \cdot n \log n) = O(n^2 \log n)$.

3) $f_c(n) = 2^n \cdot n! + n^{10} \cdot 3^n$

   Nous avons deux termes additionnés. Terme 1 : $2^n \cdot n!$. Terme 2 : $n^{10} \cdot 3^n$. Comparons $2^n \cdot n!$ et $n^{10} \cdot 3^n$. $n!$ croît beaucoup plus vite que 3^n et que n^{10}. Donc, $2^n \cdot n!$ est le terme dominant. Ainsi, $f_c(n) \in O(2^n \cdot n!)$.

4) $f_d(n) = \frac{n^4 + n^2 \log n}{n^2 + 1}$

   Le numérateur $N(n) = n^4 + n^2 \log n$. Le terme dominant est n^4. Donc $N(n) \in O(n^4)$. Le dénominateur $D(n) = n^2 + 1$. Le terme dominant est n^2. Donc $D(n) \in \Theta(n^2)$. Alors $f_d(n) \in O(\frac{n^4}{n^2}) = O(n^2)$.

Exercice 2 :

Soients les fonctions suivantes :

$$f(n) = \frac{n^3 + 2n^2 \log n}{\sqrt{n}} \quad \text{et} \quad g(n) = n^{2.5}.$$

Montrez formellement, en utilisant la définition de la notation grand-O, que :

$$f(n) \in O(g(n)).$$

Nous voulons montrer que $f(n) \in O(g(n))$, c'est-à-dire $f(n) \in O(n^{2.5})$. Par définition, nous devons trouver des constantes positives c et n_0 telles que pour tout $n \geq k$, $0 \leq f(n) \leq c \cdot n^{2.5}$.

Simplifions d'abord $f(n)$:

$$f(n) = \frac{n^3 + 2n^2 \log n}{n^{0.5}} = \frac{n^3}{n^{0.5}} + \frac{2n^2 \log n}{n^{0.5}} = n^{3-0.5} + 2n^{2-0.5} \log n = n^{2.5} + 2n^{1.5} \log n$$

Puisque $n \geq 1$, $n^{2.5} > 0$ et $2n^{1.5} \log n \geq 0$ (car $\log n \geq 0$ pour $n \geq 1$). Donc $f(n) \geq 0$ pour $n \geq 1$.

Nous devons montrer $n^{2.5} + 2n^{1.5} \log n \leq c \cdot n^{2.5}$. Considérons les termes pour $n \geq 1$:

1. $n^{2.5} \leq 1 \cdot n^{2.5}$.
2. $2n^{1.5} \log n$. Nous savons que pour toute base $b > 1$, $\log_b n \leq n$ pour $n \geq 1$. (Plus précisément, $\log_b n < n$ pour $n \geq n_x$ où n_x dépend de b. Par exemple, pour $b = 2$, $\log_2 n \leq n$ pour $n \geq 1$. Pour $b = e$, $\ln n \leq n$ pour $n \geq 1$). Supposons que $\log n$ est $\log_2 n$ ou $\ln n$. Dans les deux cas, $\log n \leq n$ pour $n \geq 1$. Alors, $2n^{1.5} \log n \leq 2n^{1.5} \cdot n = 2n^{1.5+1} = 2n^{2.5}$. Cette majoration est valable pour $n \geq 1$.

En additionnant ces inégalités : $f(n) = n^{2.5} + 2n^{1.5} \log n \leq 1 \cdot n^{2.5} + 2n^{2.5}$ $f(n) \leq (1+2)n^{2.5}$ $f(n) \leq 3n^{2.5}$.

Cette inégalité est valable pour $n \geq 1$ (car les majorations individuelles sont valables pour $n \geq 1$). Nous avons donc trouvé les témoins : $c = 3$ $n_0 = 1$

Pour tout $n \geq n_0 = 1$, on a $0 \leq n^{2.5} + 2n^{1.5} \log n \leq 3n^{2.5}$. Par conséquent, $f(n) \in O(n^{2.5})$, c'est-à-dire $f(n) \in O(g(n))$. $\square$

Exercice 3 :

Soit $f(n) = (n+1)!$ et $g(n) = n!$.

a) Montrez formellement que $f(n) \in \Omega(g(n))$.

