TP N°2 : TERMINAISON ET COMPLEXITE TEMPORELLE D'UN ALGORITHME

OBJECTIFS DU TP

- Déterminer la complexité en temps d'un algorithme (expérimentalement et à partir de la structure de l'algorithme)
- Valider l'utilisation de la notation de Landau pour la complexité
- Montrer la terminaison et la correction d'un algorithme.

PARTIE 1 : MULTIPLICATION EGYPTIENNE

Considérons le programme suivant, qui implémente un ancien algorithme égyptien.

```
def egyptienne(a:int, b:int) ->int :
    t=0
    while a>0 :
        if a%2 == 1 :
            t = t + b
        b = 2*b
        a = a //2
    return t
```

Q1. Détailler l'exécution de `egyptienne(41,3)`

Q2. Montrer que la fonction `egyptienne` termine toujours.

Q3. Montrer à l'aide d'un invariant de boucle que la fonction renvoie le produit entre les deux arguments.

PARTIE 2 : MESURE DE LA COMPLEXITE EN TEMPS D'UN ALGORITHME

La complexité en temps $C(n)$ d'un algorithme permet de déterminer la durée d'exécution du programme constituant une implémentation de cet algorithme en fonction de la taille d'instance n associée. On peut, de manière parfaitement équivalente, considérer la complexité en termes de nombre d'opérations élémentaires d'un algorithme, ces deux grandeurs étant équivalentes en notation de Landau.

Q1. Justifier qu'il soit équivalent, en notation de Landau, de considérer les complexités en temps et en nombre d'opérations élémentaires.

Afin d'étudier la complexité en temps C d'un algorithme, on mesure la durée d'exécution Δt d'une fonction donnée en faisant varier la taille d'instance n.

Q2. Expliquer pourquoi, pour un algorithme de complexité en temps C polynomiale, il peut être particulièrement intéressant de représenter la fonction $\log(\Delta t) = f(\log(n))$, avec Δt le temps

d'exécution de l'algorithme et n la taille d'instance. On supposera que n est « suffisamment grand » pour procéder à quelques simplifications que l'on explicitera.

Afin de s'appuyer sur des exemples, on travaille avec les fonctions `foo` et `bar` suivantes :

```
def foo(n: int) -> int:
    s = 0
    for k in range(n):
        s = s + k**2
    return s
```

```
def bar(n:int)-> int :
    a=0
    for i in range (n):
        for j in range (n):
            a=a+i*j
    return(a)
```

Q3. Que fait la fonction `foo` ? Déterminer sa complexité en temps C_{foo} en notation de Landau.

Q4. Que fait la fonction `bar` ? Déterminer sa complexité en temps C_{bar} en notation de Landau.

Q5. Écrire une fonction `chrono` prenant comme arguments une fonction `g` et un entier n (représentant la taille d'instance du problème étudié), et retournant la durée d'exécution de la fonction g pour la taille d'instance donnée. On utilisera la méthode time de la bibliothèque `time`.

Q6. Saisir les commandes suivantes dans le shell :

```
>>> chrono(foo, 100000)
>>> chrono(foo, 1000000)
>>> chrono(foo, 10000000)
```

Ces résultats sont-ils en accord avec la complexité en temps de la fonction `foo` déterminée précédemment ?

Q7. Saisir les commandes suivantes dans le shell :

```
>>> chrono(bar, 100)
>>> chrono(bar, 1000)
>>> chrono(bar, 10000)
```

Ces résultats sont-ils en accord avec la complexité en temps de la fonction bar déterminée précédemment ?

Q8. Écrire une fonction `complexite` prenant comme arguments une fonction `g` et une liste d'entiers L (représentant plusieurs tailles d'instance n_i), et retournant les deux listes suivantes :

- une liste constituée des valeurs $log(n_i)$;
- une liste constituée des valeurs $log\,(\Delta t(g(n_i)))$ avec Δ t(g(n_i)) le temps d'exécution de la fonction g pour laquelle la taille d'instance choisie est n_i.

On utilisera la fonction `chrono`.

Q9. Proposer une suite d'instructions permettant de tracer la courbe $log\,(\Delta t(g(n_i))) = f(log(n_i))$. On utilisera la bibliothèque `matplotlib.pyplot`.

Q10. Appliquer ce protocole aux cas suivants :

a. étude de la complexité de la fonction `foo` – on prendra les valeurs suivantes pour n : $n \in \{5 \cdot 10^5, 6 \cdot 10^5, 7 \cdot 10^5, 8 \cdot 10^5, 1 \cdot 10^6, 6 \cdot 10^6\}$

b. étude de la complexité de la fonction `bar` – on prendra les valeurs suivantes pour n : $n \in \{1000, 2000, 3000, 4000, 5000, 6000\}$

Qu'observe-t-on ?

