Calculabilité

TP8

Y. Deville
C-H. Bertrand Van Ouytsel - V. Coppé - A. Gerniers
N. Golenvaux - M. Parmentier

Avril 2021

Questions du test

1. Quelle est la complexité de cet algorithme ?

```
sum(n)
   sum = n * n * n
      for i = 0 to n
         for j = 0 to i
            sum++
```

Réponse : $T(n) = 1 + \frac{1}{2}n(n+1) \Longrightarrow \mathcal{O}(n^2)$

Questions du test

2. Quelle est la complexité de cet algorithme ?

```
more_fun(n)
  if (n > 1)
    return 2 * more_fun(n/2)
  else
    return n
```

Réponse :

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= T\left(\frac{n}{2}\right) + 1 \\
&= T\left(\frac{n}{2^k}\right) + k \qquad \frac{n}{2^k} = 1 \Rightarrow k = \log_2(n) \\
&= 1 + \log_2(n)
\end{aligned}
$$

$$\Longrightarrow \mathcal{O}\big(\ln(n)\big)$$

Questions du test

3. Quelle est la complexité de cet algorithme ?

```
fun(n)
  if (n > 1)
    return fun(n/2) + fun(n/2)
  else
    return n
```

Réponse :

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + 1 \\
&= 2^k \cdot T\left(\frac{n}{2^k}\right) + 2^k - 1 \qquad \frac{n}{2^k} = 1 \Rightarrow k = \log_2(n) \\
&= 2n - 1
\end{aligned}
$$

$$\Longrightarrow \mathcal{O}(n)$$

Questions du test

4. Quelle est la complexité de cet algorithme ?

```
explosion(n)
  if (n > 1)
    return explosion(n-1) + explosion(n-2)
  else
    return n+1
```

Réponse : On sait que $T(0) = 1$ and $T(1) = 1$. Pour tout $n \in \mathbb{N}$, on a la relation $T(n) = T(n-1) + T(n-2) + 1$. On peut calculer d'abord à la main les termes suivants : 1, 1, 3, 5, 9, 15, 25, ... Les éléments $T(n)$ de cette suite sont tous tels que $\mathcal{F}(n) \leq T(n) \leq 3^n$ pour tout $n \in \mathbb{N}$ (où $\mathcal{F}(n)$ est la suite de Fibonacci)[1]. $\Longrightarrow \mathcal{O}(3^n)$

(Pour info : $T(n) = \mathcal{F}(n) + \mathcal{L}(n) - 1$ où $\mathcal{L}(n)$ est la suite de Lucas (une variante de Fibonacci qui commence par $1, 3, ...$)).

1. Cela se prouve directement par récurrence.

Questions du test

5. L'égalité suivante est-elle vraie ?

$$\mathcal{O}\big(\log_2(x^2)\big) = \mathcal{O}\big(\log_{10}(x^{10})\big)$$

Réponse : On a $\log_2(x^2) = 2 \cdot \log_2(x) = 2 \cdot \frac{\ln(x)}{\ln(2)}$ et $\log_{10}(x^{10}) = 10 \cdot \log_{10}(x) = 10 \cdot \frac{\ln(x)}{\ln(10)}$.
Les facteurs constants pouvant être ignorés, on est bien en $\mathcal{O}\big(\ln(x)\big)$ dans les 2 cas.

Questions du test

6. Si on a deux fonctions f, g de $\mathbb{N}$ dans $\mathbb{N}$ telles que pour tout $n \in \mathbb{N}$, $g(n) \neq 0$ et $g(n) \in O(f(n))$.
   La limite suivante existe-t-elle nécessairement ?

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Réponse : Faux ! Prenons $f(n) = 2$ et

$$g(n) = \begin{cases} 1 & \text{si n impair} \\ 2 & \text{si n pair} \end{cases}$$

Pour tout $n \in \mathbb{N}$, $g(n) \neq 0$ et $g(n) \leq f(n)$ donc $g(n) \in O(f(n))$. Or $\frac{f(n)}{g(n)}$ oscille entre 1 et 2 donc la limite n'existe pas.

Questions du test

7. Vrai ou faux :
   Puisque les matrices sont des objets à deux dimensions (lignes et colonnes), l'algorithme élémentaire de multiplication matricielle (de deux matrices carrées de taille n) a une complexité dans $O(n^2)$.