Nous devons trouver des constantes positives c et k telles que pour tout $n \geq k$, $f(n) \geq c \cdot g(n)$. C'est-à-dire, $(n+1)! \geq c \cdot n!$. $(n+1) \cdot n! \geq c \cdot n!$. Puisque $n! > 0$ pour $n \geq 0$ (ou $n \geq 1$ si on considère $n!$ dans le contexte de $\mathbb{N}^*$), on peut diviser par $n!$: $n+1 \geq c$. Choisissons $c = 1$. Alors nous avons besoin que $n+1 \geq 1$, ce qui est $n \geq 0$. Si nous prenons $n_0 = 1$ (pour s'assurer que $n!$ est bien défini et positif dans le contexte usuel des complexités), alors pour $n \geq 1$, $n+1 \geq 2$. Donc, pour $c = 1$ et $n_0 = 1$, l'inégalité $n+1 \geq c$ est satisfaite. Ainsi, $(n+1)! \geq 1 \cdot n!$ pour $n \geq 1$. Donc, $f(n) \in \Omega(g(n))$ avec les témoins $c = 1, n_0 = 1$. $\square$

b) Montrez formellement que $f(n) \notin O(g(n))$.

Supposons, par l'absurde, que $f(n) \in O(g(n))$. Alors il existerait des constantes positives c et k telles que pour tout $n \geq k$, $f(n) \leq c \cdot g(n)$. C'est-à-dire, $(n+1)! \leq c \cdot n!$. $(n+1) \cdot n! \leq c \cdot n!$. Pour $n \geq n_0 \geq 0$, $n! > 0$. On peut diviser par $n!$: $n+1 \leq c$. Cette affirmation dit qu'il existe une constante c telle que pour tout $n \geq n_0$, $n+1$ est inférieur ou égal à c. Cependant, $n+1$ est une fonction qui croît indéfiniment avec n. Quel que soit le choix de c et n_0, on peut toujours trouver un $n' > \max(n_0, c-1)$ tel que $n'+1 > c$. Par exemple, choisissons $n = \max(n_0, \lceil c \rceil)$. Alors $n \geq n_0$ et $n \geq c$. Donc $n+1 > c$. Ceci contredit $n+1 \leq c$. Cette contradiction montre que notre supposition initiale était fausse. Par conséquent, $f(n) \notin O(g(n))$. $\square$

c) En utilisant les résultats de a) et b), concluez si $f(n) \in \Theta(g(n))$.

Pour que $f(n) \in \Theta(g(n))$, il faut que $f(n) \in O(g(n))$ ET $f(n) \in \Omega(g(n))$. Nous avons montré en b) que $f(n) \notin O(g(n))$. Par conséquent, $f(n) \notin \Theta(g(n))$. $\square$

Exercice 4 :

Partie 1 : Complexité de Petits Fragments

Pour chaque fragment de pseudocode suivant, déterminez sa complexité temporelle en notation Θ en fonction de n. Supposez que les opérations élémentaires (affectation, arithmétique, comparaison) prennent un temps constant.

a)

$somme \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow i$ **to** n **do**
        $somme \leftarrow somme + i \cdot j$
    end for
end for

La boucle externe s'exécute n fois (pour i de 1 à n). La boucle interne s'exécute pour j de i à n. Le nombre d'itérations est $n - i + 1$. L'opération $somme \leftarrow somme + i * j$ prend un temps constant, disons $\Theta(1)$. Le nombre total d'opérations est $\sum_{i=1}^{n}(n - i + 1) \cdot \Theta(1)$. Soit $k = n - i + 1$. Quand $i = 1, k = n$. Quand $i = n, k = 1$. La somme est $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. Donc, la complexité est $\Theta(n^2)$.

b)

$x \leftarrow n$
$y \leftarrow 0$
while $x \geq 1$ **do**
    $x \leftarrow x/3$ ▷ Division entière
    $y \leftarrow y + 1$
end while

La variable x est initialisée à n. À chaque itération de la boucle `while`, x est divisé par 3 (division entière). La boucle continue tant que $x \geq 1$. Les valeurs successives de x sont approximativement $n, n/3, n/3^2, n/3^3, \ldots, n/3^k$. La boucle s'arrête lorsque $n/3^k < 1$, c'est-à-dire $n < 3^k$, ou $k > \log_3 n$. Le nombre d'itérations est donc proportionnel à $\log_3 n$. Chaque itération prend un temps constant $\Theta(1)$. Donc, la complexité est $\Theta(\log n)$. (La base du logarithme n'affecte pas la notation Theta).

c)

$compteur \leftarrow 0$
for $i \leftarrow 1$ **to** n^2 **do**
    if i est une puissance de 2 **then**
        for $j \leftarrow 1$ **to** i **do**
            $compteur \leftarrow compteur + 1$
        end for
    end if
end for

