Algorithmique et structures de données
En plus du TP #2

nicolas.audebert@onera.fr

Mercredi 6 février 2019

Attention : faire ce TP ne dispense pas de rendre le TP #2 du cours. Il s'agit d'un exercice additionnel **facultatif** qui vous est proposé, pour celles et ceux qui maîtrisent les différents algorithmes de tri usuels.

Les différentes sous-parties ci-dessous sont indépendantes.

1 Tris exotiques

Le TP #2 présente plusieurs algorithmes de tris classiques, comme le tri à bulles, le tri par insertion, le tri sélection et le tri rapide. Nous nous proposons ici d'étudier des algorithmes de tri un peu plus exotiques. Vous êtes libre de les implémenter en remplacement d'un des tris quadratiques du TP #2.

1.1 Le tri stupide

Le tri stupide (ou *bogosort*) est un algorithme de tri probabiliste. Il consiste à mélanger aléatoirement tous les éléments du tableau jusqu'à ce que celui-ci soit trié.

Question 1 Quelle est la complexité du tri stupide dans le meilleur cas ?

Question 2 Quel est le temps d'exécution du tri stupide dans le pire cas ?

Question 3 Quelle est la complexité moyenne du tri stupide ?

1.2 Le tri cocktail

Le tri cocktail (ou tri shaker) est une variante bidirectionnelle du tri à bulles. Dans le tri à bulles classique, on peut distinguer deux types d'éléments : les *lièvres* et les *tortues*. Les lièvres sont les éléments les plus grands, qui remontent rapidement vers la droite du tableau. À l'inverse, les tortues désignent les

éléments les plus petits du tableau, qui descendent lentement vers la gauche du tableau.

Le tri cocktail consiste à trier alternativement le tableau dans un sens, puis dans l'autre. Ainsi, on réalise tout d'abord une première passe permutant les éléments i et $i+1$ si $t[i] > t[i+1]$ pour i allant de 0 à n. Puis on réalise une seconde passe permutant les éléments $i-1$ et i si $t[i] < t[i-1]$ pour i allant de n à 0. On répète l'opération jusqu'à ce que le tableau soit complètement trié.

Une implémentation possible en Python est la suivante :

```
def tri_cocktail(lst):
    echange = True
    while(echange):
        echange = False
        for i in range(0, len(lst) - 2):
            if lst[i] > lst[i+1]:
                lst[i], lst[i+1] = lst[i+1], lst[i]
                echange = True
        for i in range(len(lst) - 2, 0, -1):
            if lst[i] > lst[i + 1]:
                lst[i], lst[i+1] = lst[i+1], lst[i]
                echange = True
    return lst
```

Question 1 Quelle est la complexité dans le pire cas du tri cocktail ?

Question 2 Peut-on optimiser le tri cocktail tel que présenté ici pour diminuer sa complexité, sans en changer le principe de fonctionnement ?

1.3 Le tri à peigne

Le tri à peigne est une variante du tri à bulles. Il consiste à comparer chaque élément avec un autre élément du tableau espacé d'un certain intervalle. Les deux éléments sont échangés s'ils ne sont pas dans le bon ordre, et l'opération est répétée en diminuant l'intervalle de comparaison. Lorsque l'intervalle arrive à 1, il s'agit d'un tri à bulles classique. Généralement, le facteur de diminution de l'intervalle est fixé à $1,3$.

Une implémentation possible en Python est la suivante :

```
def tri_peigne(lst):
    intervalle = len(lst)
    echange = False

    while(intervalle > 1 or echange):
        intervalle = int(intervalle / 1.3)
        if intervalle < 1:
            intervalle = 1
```

```
        echange = False
        for i in range(0, len(lst) - intervalle):
            if lst[i] > lst[i + intervalle]:
                lst[i], lst[i+intervalle] = lst[i+intervalle], lst[i]
                echange = True
    return lst
```

Question 1 En quoi le tri à peigne est-il plus intéressant que le tri à bulles ?

Question 2 Quelle est la complexité en pire cas du tri à peigne ? Dans le meilleur cas ?

