Devoir d'informatique 3 – Corrigé

D.Malka – MPSI 2017-2018 – Lycée Saint-Exupéry

11.01.2018

Exercice 1 – Faut-il trier pour mieux rechercher ?

Dans ce problème, on se demande s'il est intéressant de trier une liste avant d'y rechercher un élément ou s'il est préférable d'opérer la recherche directement dans la liste.

1. Fonction `rech_simple` :

```
1 def rech_simple(L,e):
2     n=len(L)
3     for i in range(n):
4         if L[i]==e:
5             return e
6     return None
```

La complexité au pire de la fonction en nombre de comparaisons est $O(n)$ où n est la longueur de la liste.

2. Si la liste est triée, on sait qu'une recherche dichotomique est plus efficace. On en donne l'algorithme en langage Python :

```
1  def rech_dicho(e,L):
2      g=0
3      d=len(L)-1
4      while g<=d:
5          m=(d+g)//2
6          if e>L[m]:
7              g=m+1
8          elif e<L[m]:
9              d=m-1
10         else :#x==T[m]:
11             return m
12     return None
```

2.1 Valeur de i à chaque itération :

itération i	m	intervalle de recherche
0	non définie	[10, 23, 30, 33, 38, 39, 40, 43, 52, 66, 90, 95, 97, 115, 117]
1	7	[10, 23, 30, 33, 38, 39, 40]
2	3	[10, 23, 30]
3	2	[30]
4	1	non défini, les indices g et d se croisent

L'appel `rech_dicho(L,27)` renvoie donc `None` au bout de 4 itérations.

2.2 Montrons que la complexité au pire de la fonction de recherche dichotomique est $O(\log_2(n))$ où n est la longueur de la liste passée en argument.

On note $l_i = d_i - g_i + 1$ la longueur du tableau de recherche à l'issu de l'itération i. Dans un premier temps, montrons que $\forall i,\ l_i \leq \dfrac{n}{2^i}$.

— Pour $i = 0$, $d_0 = n - 1$ et $g_0 = 0$ donc $l_0 = n = \dfrac{n}{2^0}$: proposition vraie.

— Supposons alors $l_i \leq \dfrac{n}{2^i}$. A la $i+1^{\text{ème}}$ itération : $l_{i+1} = l_i//2$ où la division s'entend comme euclidienne donc $l_{i+1} \leq \dfrac{l_i}{2}$. En utilisant l'hypothèse de récurrence il vient $l_{i+1} \leq \dfrac{n}{2^{i+1}}$.

Donc par récurrence : $\forall i,\ l_i \leq \dfrac{n}{2^i}$

Notons k le nombre d'itérations de la boucle `while`. Au pire, à la sortie de boucle, $d_k = g_k$ et donc $l_k = 1$. Or $l_k \leq \dfrac{n}{2^k}$ donc, au pire :

$$1 \leq \frac{n}{2^k} \Leftrightarrow \boxed{k \leq \log_2(n)}$$

La complexité au pire de l'algorithme de recherche dichotomique est $O(\log_2(n))$.

3. Pour trier la liste, on propose la fonction `tri` suivante :

```
1 def tri(L):
2     n = len(L)
3     for i in range(n):
4         for j in range(i,0,-1):#parcourt la liste a l'envers
5             if L[j]<L[j-1]:
6                 L[j],L[j-1] = L[j-1],L[j]
```

3.1 Appliquer la fonction `tri` à la liste `[11, 48, 25, 6, 24]`. On représentera l'état de la liste pour chaque itération sur `i`.

i	L
0	[11, 48, 25, 6, 24]
1	[11, 48, 25, 6, 24]
2	[11, 25, 48, 6, 24]
3	[6, 11, 25, 48, 24]
4	[6, 11, 24, 25, 48]

3.2 Montrons que $\mathcal{P}_i$: « la liste `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération i » est un invariant de boucle.