La boucle externe s'exécute n^2 fois. La condition `If` vérifie si i est une puissance de 2. Les puissances de 2 inférieures ou égales à n^2 sont $2^0, 2^1, \ldots, 2^k$ où $2^k \leq n^2$. Cela signifie $k \leq \log_2(n^2) = 2\log_2 n$. Il y a environ $2\log_2 n$ telles puissances de 2. Pour chaque i qui est une puissance de 2 (disons $i = 2^p$), la boucle interne s'exécute $i = 2^p$ fois. Le travail total est $\sum_{p=0}^{\lfloor 2\log_2 n \rfloor} 2^p$. La plus grande valeur de i (puissance de 2) est environ n^2 (si n^2 est une puissance de 2, sinon la puissance de 2 juste avant). Soit $M = n^2$. La somme est $\sum_{p=0}^{\log_2 M} 2^p = 2^{\log_2 M + 1} - 1 = 2M - 1 = 2n^2 - 1$. Cependant, la boucle externe s'exécute n^2 fois, et la condition `If` est testée à chaque fois. Le travail de la boucle interne n'est effectué que $\log(n^2)$ fois. Le coût total est $\sum_{i=1}^{n^2}(\text{test if}) + \sum_{i=2^p \leq n^2}(\sum_{j=1}^{i} \Theta(1))$. Coût du test if : $\Theta(1)$ si on suppose qu'on peut le faire vite (sinon $\Theta(\log i)$). Supposons $\Theta(1)$. Coût total

$\approx n^2 \cdot \Theta(1) + \sum_{p=0}^{2\log n} 2^p = \Theta(n^2) + \Theta(2^{2\log n+1}) = \Theta(n^2) + \Theta(2 \cdot n^2) = \Theta(n^2)$. Le terme dominant est la somme des itérations de la boucle interne. $\sum_{p \text{ t.q. } 2^p \leq n^2} 2^p$. Soit $K = \lfloor \log_2(n^2) \rfloor$. La somme est $2^{K+1} - 1$. Si n^2 est une puissance de 2, $K = \log_2(n^2)$, la somme est $2n^2 - 1$. Si n^2 n'est pas une puissance de 2, $K < \log_2(n^2)$, mais $2^K \approx n^2/c'$ pour une constante c'. La somme est $\Theta(n^2)$. La boucle externe fait n^2 itérations. Dans la plupart, la boucle interne n'est pas exécutée. Le coût est dominé par la dernière exécution de la boucle interne (quand i est la plus grande puissance de $2 \leq n^2$, disons $i \approx n^2$). Cette boucle interne fait $\Theta(n^2)$ opérations. Mais cela ne se produit qu'une fois. La somme des 2^p est $\Theta(n^2)$. La complexité est $\Theta(n^2)$.

Partie 2 : Comparaison d'Implémentations

On considère le problème de déterminer si un tableau T de n entiers contient deux éléments distincts $T[i]$ et $T[j]$ $(i \neq j)$ dont la somme est égale à une valeur cible X.

Implémentation 1 : `BruteForcePairSum`

```
function BruteForcePairSum(T, n, X)
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if T[i] + T[j] = X then
                return true
            end if
        end for
    end for
    return false
end function
```

Implémentation 2 : `SortedPairSum` (Suppose que le tableau T est **déjà trié** par ordre croissant.)

```
function SortedPairSum(T, n, X)
    gauche ← 0
    droite ← n − 1
    while gauche < droite do
        sommeActuelle ← T[gauche] + T[droite]
        if sommeActuelle = X then
            return true
        else if sommeActuelle < X then
            gauche ← gauche + 1
        else
            droite ← droite − 1
        end if
    end while
    return false
end function
```

a) Analysez la complexité temporelle dans le pire des cas pour `BruteForcePairSum`.

`BruteForcePairSum` : La boucle externe pour i s'exécute de 0 à $n-2$, soit $n-1$ fois. La boucle interne pour j s'exécute de $i+1$ à $n-1$. Quand $i=0$, j va de 1 à $n-1$ ($n-1$ itérations). Quand $i=1$, j va de 2 à $n-1$ ($n-2$ itérations). ... Quand $i=n-2$, j va de $n-1$ à $n-1$ (1 itération). Le nombre total d'exécutions de l'instruction `If` (pire cas, la paire n'est pas trouvée ou trouvée à la fin) est $\sum_{i=0}^{n-2}(n-1-(i+1)+1) = \sum_{i=0}^{n-2}(n-1-i)$. Soit $k = n-1-i$. Quand $i=0, k=n-1$. Quand $i=n-2, k=1$. La somme est $\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$. La complexité dans le pire des cas est $\Theta(n^2)$.

b) Analysez la complexité temporelle dans le pire des cas pour `SortedPairSum`. (Rappel : le tableau est déjà trié).