   Réponse : Faux ! Les matrices sont des objets à deux dimensions **mais** le calcul de chaque élément du résultat d'une multiplication matricielle implique $2n$ termes et nécessite $2n-1$ opérations ! Comme il y a en tout n^2 éléments à calculer, cet algoritme a donc une complexité dans $O(n^3)$.

Questions du test

8. Vrai ou faux :
   Un problème résolu par un algorithme dont la complexité est dans $O(2^n)$ est intrinsèquement complexe.

   Réponse : Faux ! Un tel problème **n'est pas forcément** intrinsèquement complexe. La complexité d'un problème est la complexité de l'algorithme **le plus efficace** résolvant ce problème. Il est tout à fait possible d'écrire un programme dans $O(2^n)$ qui résoud un problème dans $O(n^2)$ mais il ne serait pas le plus efficace.

Questions du test

9. Vrai ou faux :
   Il est toujours impossible de trouver une solution à un problème intrinsèquement complexe en un temps réaliste quelque soit la donnée.

   Réponse : Faux ! Pour des **petits exemples**, il est possible de trouver une solution à un problème intrinsèquement complexe en un temps réaliste. De plus, il est même possible que pour **certains** grands exemples, une solution puisse également être trouvée en un temps réaliste.

Questions du test

10. Vrai ou faux :
    Comme la loi de Moore prédit que la progression de la puissance de calcul des ordinateur suit une progression exponentielle, les problèmes intrinsèquement complexes pourront être facilement résolus dans quelques dizaines d'années.

    Réponse : Faux ! Alors que la progression de la puissance de calcul permet d'améliorer la résolution de problèmes polynomiaux d'un facteur **multiplicatif**, elle n'améliore la résolution des problèmes intrinsèquement complexes que d'un facteur **additif**.
    Peu importe les améliorations technologiques, **un problème intrinsèquement complexe ne pourra être résolu que pour des exemples de petites tailles**.

Question 1 du TP

1. Pourquoi ne définit-on généralement pas la complexité d'un algorithme comme le nombre d'opérations ou le temps d'exécution dans le *meilleur des cas* ?
2. Pourquoi ne définit-on généralement pas la complexité d'un algorithme comme la *moyenne* du nombre d'opérations ou du temps d'exécution ?

Pour les deux questions ci-dessus, donner au moins une raison théorique et une raison pratique pour justifier votre réponse.

Question 1 du TP

1. Pourquoi ne définit-on généralement pas la complexité d'un algorithme comme le nombre d'opérations ou le temps d'exécution dans le *meilleur des cas* ?

Réponse : La meilleur cas peut être vu comme une borne inférieure de la complexité d'un algorithme. Mais cette borne ne nous permet pas de déduire la complexité des autres cas, et donc la classe de complexité (ni même la terminaison) de l'algorithme. Un algorithme dont le meilleur cas est $\mathcal{O}(n)$ pourrait donc très bien être intrinsèquement complexe, i.e. $\mathcal{O}(2^n)$, pour la majorité des cas. Si on considère le meilleur des cas, il n'y a en fait même plus aucun problème intrinsèquement complexe, puisqu'il est toujours possible de créer un algorithme qui résoud un problème en un temps n pour au moins une entrée de taille n (pour tout $n \in \mathbb{N}$).

En pratique, le meilleur cas n'est pas très intéressant, car on veut plutôt garantir une certaine rapidité d'exécution à l'utilisateur final (on considère donc les pires exécutions possible).

Question 1 du TP

2. Pourquoi ne définit-on généralement pas la complexité d'un algorithme comme la *moyenne* du nombre d'opérations ou du temps d'exécution ?

Réponse : La complexité moyenne considère une moyenne pondérée de toutes les entrées possibles (qui sont souvent en un nombre infini et parfois même en partie inconnues !). On doit donc choisir une distribution qui représente la probabilité d'occurrence des différentes entrées. Celle-ci peut être difficile à définir précisément (et le résultat peut dépendre fortement de la distribution choisie). De plus, une telle notion a le désavantage de n'avoir aucune des propriétés élémentaires de calcul : dès que vous combinez deux algorithmes, vous devez systématiquement recalculer la complexité de l'algorithme ainsi obtenu à partir de 0.