- Initialisation : $i = 0$. La liste `L[0:1]` est égale au seul élément `L[0]`. Une liste de longueur 1 est nécessairement triée. $\mathcal{P}_0$ vraie.
- Hérédité. Supposons `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération `i` et montrons que `L[0:i+2]` est est triée par ordre croissant à l'issue de l'itération `i+1`.
  - A l'itération `i+1`, on permute tout couple d'éléments `L[j],L[j-1]` si `L[j]<L[j-1]` pour j allant de 0 à $i+1$.
  - La liste `L[0:i+1]` étant triée, seules des permutations avec l'élément `e` d'indice initialement `i+1` peuvent avoir lieu.
  - Considérons donc la série de permutations potentielles de l'élément `e` d'indice initialement `i+1` avec les éléments d'indice inférieur.
  - Pour cet élément, le test `L[j]<L[j-1]` s'avère vrai jusqu'au premier indice μ tel que `L[`μ`]>=e`. On note `k`, le nouvel indice de `e`. La liste `L[0:i+1]` étant triée, on est assuré que pour $\forall j \leq k,\ e \geq L[j]$.
  - L'élément `e` (indice k) est donc positionné correctement dans la liste i.e. $\forall j \leq k,\ e \geq L[j]$ et $\forall j \geq k,\ e \leq L[j]$ (1).
  - La liste `L[0:i+1]` étant initialement triée, la liste `L[0:k+1]` est triée (2).
  - La liste `L[k+1:i+2]` étant initialement triée, la liste `L[0:k+1]` est triée (3).
  - On déduit de (1), (2) et (3) que la liste `L[0:i+2]`=`L[0:k+1]`+`L[k+1:i+2]` est triée par ordre croissant.

Donc la proposition $\mathcal{P}_i$: « la liste `L[0:i+1]` est triée par ordre croissant à l'issue de l'itération i » est un invariant de boucle.

A la sortie de la boucle for la plus externe $i = n-1$ ainsi la liste `L[0:n]` soit la liste `L` dans sa totalité est triée.

3.3 Au pire, la fonction `tri` réalise $O(n^2)$ comparaisons.

4. On suppose qu'en moyenne, on est amené à réaliser quotidiennement R recherches dans des listes et E enregistrement de listes. En supposant toutes les listes de même taille n, pour quelles valeurs du rapport $\dfrac{R}{E}$ est-il intéressant de trier les listes avant d'y rechercher un élément plutôt que d'utiliser simplement la fonction `rech_simple` ?

Comparons les coûts :

— Le coût c_1 de R recherches séquentielles sur des listes non triées : $R \times n$
— Le coût $c_2$de E tris et de R recherche dichotomique : $E \times n^2 + R \times \log_2(n)$.
Il est intéressant de trier les listes si :

$$c_1 < c_2$$

$$Rn > En^2 + R\log_2(n)$$

$$1 > \frac{E}{R}n + \frac{\log_2(n)}{n}$$

Comme de façon asymptotique $\frac{\log_2(n)}{n} \ll 1$,

$$\boxed{\frac{R}{E} > n}$$

Il faut réaliser au moins n fois plus de recherche que d'enregistrement pour qu'il soit intéressant de trier les listes.

5. Avec un algorithme de tri de complexité $O(n\log_2(n))$ (par exemple le tri-fusion), alors il est intéressant de trier les listes si :

$$\boxed{\frac{R}{E} > \log_2(n)}$$

Cette condition a plus de chance de se produire dans un cas concret.

Exercice 2 – Séquençage génomique

C

1. Distance de Hamming.

1.1 La Distance de Hamming entre les séquences :

```
ATATACAGATCGGCTGTTGCGTGCGTAGTTGCGT
ATATACGGATCGGCTGTTGCCTGCGTAGTAGCGT
```

vaut 3.