`SortedPairSum` : Les pointeurs *gauche* et *droite* se rapprochent l'un de l'autre à chaque itération de la boucle `while`. Soit *gauche* augmente, soit *droite* diminue. La boucle s'arrête lorsque $gauche \geq droite$. La distance $droite - gauche$ diminue d'au moins 1 à chaque itération. Initialement, la distance est $n-1$. Dans le pire des cas, la boucle s'exécute $n-1$ fois (par exemple, si *gauche* augmente à chaque fois jusqu'à rencontrer *droite*). Chaque itération de la boucle prend un temps constant $\Theta(1)$. Donc, la complexité dans le pire des cas est $\Theta(n)$.

c) Comparez les deux algorithmes. Sous quelles conditions (taille de n, cas spécifiques des données) l'un pourrait-il être préférable à l'autre, en considérant uniquement le temps d'exécution de ces fonctions (ignorez le temps de tri pour l'Implémentation 2) ? Discutez des cas où X est très petit, très grand, ou si la paire existe tôt/tard dans les itérations.

`BruteForcePairSum` est $\Theta(n^2)$ dans le pire des cas. `SortedPairSum` est $\Theta(n)$ dans le pire des cas (si le tableau est déjà trié).

Petits n : Pour de très petits n, les constantes cachées peuvent rendre `BruteForcePairSum` plus rapide si la surcharge de la logique de `SortedPairSum` est importante, mais c'est peu probable ici car les deux sont simples. Cependant, la différence de complexité asymptotique (n^2 vs n) devient rapidement significative. `SortedPairSum` sera généralement préférable même pour des n modérément petits, à condition que le tableau soit déjà trié. Si le tri doit être effectué, la complexité du tri (typiquement $\Theta(n \log n)$) dominerait pour `SortedPairSum`. Mais la question précise "ignorez le temps de tri".

Grands n : `SortedPairSum` ($\Theta(n)$) sera nettement plus performant que `BruteForcePairSum` ($\Theta(n^2)$).

Cas spécifiques des données : - Si la paire $(T[i], T[j])$ qui somme à X est trouvée très tôt par `BruteForcePairSum` (par exemple, $T[0] + T[1] = X$), son temps d'exécution réel sera $O(1)$ dans ce meilleur cas. - Le meilleur cas pour `SortedPairSum` est aussi $O(1)$ (si $T[0] + T[n-1] = X$). - Si X est très petit (par exemple, plus petit que la somme des deux plus petits éléments du tableau trié), `SortedPairSum` le déterminera rapidement. $T[gauche] + T[droite]$ sera toujours $> X$, donc *droite* diminuera jusqu'à ce que $gauche \geq droite$. Temps $\Theta(n)$. - Si X est très grand (par exemple, plus grand que la somme des deux plus grands éléments du tableau trié), `SortedPairSum` le déterminera rapidement. $T[gauche] + T[droite]$ sera toujours $< X$, donc *gauche* augmentera. Temps $\Theta(n)$. - `BruteForcePairSum` ne tire pas avantage du fait que le tableau soit trié ou de la valeur de X pour son pire cas (sauf si la paire est trouvée tôt).

En résumé, si le tableau est déjà trié, `SortedPairSum` est asymptotiquement et pratiquement meilleur pour presque toutes les tailles de n (sauf peut-être pour $n < 3$ ou 4 où la différence est négligeable). Si le tableau n'est pas trié, le coût du tri doit être ajouté à `SortedPairSum`, ce qui le rendrait $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$. Dans ce cas, pour des n très petits, `BruteForcePairSum` ($\Theta(n^2)$ mais avec petites constantes) pourrait être plus rapide qu'un tri suivi de `SortedPairSum`.