En pratique, on préfère considérer le pire cas, plus facile à identifier. Il se peut cependant que ce cas là soit rare, et fournit donc une estimation trop pessimiste. Dans ce cas, on tente généralement d'approximer la complexité moyenne à partir d'un cas d'exécution $\ll$ typique $\gg$. Il existe des situations où ce type de complexité moyenne est intéressante en pratique, mais la difficulté (théorique et pratique) de calcul de celle-ci fait qu'on préfère bien souvent la notion classique de complexité.

Question 2 du TP

Justifier rigoureusement les affirmations suivantes à partir de la définition de grand O :

1. $3n^3 + 2n^2 + n \in O(n^3)$
2. $\log_{10}(n^{100}) \in O(n)$
3. $n^4 \in O(3^n)$
4. $2^n \in O(n!)$

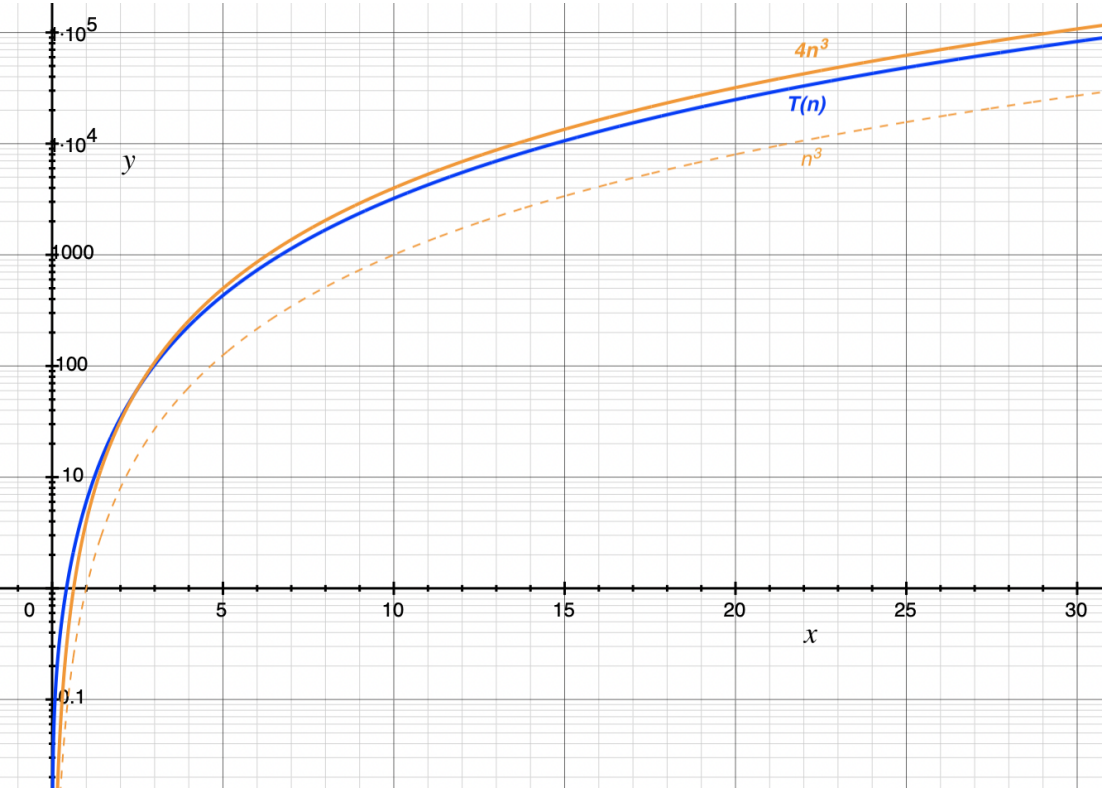
Question 2 du TP

1. $3n^3 + 2n^2 + n \in O(n^3)$

Réponse : $T(n) \in \mathcal{O}(n^3)$ ssi $\exists K \in \mathbb{N}$ et $\exists n_0 \in \mathbb{N}$ tel que :

$$T(n) \leq Kn^3 \quad \forall n \geq n_0$$

C'est le cas pour $K = 4$ et $n_0 = 3$:



$T(n)$ ne croit pas plus vite que $4n^3$ car $\forall n \geq 3$:

$$3n^3 \leq 3n^3$$

$$2n^2 + n \leq n^3$$

$$\Rightarrow 3n^3 + 2n^2 + n \leq 4n^3$$

Question 2 du TP

2. $\log_{10}(n^{100}) \in O(n)$

Réponse : $T(n) \in \mathcal{O}(n)$ ssi $\exists K \in \mathbb{N}$ et $\exists n_0 \in \mathbb{N}$ tel que :

$$T(n) \leq Kn \quad \forall n \geq n_0$$

$T(n)$ ne croit pas plus vite que n car $\forall n \geq 300$:

$$n \leq 10^{\frac{n}{100}}$$

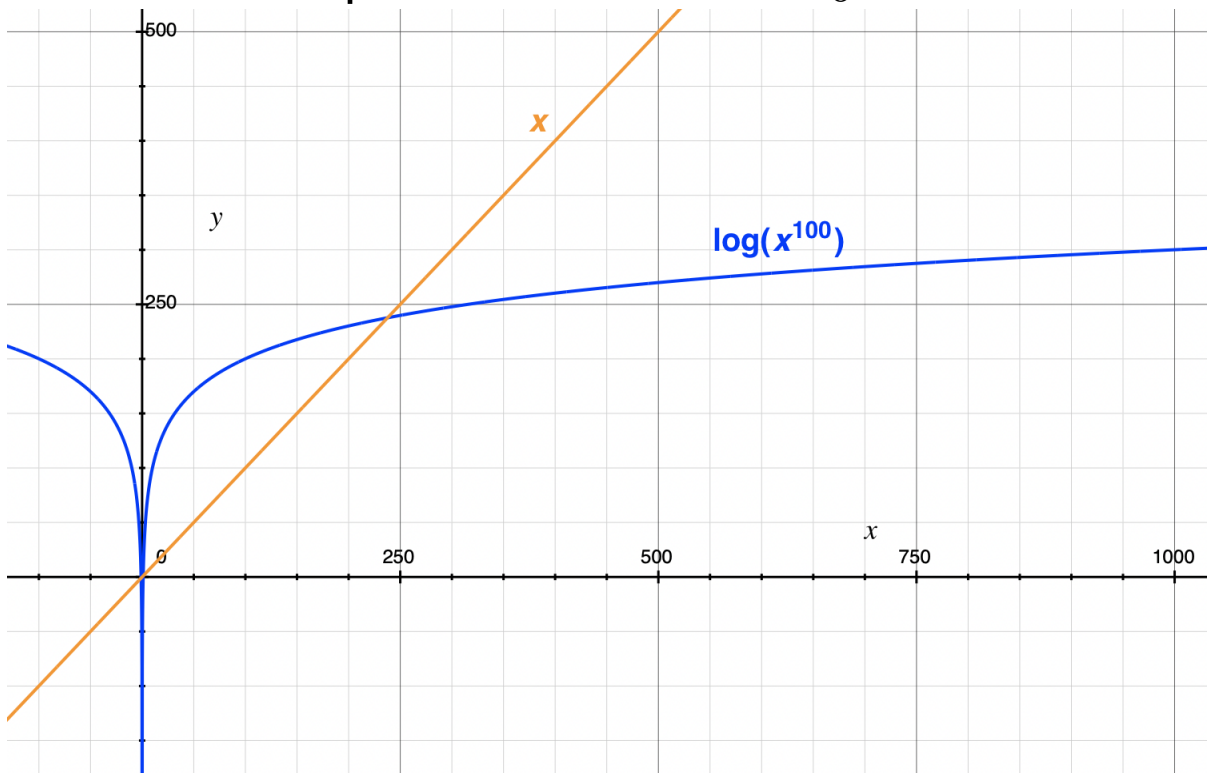
$$(\text{car } 300 \leq 10^3)$$

$$\Rightarrow \log_{10}(n) \leq \frac{n}{100}$$

$$\Rightarrow 100 \cdot \log_{10}(n) \leq n$$

$$\Rightarrow \log_{10}(n^{100}) \leq n$$

C'est le cas pour $K = 1$ et $n_0 = 300$:



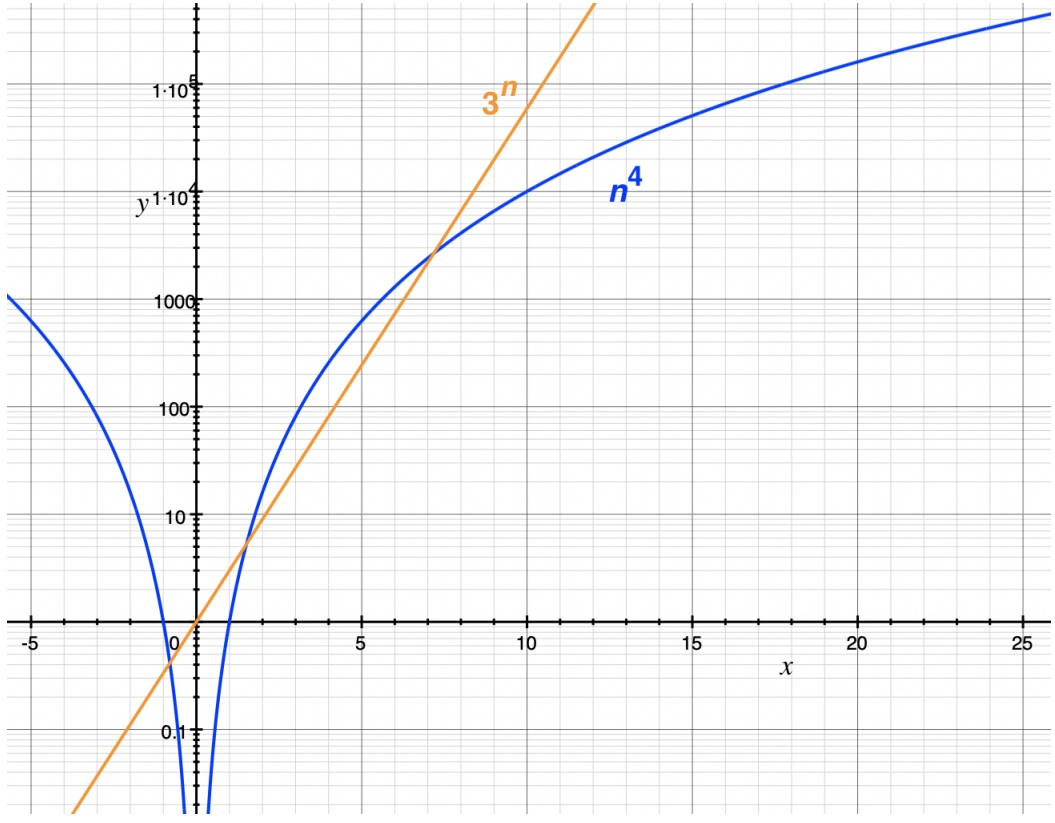
Question 2 du TP

3. $n^4 \in O(3^n)$

Réponse : $T(n) \in \mathcal{O}(3^n)$ ssi $\exists K \in \mathbb{N}$ et $\exists n_0 \in \mathbb{N}$ tel que :

$$T(n) \leq Kn \quad \forall n \geq n_0$$

C'est le cas pour $K = 1$ et $n_0 = 8$:



$T(n)$ ne croit pas plus vite que n car $\forall n \geq 8$:

$$n \leq 3^{\frac{n}{4}}$$

(car $8 \leq 3^2$)

$$\Rightarrow (n)^4 \leq (3^{\frac{n}{4}})^4$$

$$\Rightarrow n^4 \leq 3^n$$

Question 2 du TP

4. $2^n \in O(n!)$

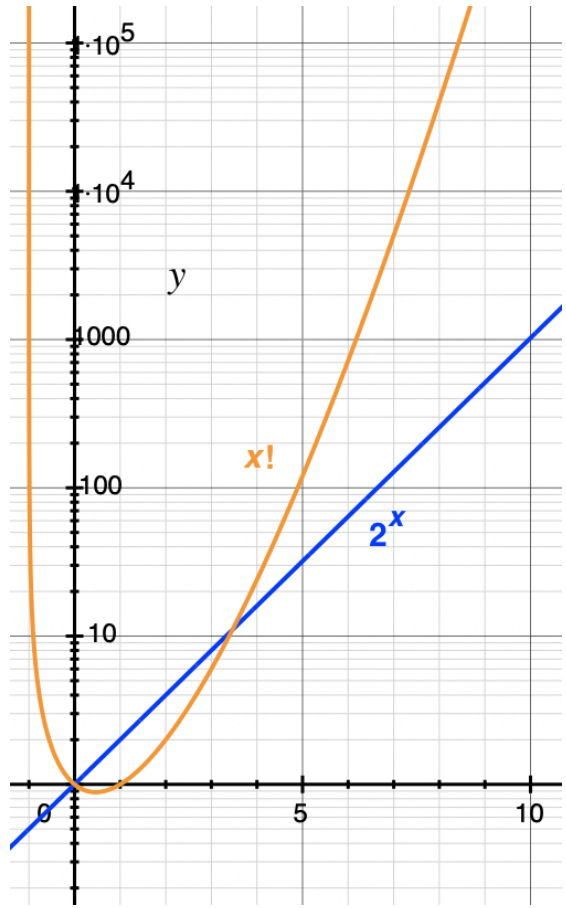
Réponse : On peut montrer par récurrence que $\forall n \geq 4 : n! > 2^n$.

