Informatique – TD7 : Complexité et structures de données

CPP

Septembre - novembre 2015

1 Complexité

1.1 Introduction (5mn)

Un algorithme est dit de complexité polynomiale si sa complexité est une fonction polynomiale de la taille du problème n, par exemple $f(n) = 3n^3 - n^2 + 10$. Dans le cas d'une complexité polynomiale, les coefficients des monômes n'importent pas puisqu'on s'intéresse à la complexité asymptotique (quand la taille des données devient grande), et seul le degré du polynôme compte (n^3 dans l'exemple ci-dessus).

Donc, en général, quand on « compte » les opération pour évaluer la complexité, on compte 1 pour chaque opération (affectation, comparaison, arithmétique sur des entiers ou des flottants), même si la réalité (le temps mis par chaque opération à s'exécuter) est différente.

Attention quand même attention aux « opérations » qui coûtent $O(n)$ (comme les comparaisons ou copies de liste). L'amalgame « 1 ligne de code = $O(1)$ » est délicat avec un langage de haut niveau comme Python.

Cette observation s'applique à tous les calculs de complexité asymptotique, et même au cas des complexités non-polynomiales : on s'intéresse uniquement au terme dominant.

1.2 Multiplication de deux matrices (10mn)

Soient $A = (a_{ij})$ une matrice $n \times m$ et $B = (b_{ij})$ une matrice $m \times p$.
Le produit de A et B est une matrice $n \times p$, $C = (c_{ij})$, avec $c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$

— Quelle est la complexité du calcul d'un élément de la matrice C ?

$O(m)$

— Combien y a-t-il d'éléments dans la matrice C ?

$O(n \times p)$

— Écrire la fonction python `matmult(A,B)`, qui renvoie le produit de deux tableaux numpy A et B, sans utiliser numpy.dot (rappel : `A.shape[0]` est le nombre de lignes et `A.shape[1]` le nombre de colonnes de A) – attention aux indices !

```
def matmult(A, B):
    n = A.shape[0]
    m = A.shape[1]
    p = B.shape[1]
    C = np.zeros([n, p])
    for i in range(n):
        for j in range(p):
            for k in range(m):
                C[i, j] += A[i, k] * B[k, j]
    return C
```

— Calculer la complexité dans le cas meilleur, le cas pire, en moyenne, en fonction de la *taille des données*, ici les dimensions (n, m, p) des matrices A et B.

Meilleur cas = pire cas = $O(m * n * p)$.

Note : En théorie, on peut faire plus efficace. Trouver un algorithme optimal de multiplication de matrices est un problème de recherche ouvert à l'heure actuelle.

1.3 Recherche dans un tableau (5mn)

On vous demande de concevoir une fonction qui recherche *par dichotomie* un élément dans un tableau de n entiers *triés*, et qui renvoie soit son index, soit -1 si l'élément n'est pas présent. Quelle est la complexité asymptotique dans le pire cas en temps d'exécution de cette fonction ? (On peut montrer qu'elle correspond aussi à la complexité moyenne)

Codage en Python :

```
def recherche(T, e):
    i = 0
    j = len(T)
    while j > i:
        # Invariant : si l'élément est dans T, son indice est
        # dans [i, j[.
        pivot = (i + j) // 2
        if T[pivot] == e:
            return pivot
        elif T[pivot] > e:
            j = pivot
        else:
            i = pivot + 1
    return None

# Test :
tableau = [0, 1, 5, 10, 42]
print(recherche(tableau, 0))
print(recherche(tableau, 2))
print(recherche(tableau, 42))
print(recherche(tableau, 43))
```

Complexité : on l'a vu en cours, il suffit d'appliquer le "Master Theorem" avec $k = 1, b = 2, d = 0$: $O(\log n)$ ($d = \log_b k = 0$).

Rappel : Si $f(n) = kf(\lfloor n/b \rfloor) + O(n^d)$, avec $k > 0, b > 1, d \geq 0$ alors

— si $d > \log_b k$, $f(n) = O(n^d)$

— si $d = \log_b k$, $f(n) = O(n^d \log n)$

— si $d < \log_b k$, $f(n) = O(n^{\log_b k})$

2 Structures de données

Une structure de données est un assemblage de types simples (dans notre cas listes, booléens, entiers, flottants, chaînes) permettant de représenter un objet plus complexe (exemples : un tuple d'une relation, un arbre généalogique, un graphe de villes avec leurs distances)

Nous allons créer une structure de données pour représenter un ensemble, c'est-à-dire une liste de valeurs du même type (entier, chaîne de caractères) dans lequel chaque valeur apparaît au plus une fois.