Exercice 5 (facultatif) :

a) Ordonnez les fonctions suivantes par ordre **croissant** de complexité asymptotique. Regroupez celles qui appartiennent à la même classe de complexité Θ.

$$f_1(n) = n^{2.1} \qquad f_5(n) = n(\log n)^2$$

$$f_2(n) = 100n^2 + \frac{n^3}{\log n} \qquad f_6(n) = \sum_{i=1}^{n} i \log i$$

$$f_3(n) = \frac{n!}{(n-2)!} \qquad f_7(n) = \sqrt{n} \cdot 2^{\log_2 n}$$

$$f_4(n) = 2^{n+1} \qquad f_8(n) = 5^{\log_2 n}$$

Simplifions d'abord chaque fonction : $f_1(n) = n^{2.1}$ $f_2(n) = 100n^2 + n^3/\log n$. Le terme dominant est $n^3/\log n$. $f_3(n) = n!/(n-2)! = n(n-1) = n^2 - n \in \Theta(n^2)$. $f_4(n) = 2^{n+1} = 2 \cdot 2^n \in \Theta(2^n)$. $f_5(n) = n(\log n)^2$. $f_6(n) = \sum_{i=1}^{n}(i \log i)$. On peut borner : $\sum_{i=1}^{n} i \log i \leq \sum_{i=1}^{n} n \log n = n^2 \log n$. Aussi $\sum_{i=n/2}^{n} i \log i \geq \sum_{i=n/2}^{n} (n/2)\log(n/2) = (n/2+1)(n/2)\log(n/2) \approx (n^2/4)\log n$. Donc $f_6(n) \in \Theta(n^2 \log n)$. $f_7(n) = \sqrt{n} \cdot 2^{\log_2 n} = \sqrt{n} \cdot n = n^{1.5} = n^{3/2}$. $f_8(n) = 5^{\log_2 n}$. Soit $k = \log_2 n$, alors $n = 2^k$. $5^k = (2^{\log_2 5})^k = 2^{k \log_2 5} = (2^k)^{\log_2 5} = n^{\log_2 5}$. $\log_2 5 \approx \log_2 4 = 2$ et $\log_2 8 = 3$. $\log_2 5 \approx 2.32$. Donc $f_8(n) = n^{\log_2 5} \approx n^{2.32}$.

Liste des fonctions en Θ : $f_3(n) \in \Theta(n^2)$ $f_7(n) \in \Theta(n^{1.5})$ $f_5(n) \in \Theta(n(\log n)^2)$ $f_1(n) \in \Theta(n^{2.1})$ $f_8(n) \in \Theta(n^{\log_2 5}) \approx \Theta(n^{2.32})$ $f_6(n) \in \Theta(n^2 \log n)$ $f_2(n) \in \Theta(n^3/\log n)$ $f_4(n) \in \Theta(2^n)$

Ordre croissant (en utilisant $\log n < n^\epsilon < n^c < n^d < a^n < b^n < k!$ pour $0 < \epsilon$, $c < d$, $1 < a < b$) : 1. $f_5(n) = n(\log n)^2$ 2. $f_7(n) = n^{1.5}$ 3. $f_3(n) = n^2$ 4. $f_6(n) = n^2 \log n$ 5. $f_1(n) = n^{2.1}$ 6. $f_8(n) = n^{\log_2 5} \approx n^{2.32}$ 7. $f_2(n) = n^3/\log n$ 8. $f_4(n) = 2^n$

Regroupement par classe Θ : $\Theta(n(\log n)^2)$: $f_5(n)$ $\Theta(n^{1.5})$: $f_7(n)$ $\Theta(n^2)$: $f_3(n)$ $\Theta(n^2 \log n)$: $f_6(n)$ $\Theta(n^{2.1})$: $f_1(n)$ $\Theta(n^{\log_2 5})$: $f_8(n)$ $\Theta(n^3/\log n)$: $f_2(n)$ $\Theta(2^n)$: $f_4(n)$

Ordre final : $f_5(n) \prec f_7(n) \prec f_3(n) \prec f_6(n) \prec f_1(n) \prec f_8(n) \prec f_2(n) \prec f_4(n)$.

b) **Problème du monde réel :** Un réseau social veut suggérer des "amis potentiels" à ses utilisateurs. Pour un utilisateur donné U_0, l'algorithme doit :

(a) Parcourir tous ses amis actuels A_0.

(b) Pour chaque ami $A_i \in A_0$, parcourir tous les amis de cet ami F_i.

(c) Pour chaque "ami d'ami" $P_j \in F_i$ qui n'est pas U_0 et qui n'est pas déjà un ami direct de U_0 :

— Calculer un score basé sur le nombre d'amis communs.

(d) Retourner les 10 suggestions avec les meilleurs scores.