On remarque que $4! = 24 > 16 = 2^4$.

Pour tout $n \geq 4$:

$$\frac{(n+1)!}{n!} = n > 2 = \frac{2^{n+1}}{2^n}$$

ce qui signifie que $n!$ croit plus vite que 2^n.



Question 3 du TP

Supposons avoir deux algorithmes qui ont comme input et output des matrices carrées dont la taille n peut varier (la taille de la matrice en output n'est pas nécessairement la même que celle de la matrice en input). Supposons que la complexité du premier est dans $O(n^2)$ et que celle du second est dans $O(n^3)$.

1. Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme sur cette entrée, puis applique le second algorithme sur cette entrée, puis renvoie le résultat ainsi obtenu ?
2. Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme sur la matrice pour chaque nombre pair plus petit ou égal à n, puis applique le second algorithme sur la matrice pour chaque nombre impair plus petit ou égal à n, puis renvoie la somme des traces des résultats obtenus ?
3. Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme pour obtenir une seconde matrice, puis applique le second algorithme sur cette seconde matrice, puis renvoie le résultat ainsi obtenu ?

Question 3 du TP

1. Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme sur cette entrée, puis applique le second algorithme sur cette entrée, puis renvoie le résultat ainsi obtenu ?

Réponse :
Le temps d'exécution de plusieurs programmes exécutés les uns après les autres équivaut à la somme des temps d'exécution de ces programmes.
On a donc une complexité de la forme :
$T(n) = T_1(n) + T_2(n) = O(n^2) + O(n^3) = O(n^3 + n^2) = O(n^3)$
car on se concentre sur le terme dominant et on ignore le terme $O(n^2)$.

Question 3 du TP

2. Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme sur la matrice pour chaque nombre pair plus petit ou égal à n, puis applique le second algorithme sur la matrice pour chaque nombre impair plus petit ou égal à n, puis renvoie la somme des traces des résultats obtenus ?

Réponse :
On peut exprimer la complexité de cet algorithme via la somme des complexités des algorithmes qu'il exécute :

- Si n pair :

$$T(n) = \sum_{i=0}^{n/2} T_1(n) + \sum_{j=0}^{(n/2)-1} T_2(n)$$

- Si n impair :

$$T(n) = T_2(n) + \sum_{i=0}^{(n-1)/2} T_1(n) + \sum_{j=0}^{((n-1)/2)-1} T_2(n)$$

Question 3 du TP

Réponse (suite) :

Dès lors, puisque que O est une borne supérieure, on peut donc écrire :

- Si n pair :

$$T(n) = \sum_{i=0}^{n/2} O(n^2) + \sum_{j=0}^{(n/2)-1} O(n^3)$$

$$T(n) = \underbrace{O(n^2) + \cdots + O(n^2)}_{n/2 \text{ fois}} + \underbrace{O(n^3) + \cdots + O(n^3)}_{(n/2)-1 \text{ fois}}$$

$$T(n) = O(\frac{1}{2} \cdot n^3) + O(\frac{1}{2} \cdot n^4 - n^3) = O(n^4)$$

- Si n impair :

$$T(n) = O(n^3) + \sum_{i=0}^{(n-1)/2} O(n^2) + \sum_{j=0}^{((n-1)/2)-1} O(n^3)$$

$$T(n) = O(n^3) + \underbrace{O(n^2) + \cdots + O(n^2)}_{(n-1)/2 \text{ fois}} + \underbrace{O(n^3) + \cdots + O(n^3)}_{((n-1)/2)-1 \text{ fois}}$$

$$T(n) = O(n^3) + O(\frac{1}{2} \cdot n^3 - \frac{1}{2} \cdot n^2) + O(\frac{1}{2} \cdot n^4 - \frac{1}{2} \cdot n^3 - n^3) = O(n^4)$$