2.1 Structure de données = implantation + constructeurs + testeurs + sélecteurs + modifieurs (10mn)

L'utilisateur de notre structure de données ne devrait pas avoir à se soucier de la représentation interne d'un ensemble (on parle de structure de données "opaque" ou "abstraite").

Lorsqu'on doit choisir une structure de données, avant de se poser la question « *Comment* représenter un ensemble en Python ? » (c'est-à-dire « Quelle structure de données utiliser ? »), il faut se poser la question « *Pourquoi* représenter un ensemble ? », c'est à dire « Quel jeu de fonctions proposer à l'utilisateur ? ».

Nous allons fournir à l'utilisateur des fonctions permettant de modifier ou de lire cette structure de données.

Ces fonctions sont de quatre types :

— les *constructeurs* permettent de créer une structure de données vide ou de créer une structure de données à partir d'un fichier ou d'une autre structure de données ;
— les *testeurs* et *sélecteurs* permettent d'interroger la structure de données sans la modifier (est-ce qu'un élément est présent dans l'ensemble ? combien d'éléments contient l'ensemble ?) ;
— les *modifieurs* permettent de modifier la structure de données (ajouter un élément, en supprimer un, extraire un élément quelconque).

Elles constituent l'*interface* de la structure de données.

2.2 Spécification de l'interface (10mn)

Spécifier l'interface, c'est donner la réponse à la question « Pourquoi ? » ci-dessus, sous la forme de liste de fonctions avec leurs paramètres et leurs valeurs de retour, sans en donner le contenu.

Donner une liste des fonctions de l'interface d'une structure de données « ensemble », par catégorie, avec pour chaque fonction ce qu'elle fait et la liste des paramètres.

Par exemple :
— `cree_ensemble()` renvoie un ensemble vide
— `taille_ensemble()` renvoie le nombre d'éléments (type `int`, positif ou nul)
— `est_present(ensemble, elem)` renvoie *True* si elem est dans ensemble
— `ajoute_element(ensemble, elem)` ajoute un élément à ensemble, renvoie *True* si l'élément était présent, *False* sinon
— `supprime_element(ensemble, elem)` supprime un élément de l'ensemble, renvoie *True* si l'élément était présent, *False* sinon
— `prend_element(ensemble)` renvoie un élément arbitraire et le supprime de l'ensemble – utile par exemple pour afficher le contenu d'un ensemble

2.3 Représentation et implantation (30mn)

Implanter une fonction ou un algorithme, c'est la réponse à la question « Comment ? » ci-dessus.
Il faut choisir quel(s) type(s) Python utiliser, mais aussi comment les valeurs y sont stockées.
Nous proposons d'étudier la représentation d'un ensemble par une liste non triée.

— Donner l'implantation en Python des fonctions suivantes :
  — `est_present(ensemble, elem)` renvoie *True* si elem est présent dans ensemble, *False* sinon
  — `supprime_element(ensemble, elem)` supprime un élément de l'ensemble, renvoie *True* si l'élément était présent, *False* sinon

Voir fichier `ensemble.py` séparé.

— Évaluer la complexité de ces deux fonctions.

  — `est_present(ensemble, elem)`
    — Meilleur cas = $O(1)$ (élément présent en début de liste)
    — Pire cas = $O(n)$ (élément absent ou présent en fin de liste)
  — `supprime_element(ensemble, elem)` se fait en deux temps : recherche de la valeur à supprimer, puis suppression.
    — Meilleur cas = pire cas = $O(n)$ (si l'élément est au début, on paye peu pour la recherche mais plus pour la suppression, et vice-versa).

— Quelle serait la complexité de ces deux fonctions si la liste était toujours triée (attention, la suppression d'un élément nécessite de déplacer tous les éléments suivants) ?

  — `est_present(ensemble, elem)` peut maintenant faire une recherche dichotomique.
    — Pire cas = $O(n)$
    — Meilleur cas = $O(1)$ avec une recherche dichotomique qui s'arrête dès qu'elle trouve l'élément (comme ci-dessus).

— `supprime_element(ensemble, elem)` peut aussi faire une recherche dichotomique, mais la suppression de l'élément a toujours une complexité linéaire en fonction du nombre d'éléments derrière l'élément à supprimer dans la liste.
  — Pire cas = $O(\log n) + O(n) = O(n)$ (si l'élément est en début de liste)
  — Meilleur cas = $O(\log n)$ (si l'élément est en fin de liste, la suppression coûte $O(1)$).

Note : il existe des structures de données et algorithmes beaucoup plus efficaces, mais beaucoup plus complexes que celle proposées ici (chercher « Arbre bicolore » sur Wikipedia). En Python, une structure de données efficace de représentation d'ensemble est disponible dans le langage : c'est la classe `set`.