2 Échange rapide de deux variables

Échanger les valeurs de deux variables a et b est généralement implémenté à l'aide d'une variable temporaire, de la façon suivante :

```
int a = +1, b = -1;
int tmp;

tmp = a;
a = b;
b = tmp;
```

Toutefois, il faut bien constater que cela pourrait être plus efficace. En effet, il nous faut allouer une variable temporaire et exécuter une opération d'affectation supplémentaire ! Pouvez-vous trouver une façon d'échanger a et b qui n'utilise pas de variable temporaire ?

3 Anagrammes

Une anagramme est une permutation des lettres d'un ou plusieurs mots permettant d'écrire un autre mot. Par exemple :

- "gare maman" est une anagramme du mot "anagramme",
- "Pascal Obispo" est une anagramme de "Pablo Picasso",
- "chicane" est une anagramme de "caniche".

Question 1 Proposer un algorithme naïf permettant de vérifier que deux chaînes de caractères sont des anagrammes. Quelle est sa complexité ?

Question 2 Il est facile de démontrer qu'un algorithme permettant de vérifier que deux chaînes de caractères sont des anagrammes a une complexité linéaire au minimum. Pouvez-vous exhiber un algorithme de cette complexité ?

Solutions

1 Tris exotiques

1.1 Le tri stupide

Tout d'abord, il est nécessaire de comprendre quelles sont les deux étapes du tri stupide. La première étape consiste à permuter aléatoirement et de façon uniforme les éléments du tableau. Cela peut se réaliser en $O(n)$ avec l'implémentation suivante :

```
from random import randint

def permute(lst):
    # Permutation en place
    for i in range(0, len(lst)):
        other = randint(i, len(lst) - 1)
        lst[i], lst[other] = lst[other], lst[i]
```

En outre, il faut vérifier après chaque permutation si le tableau est trié ou non, ce qui se réalise en $O(n)$. Autrement dit, pour chaque itération du tri stupide, on réalise $O(2n) = O(n)$ opérations.

Question 1 Dans le meilleur des cas, le tri stupide va permuter une fois le tableau et celui-ci sera immédiatement trié. La complexité est donc en $O(n)$.

Question 2 Dans le pire des cas, à chaque permutation aléatoire le hasard fait que le tableau n'est jamais trié. On s'aperçoit alors que le temps d'exécution de l'algorithme n'est pas borné, car si improbable que ce soit, il est possible que l'on ne tombe jamais sur la bonne permutation !

Question 3 Dans un premier temps, il faut estimer l'espérance probabiliste du nombre de tirages à réaliser pour tomber sur la bonne permutation. Pour un tableau de longueur n, il y a $n!$ permutations possibles. Parmi toutes celles-ci (en supposant que tous les éléments sont distincts), il n'y en a qu'une seule qui donne un tableau trié. Comme le tirage des permutations est uniforme, on a à chaque tirage une probabilité $1/(n!)$ de tomber sur le tableau trié. Il faut donc en moyenne $n!$ tirages pour trier le tableau. Or, chaque tirage nécessite $O(n)$ opérations (pour la permutation et la vérification), soit une complexité moyenne de $O(n.n!)$. Oups...

1.2 Le tri cocktail

Question 1 La situation dans le pire cas est analogue au tri à bulles et la complexité est en $O(n^2)$.

Question 2 On peut diviser par 2 le nombre de comparaisons à réaliser. En effet, à chaque passe du tri, on peut être certain qu'un nouvel élément a été positionné à son emplacement final et il n'est donc pas nécessaire de le vérifier à nouveau. On peut donc réitérer le tri cocktail sur la sous-liste centrale, en excluant les éléments aux extrémités (qui sont déjà triés).

1.3 Le tri à peigne

Question 1 Comme pour le tri cocktail, le tri à peigne permet de se débarrasser plus rapidement des tortues. En effet, si des éléments de petite valeur sont initialement à droite du tableau, la comparaison avec intervalle va permettre de les ramener immédiatement sur la gauche du tableau, les rapprochant ainsi de leur position finale.