Question 3 du TP

3. Que peut-on dire sur la complexité de l'algorithme qui prend en entrée une matrice de taille n, applique le premier algorithme pour obtenir une seconde matrice, puis applique le second algorithme sur cette seconde matrice, puis renvoie le résultat ainsi obtenu ?

Réponse :
La première partie de l'algorithme a une complexité dans $O(n^2)$. La deuxième partie de l'algorithme a une complexité dans $O(m^3)$, où m est la taille de la matrice créée par la première partie de l'algorithme. Or, il faut exprimer la classe de complexité de l'algorithme complet en terme de n, pas de m.

- Si on sait que la taille de m est bornée par le nombre d'opération de l'algorithme qui a produit la matrice de taille m, on a $m \leq n^2$ et donc la complexité de l'algorithme complet est $O(n^6)$.
- Si on sait (pour une raison ou pour une autre) que m est borné en fonction de n par une fonction $f(n)$, alors la complexité de l'algorithme complet est $O((f(n))^3)$.
- Si on ne connait aucune borne supérieure pour m en fonction de n, on est bloqué.

Question 4 du TP

Est-il possible que le nombre d'opérations réalisées par un algorithme qui prend en entrée des listes de taille variable soit égal à 100 pour toute liste dont la taille est 42 si...

1. la complexité de l'algorithme est dans $O(2^n)$
2. la complexité de l'algorithme est dans $\Omega(2^n)$
3. la complexité de l'algorithme est dans $\Theta(2^n)$

Question 4 du TP

Est-il possible que le nombre d'opérations réalisées par un algorithme qui prend en entrée des listes de taille variable soit égal à 100 pour toute liste dont la taille est 42 si...

1. la complexité de l'algorithme est dans $O(2^n)$

Réponse : Oui.
La notation O est une **borne supérieure**. En reprenant la définition de O, il existe bien un facteur multiplicatif c (par exemple $c = 1$) tel que $T(42) = 100 \leq c * 2^{42}$.

Exemple : le programme qui effectue 99 opérations inutiles puis qui renvoie son entrée a une complexité constante dans $O(1)$ et qui est donc bien également dans $O(2^n)$.

Question 4 du TP

Est-il possible que le nombre d'opérations réalisées par un algorithme qui prend en entrée des listes de taille variable soit égal à 100 pour toute liste dont la taille est 42 si...

2. la complexité de l'algorithme est dans $\Omega(2^n)$

Réponse : Oui.
La borne inférieure de complexité de cet algorithme pourrait être n'importe laquelle. La complexité d'un algorithme n'est définie que pour des valeurs d'entrée n assez grandes (autrement dit : il s'agit d'une notion asymptotique). En ne connaissant le temps d'exécution que d'une seule entrée, on ne peut savoir comment celui-ci va évoluer lorsque n augmentera.

Exemple : le programme détermine si une formule de la logique des propositions à n variables construite à partir de la liste donnée est satisfaisable par model checking (brute force), mais qui pour toute entrée de taille 42 réalise à la place 99 opérations inutiles, puis renvoie la liste donnée en entrée.

Question 4 du TP

Est-il possible que le nombre d'opérations réalisées par un algorithme qui prend en entrée des listes de taille variable soit égal à 100 pour toute liste dont la taille est 42 si...

3. la complexité de l'algorithme est dans $\Theta(2^n)$

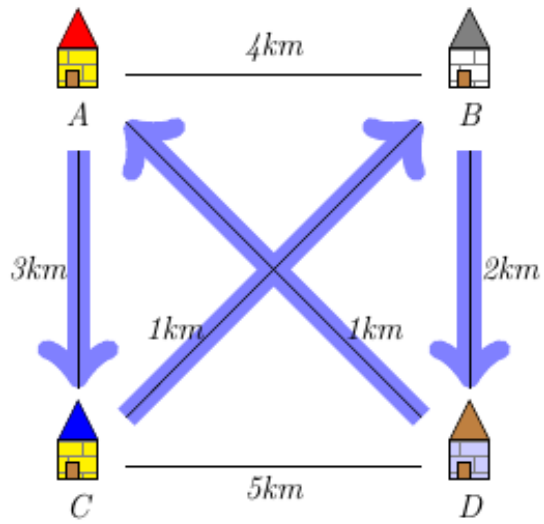
Réponse : Oui.
Même réponse que pour la deuxième sous-question.