Supposons que chaque utilisateur a en moyenne m amis.

i) Convertissez cette description en un pseudocode ou une fonction mathématique pour le calcul du score pour **un** ami d'ami potentiel P_j.

function CalculerScoreAmisCommuns(U_0, P_j)
  $score \leftarrow 0$
  $AmisDeU0 \leftarrow$ ObtenirAmis(U_0) ▷ Coût $O(m)$
  $AmisDePj \leftarrow$ ObtenirAmis(P_j) ▷ Coût $O(m)$
  for chaque $A_k \in AmisDeU0$ **do** ▷ Boucle : m itérations
    if $A_k \in AmisDePj$ **then** ▷ Test en $O(1)$ avec set pour $AmisDePj$, ou $O(m)$ si liste
      $score \leftarrow score + 1$
    end if
  end for
  return $score$
end function

En supposant que ‘ObtenirAmis‘ retourne une structure permettant une recherche en $O(1)$ (comme un ensemble de hachage) ou que les listes d'amis sont triées pour permettre une intersection en $O(m)$, la fonction ‘CalculerScoreAmisCommuns‘ est $O(m)$.

i) Analysez et justifiez la complexité temporelle (pire cas) pour générer et trier les suggestions pour l'utilisateur U_0. Exprimez la complexité en fonction de m.

Génération des scores : 1. Obtenir les amis de U_0, A_0 : $O(m)$. 2. Initialiser une structure pour stocker les scores des suggestions potentielles (par exemple, une table de hachage $ScoresPotentiels$). 3. Pour chaque ami $A_i \in A_0$ (boucle externe, m itérations) : a. Obtenir les amis de A_i, F_i : $O(m)$. b. Pour chaque $P_j \in F_i$ (boucle interne, m itérations) : i. Vérifier si $P_j \neq U_0$ et $P_j \notin A_0$. Si A_0 est une table de hachage, ce test est en moyenne $O(1)$. ii. Si P_j est un candidat valide : Calculer le score de P_j (nombre d'amis communs avec U_0) en appelant une fonction comme `CalculerScoreAmisCommuns`(U_0, P_j). Cela prend $O(m)$. Stocker/mettre à jour le score de P_j dans $ScoresPotentiels$.

Le nombre total de paires (A_i, P_j) est $m \times m = m^2$. Pour chaque P_j (qui peut apparaître plusieurs fois si c'est un ami de plusieurs amis de U_0), on calcule le score. Une approche plus efficace est de d'abord collecter tous les amis d'amis potentiels uniques, puis de calculer le score pour chacun.

Collecte des amis d'amis potentiels (S'_{AA}) : - Itérer sur les m amis de U_0. - Pour chaque ami, itérer sur ses m amis. Total $O(m^2)$ opérations pour lister tous les amis d'amis (avec duplicatas). - Stocker ces amis d'amis dans une structure qui gère les duplicatas et permet le filtrage (e.g., table de hachage pour compter les occurrences, ou simplement un ensemble pour les uniques). La création de cet ensemble S'_{AA} des amis d'amis uniques (après filtrage de U_0 et A_0) peut prendre $O(m^2)$ dans le pire des cas (si tous les m^2 sont distincts et nécessitent des insertions). $|S'_{AA}|$ est $O(m^2)$.

Calcul des scores pour chaque $P_j \in S'_{AA}$: - Pour chacun des $O(m^2)$ candidats P_j, calculer le score (nombre d'amis communs) prend $O(m)$. - Coût total pour cette étape : $O(m^2 \cdot m) = O(m^3)$.

Tri et sélection des 10 meilleures suggestions : Nous avons $O(m^2)$ scores potentiels. - Pour trouver les 10 meilleurs, on peut utiliser un tas-min de taille 10. On itère sur les $O(m^2)$ scores. Pour chaque score, on le compare au minimum du tas. Si le score est plus grand, on retire le minimum et on insère le nouveau score. Chaque opération sur le tas est $O(\log 10) = O(1)$. Donc, cette étape prend $O(m^2 \cdot 1) = O(m^2)$. - Alternativement, si on trie tous les $O(m^2)$ scores, cela prend $O(m^2 \log(m^2)) = O(m^2 \cdot 2 \log m) = O(m^2 \log m)$.

Complexité Totale (pire cas) : La génération des scores est $O(m^3)$. La sélection des 10 meilleurs est $O(m^2)$ ou $O(m^2 \log m)$. La complexité globale est dominée par la génération des scores, donc $\Theta(m^3)$.

Feuille supplémentaire