1.2 Programme calculant la distance de Hamming entre deux séquences nucléiques :

```
1 def Hamming(s1,s2):
2     n=len(s1)
3     d=0
4     for i in range(n):
5         if s1[i]!=s2[i]:
6             d=d+1
7     return d
```

FIGURE 1 – Calcul de la distance de Hamming entre deux séquences nucléiques

2. Traduction d'un gêne.

2.1 Fonction `Start` qui recherche la position de la première occurrence du codon AUG.

2.2 La fonction traduit un gêne présent sur un brin d'ARNm en la protéine qu'il code. Pour cela il prend en argument la séquence nucléique en argument et recherche le début et la fin de la séquence codante via les les fonction `start` et `stop`. Dans la boucle, le programme extrait des sous-chaîne de trois caractères : ce sont les codons. Il recherche l'acide aminé correspondant grâce à la fonction `rech` et l'ajoute à la liste `prot`. A la fin du programme, la liste `prot` contient la succession des acides aminés constituant la protéine codée par le gêne.

Complexité : $O(n)$ puisque, finalement, le programme consiste à parcourir la chaîne de longueur n par morceau de longueur 3.

```python
def start(seq):
    d=0
    sous_seq=seq[d:d+3]
    while sous_seq!="AUG" and d<len(seq)-3:
        d=d+1
        sous_seq=seq[d:d+3]
    if d<len(seq)-3:
        return d
    else:#AUG n'a pas ete trouve dans la sequence
        return None
```

FIGURE 2 – Fonction `Start`

```python
def ARNm_to_prot(seq,code):
    d=start(seq)
    f=stop(seq)
    prot=[]
    a=""
    i=d
    j=i+3
    while a !="Stop" and j<=f:
        codon=seq[i:j]
        a=rech(codon)
        prot.append(a)
        i=j
        j=i+3
    return prot

def start(seq):
    d=0
    sous_seq=seq[d:d+3]
```

FIGURE 3 – Algorithme 2

Exercice 3 – Autour des tables et des fonctions de hachage

1. Un (mauvais) exemple de fonction de hachage :

$$h : c \to c\%255$$

où c est un entier naturel (c'est-à-dire une chaîne de chiffres !)

1.1 Empreinte de 1753 : $h(1753) = 1753\%255 = 223$

1.2 Très simple, l'ensemble des entiers n tel que $n = m + k \times 255$ où $k \in \mathcal{N}$ a même empreinte que m. Exemple pour $m = 1753$: 2008, 2263,...

La fonction proposée n'est donc pas résistante aux collisions.

2. Mots de passe

2.1 Evaluons la possibilité technique de casser un mot de passe par la force brute suivant la manière dont il a été choisi.

- Mot du dictionnaire : la langue française compte environ 100 000 mots (même ordre de grandeur dans les autres langues). Il suffit d'essayer chaque mot de manière séquentielle. Le coût de chaque essai en comparaison est de l'ordre de la longueur du mot soit en moyenne 10. En quelque secondes, le mot de passe est trouvé.
- Suite aléatoire de caractères : il existe 95^n suite aléatoires de n caractères parmi les 95 caractères imprimables de la table ASCII : la complexité est exponentielle. Pour $n = 8$ (le minimum souvent recommandé), il existe déjà environ 6.10^{15} possibilités à tester. Le code ne peut pas être cassé en un temps raisonnable.

2.2 Fonction de hachage.

2.2.1 Fonction `authentification`.

```
1 def authentification(e,p):
2     if h(p)==e:
3         return True
4     else:
5         return False
```

FIGURE 4 – Fonction `authentification`

2.2.2 Si la fonction h n'est pas résistante aux collisions, on peut générer la même empreinte e qu'un mot de passe p à partir d'une séquence de caractères p' différente de p et ainsi falsifier une authentification.

3. Table de hachage

3.1 Exemple : pour $m = 10$, représenter le dictionnaire dont la clé est un entier et dont la valeur indique par `True` ou `False` si ce nombre est premier.

clé	0	1	2	3	4	5	6	7	8	9
valeur	False	False	True	True	False	True	False	True	False	True

3.2 L'implémentation naïve d'un dictionnaire conduit à une structure de complexité en espace de l'ordre de $O(m)$ pour stocker seulement $n \ll m$ valeurs. Ce n'est pas très efficace.

3.3 Pour que la lecture et l'écriture reste de complexité constante, il faut que le hachage de la clé soit de complexité $O(1)$.