Q11. Déterminer la pente des droites obtenues pour les fonctions `foo` et `bar`. On pourra utiliser la méthode `polyfit` de la bibliothèque `numpy` ou la méthode `curve_fit` de la bibliothèque `scipy.optimize`. Conclusion ?

PARTIE 3 : APPLICATION AU PROBLEME DU VOYAGEUR DE COMMERCE

Le problème du voyageur de commerce peut s'énoncer de la manière suivante :

Un voyageur de commerce doit – au cours de sa tournée – visiter les n villes de la zone géographique qu'il souhaite couvrir. Connaissant les coordonnées $(x_i, y_i)_{i \in [\![1,n]\!]}$ des n villes visitées par le voyageur de commerce, quel est le trajet le plus court permettant de visiter une et une seule fois chaque ville ?

Ce problème « anodin en apparence » est en réalité un problème majeur en Informatique, et permet d'illustrer l'une des questions centrales des Mathématiques : « tout problème possédant une solution peut-il être résolu par un algorithme de complexité en temps polynomiale ? ».
Cette question est, à ce jour, sans réponse – elle constitue l'un des « problèmes du prix du millénaire » (chacun dotés d'un prix d'un million de dollars US). La résolution de ce problème aurait des répercussions majeures, notamment en cryptographie.
Afin d'étudier ce problème, on suppose que chaque ville i se situe au point de coordonnées (x_i, y_i) avec $x_i \in [0,1000]$ u.a. et $y_i \in [0,1000]$ u.a. Chaque ville est donc positionnée au hasard dans un carré de côté 1000 u.a. Dans la suite de l'exercice, chaque ville est représentée par un tuple comprenant :

- un entier représentant le numéro i de la ville ;
- un flottant représentant l'abscisse x_i de la ville ;
- un flottant représentant l'ordonnée y_i de la ville.

Q12. Écrire une fonction `posville` ne prenant pas d'argument, et retournant un tuple de deux flottants représentant les coordonnées (x_i, y_i) de la ville i. Les coordonnées x_i et y_i sont chacune tirées aléatoirement de manière uniforme sur l'intervalle [0, 1000]. On utilisera la bibliothèque `random`.

Q13. Écrire une fonction `carte` prenant comme argument un entier n, et retournant une liste de n villes numérotées de 1 à n selon le format décrit précédemment (chaque ville est représentée par un tuple comportant 3 éléments). On utilisera la fonction `posville.`

Q14. Quelle est la complexité en temps de la fonction carte ? Vérifier ce résultat à l'aide des outils développés dans la première partie. Commenter le résultat obtenu et proposer une interprétation.

Q15. Écrire une fonction `trajets` prenant comme argument une liste C (représentant la liste des villes visitées par le voyageur de commerce), et retournant tous les trajets possibles passant une seule fois par chaque ville sous la forme d'une liste de tuples (chacun de ces tuples représentant une ville visitée). On pourra utiliser la méthode `permutations` de la bibliothèque `itertools` afin de générer toutes les permutations des éléments d'une liste :

```
>>> import itertools
>>> L = [4, 0, 1]
>>> P = itertools.permutations(L)
>>> for p in P:
...     print(p)
(4, 0, 1)
(4, 1, 0)
(0, 4, 1)
(0, 1, 4)
(1, 4, 0)
(1, 0, 4)
```

Q16. Écrire une fonction `distance` prenant comme arguments deux tuples (représentant les villes i et j), et retournant la distance (à vol d'oiseau) entre les villes i et j.

Q17. Écrire une fonction `voycom` prenant comme argument une liste C (représentant la liste des villes visitées par le voyageur de commerce), et retournant une liste représentant le trajet effectué par le voyageur de commerce entre ces villes permettant de minimiser la distance totale parcourue. On utilisera les fonctions `trajets` et `distance`.

Q18. Ecrire une fonction `voycomalea` prenant comme argument un entier n, et retournant une liste représentant le trajet effectué par le voyageur de commerce entre n villes positionnées aléatoirement dans un carré de côté 1000 u.a. et permettant de minimiser la distance totale parcourue. On utilisera les fonctions `carte` et `voycom`.

Q19. Quelle est la complexité en temps de la fonction `voycomalea` ? Vérifier ce résultat à l'aide des outils développés dans la première partie. On rappelle la formule de Stirling :

$$\text{Lorsque n} \to \infty, \quad \ln(n!) \sim n \times \ln(n) - n$$

Attention, ne testez pas avec plus de 9 villes, le temps de calcul devient vite conséquent.