Question 5 du TP

Le problème de “voyageur de commerce” (“*travelling salesman*” en anglais ; TSP) est un problème algorithmique classique. Un commerçant veut passer par chacune des n villes données. Il existe une route directe entre chaque paire de villes, chaque route ayant une certaine distance en km.

Une instance du problème est donc représenté par un graphe avec n nœuds (les villes), qui est complet (une arrête entre chaque paire de nœuds) et pondéré (un poid (= distance) pour chaque arrête).

Afin de minimiser ses coûts, notre commerçant souhaite trouver le circuit le plus court lui permettant de revenir à sa ville de départ après être passé une et une seule fois par chacune des autres villes.



Question 5 du TP

- Afin d'obtenir une solution exacte (pour toute entrée), on doit énumérer tous les circuits possibles visitant une et une seule fois chaque nœud du graphe, afin de sélectionner celui totalisant le moins de kilomètres. Quelle est la complexité d'une telle approche ?
- Existe-t-il des cas pour lesquels on pourrait trouver la solution exacte en temps polynomial ? Si oui, donnez un exemple. Si non, justifiez.
- Résoudre le TSP pour des grandes instances peut être intéressant pour l'industrie : e.g. un bras de robot qui resserre une centaine de boulons par voiture dans une ligne de production (minimiser la distance à parcourir peut augmenter la productivité de la ligne). Comment feriez-vous en pratique afin de définir le trajet du robot dans un temps raisonnable ?

Question 5 du TP

- Afin d'obtenir une solution exacte (pour toute entrée), on doit énumérer tous les circuits possibles visitant une et une seule fois chaque nœud du graphe, afin de sélectionner celui totalisant le moins de kilomètres. Quelle est la complexité d'une telle approche ?

  Réponse : $\mathcal{O}(n!)$

  - $n-1$ chemins partant de la 1ère ville ;
  - pour chaque cas, $n-2$ chemins sortant ;
  - etc.

Question 5 du TP

2. Existe-t-il des cas pour lesquels on pourrait trouver la solution exacte en temps polynomial ? Si oui, donnez un exemple. Si non, justifiez.

   Réponse : Un algorithme glouton (*greedy*) consiste à prendre à chaque étape la solution localement optimale, i.e. dans ce cas-ci choisir pour l'étape suivante la ville la plus proche parmi celles restantes.

   Une telle approche peut donner la solution optimale dans le cas où toutes les villes sont situées sur un cercle, par exemple.

   Attention ! Cette approche ne donne en général pas la solution optimale, car des choix précédents ne sont jamais remis en question.

Question 5 du TP

3. Résoudre le TSP pour des grandes instances peut être intéressant pour l'industrie : e.g. un bras de robot qui resserre une centaine de boulons par voiture dans une ligne de production (minimiser la distance à parcourir peut augmenter la productivité de la ligne). Comment feriez-vous en pratique afin de définir le trajet du robot dans un temps raisonnable ?

   Réponse : En pratique, on n'a pas forcément besoin de la solution optimale car il est souvent possible de trouver une bonne approximation en temps polynomial. Par exemple :

   - Utiliser un algorithme *greedy*, dans l'espoir qu'il s'approche de la solution optimale.
   - Utiliser un algorithme de recherche locale, qui part d'un certain circuit (e.g. la solution *greedy*) et fait une série de permutations afin de continuellement réduire la distance.
   - Etc. (*Want more ? ⇒ LINGI2261 - Artificial Intelligence ;-)*)

Challenge

Supposons un instant que grâce à une technologie qui nous est gracieusement offerte par des extra-terrestres, nous sommes soudainement capables de toujours résoudre tout problème pour lequel il existe un algorithme dont la complexité (temporelle) est exponentielle en un temps polynomial. Cela implique-t-il que nous pourrons toujours résoudre tout problème en un temps polynomial ?