Question 2 Ces cas sont analogues en tout point au tri à bulles.

2 Échange rapide de deux variables

Deux propositions :

```
int a, b;
a = a + b;
b = a - b; // b = a_0 + b_0 - b_0 = a_0
a = a - b; // a = a_0 + b_0 - a_0 = b_0
```

```
int a, b;
// Maîtrisez-vous le XOR ?
// Avantage de cette méthode : elle fonctionne
// directement sur la représentation binaire des
// variables et s'applique donc à n'importe quel type !
a = a ^ b;
b = a ^ b; // b = a_0 ^ b_0 ^ b_0 = a_0
a = a ^ b; // a = a_0 ^ b_0 ^ a_0 = b_0
```

Attention tout de même, cela n'a que très peu d'intérêt dans la pratique. Les processeurs disposent presque tous d'une instruction `SWAP` permettant de réaliser un échange en une seule instruction et les compilateurs modernes optimisent le code pour en tirer profit.

3 Anagrammes

Question 1 Notons $s1$ et $s2$ nos deux chaînes de caractères. Si l'on trie $s1$ et $s2$ avec l'ordre lexicographique, il suffit de vérifier que les deux chaînes triées sont identiques. Le tri peut s'effectuer en $O(n\ log(n))$ et la comparaison en $O(n)$, donc un algorithme final en $O(n\ log(n))$.

Note : bien sûr, on peut faire pire. Par exemple, pour chaque élément de $s1$, chercher s'il existe dans $s2$ et le cas échéant, le retirer de $s2$ et $s1$, puis répéter l'opération. Cet algorithme a une complexité en $O(n^2)$.

Question 2 L'affirmation de l'énoncé est facile à démontrer : pour vérifier que deux chaînes sont anagrammes, il est nécessaire d'effectuer une comparaison pour au moins tous les éléments de la plus longue des chaînes.

Maintenant, comment se contenter de $O(n)$ opérations ? Un premier algorithme consiste à utiliser des histogrammes d'apparition des différents caractères. On parcourt tout d'abord $s1$ et pour chaque occurrence d'une lettre, on incrémente sa valeur dans l'histogramme de 1. Puis on fait de même avec $s2$. Ces deux étapes se font en $O(n)$. Il suffit alors de vérifier que les histogrammes sont égaux pour s'assurer que $s1$ et $s2$ contiennent les mêmes lettre en quantités égales ! Soit un algorithme final en $O(n)$. On peut même ne calculer qu'un seul histogramme en l'incrémentant pour $s1$, en le décrémentant pour $s2$ et en vérifiant que tous les éléments de l'histogramme sont à 0.

Exemple d'implémentation en Python :

```
def anagramme(s1, s2):
    hist = {}
    for c in s1:
        if not c in hist:
            hist[c] = 1
        else:
            hist[c] += 1
    for c in s2:
        if not c in hist:
            return False
        else:
            hist[c] -= 1

    for val in hist:
        if val != 0:
            return False
    return True
```

Une autre proposition consiste à associer à chaque caractère un nombre premier unique. Par exemple, aux 26 lettres de l'alphabet romain de 'a' à 'z', on associe les 26 premiers nombres premiers de 2 à 101. On peut alors transformer une chaîne de caractères en le produit des nombres premiers associés à ses éléments. Par exemple, "caniche" devient $5\times2\times43\times23\times5\times19\times11 = 10335050$. Par l'unicité de la décomposition en facteurs premiers, si deux chaînes ont le même produit, alors elles contiennent les mêmes caractères !

Exemple d'implémentation en Python :

```
alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
            'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
```

```
            'y', 'z']
premiers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
            59, 61, 67, 71, 73, 79, 83, 89, 97, 101]
correspondance = {c: p for c, p in zip(alphabet, premiers)}

def chaine_vers_produit(str):
    produit = 1
    for c in str:
        produit = produit * correspondance[c]
    return produit

def anagramme(s1, s2):
    return chaine_vers_produit(s1) == chaine_vers_produit(s2)
```