Exercices de Programmation & Algorithmique 1

Série 8 – Évaluer l'efficacité d'un algorithme — La notion de complexité
(17 novembre 2022)

Département d'Informatique – Faculté des Sciences – UMONS

Pré-requis : invariant de boucles; efficacité des algorithmes; calcul du temps CPU; notation grand-O et complexité dans le pire des cas; complexité des opérations sur les listes (cours jusqu'au **Chapitre 9** compris).

Objectifs : être capable d'évaluer la complexité d'un algorithme en notation grand-O; calculer le temps CPU pris par un script Python et illustrer son évolution en fonction de la taille du problème.

---

1 Le contexte

L'objectif de cette série est de comparer l'efficacité d'algorithmes destinés à résoudre un même problème. Vous allez effectuer cette comparaison pour les deux problèmes suivants :

— *Tri d'une séquence d'éléments* ;
— *Recherche d'un élément donné dans une séquence triée.*

Pour le premier problème (tri), trois algorithmes ont été présentés dans les chapitres 8 et 9 du cours : le tri par sélection, le tri par insertion et le tri par fusion. Pour rappel, pour trier n éléments d'une liste, les deux premiers tris ont une complexité dans le pire de cas en $O(n^2)$, tandis que le tri par fusion possède une complexité en $O(n \log n)$. Durant cette séance, vous allez découvrir un nouvel algorithme de tri (le tri à bulles) et analyser son efficacité dans le pire des cas. Enfin, vous confronterez la théorie à la pratique en comparant sur machine les temps d'exécution CPU des différentes méthodes de tri.

Pour le second problème (recherche d'un élément dans une séquence triée), vous comparerez l'efficacité de la recherche dichotomique et de la recherche linéaire.

1.1 Module `sort` et tris « in place »

Un module `sort` est disponible pour cette séance (fichier `sort.py` à télécharger sur Moodle). Celui-ci contient les implémentations des 3 tris vus au cours :

— le tri par insertion : `insertion_sort(t)` ;
— le tri par sélection : `selection_sort(t)` ;
— le tri par fusion : `merge_sort(t)` ;

Ces trois fonctions de tris ont un effet de bord puisqu'elles ne retournent pas une nouvelle liste, mais modifient la liste passée en paramètre (on parle de tri « *in place* »). Cela signifie que pour trier une liste `t` via, par exemple, le tri par fusion, ce module peut être utilisé comme suit :

```
import sort
...
sort.merge_sort(t)
```

Le module `sort` contient une quatrième fonction de tri `python_sort(t)` et qui s'utilise de la même façon que les trois autres. En réalité, cette fonction se contente de faire appel à la méthode `sort()` « built-in » des listes de la libraire standard Python. Elle vous sera utile pour réaliser l'Exercice 8 ci-dessous.

1.2 Le tri à bulles

Le *tri à bulles* (ou tri par propagation) imite le comportement de bulles d'air dans l'eau : il consiste à faire « remonter » les plus grands éléments d'une liste en comparant, à chaque parcours, deux éléments successifs. Plus précisément, il consiste à réaliser autant de parcours de la liste que nécessaire. Lors d'un parcours, on commence par comparer les 2 premiers éléments. Si ceux-ci ne sont pas correctement ordonnés (c.-à-d., le premier est plus grand que le deuxième), ils sont échangés. Ensuite, le parcours poursuit en comparant le deuxième et le troisième élément et ainsi de suite jusqu'à la fin de la liste.

S'il n'y a eu aucun échange lors d'un parcours, le tri à bulles s'arrête, car cela signifie que les éléments sont triés.

Voici un exemple d'exécution du tri à bulles sur la liste $[4, 1, 3, 2, 5]$

— **Premier parcours** (remarquez la montée de la « bulle » 4) :

$$[\underline{4, 1}, 3, 2, 5] \to [1, \underline{4, 3}, 2, 5] \to [1, 3, \underline{4, 2}, 5] \to [1, 3, 2, \underline{4, 5}] \to [1, 3, 2, 4, 5]$$

— **Deuxième parcours** :

$$[\underline{1, 3}, 2, 4, 5] \to [1, \underline{3, 2}, 4, 5] \to [1, 2, \underline{3, 4}, 5] \to [1, 2, 3, \underline{4, 5}] \to [1, 2, 3, 4, 5]$$

— **Troisième parcours** :

$$[\underline{1, 2}, 3, 4, 5] \to [1, \underline{2, 3}, 4, 5] \to [1, 2, \underline{3, 4}, 5] \to [1, 2, 3, \underline{4, 5}] \to [1, 2, 3, 4, 5]$$

Comme la liste est triée, il n'y a eu aucun échange lors de ce troisième parcours et le tri à bulles s'arrête.

1.3 La recherche dichotomique

Étant donné une séquence triée, la recherche dichotomique permet de chercher si un élément x est présent dans la séquence selon le principe suivant. On calcule l'indice du milieu de la séquence m et on regarde l'élément y correspondant :

— si x est égal à y, alors on l'a trouvé et son indice est m ;
— si x est plus petit que y, alors on doit le chercher dans la première partie de la séquence (avant m) ;
— sinon, x doit se trouver dans la deuxième partie de la séquence (après m).

En répétant ce processus, on réduit la taille des sous-séquences à explorer tant qu'on ne trouve pas x. Si finalement on explore une sous-séquence de taille 0 ou 1 qui ne contient pas x, c'est qu'il ne se trouve pas dans la séquence.

Une implémentation de la recherche dichotomique est reprise ci-dessous. La fonction `dicho_search` retourne l'indice de x s'il est présent, ou `None` sinon. Cette fonction a pour précondition que la liste t est triée (et que les opérateurs de comparaisons peuvent être appliqués sur ses éléments).

```
1  def dicho_search(t, x):
2      start = 0
3      end = len(t) - 1
4      mid = (end - start) // 2
5      while (end - start > 0) and x != t[mid]:
6          if x < t[mid]:
7              end = mid - 1
8          else:
9              start = mid + 1
10         mid = start + (end - start) // 2
11     if len(t) > 0 and x == t[mid]:
12         return mid
13     else:
14         return None
```

2 Le contrat

2.1 À réaliser sur papier

Dans cette partie sur papier, vous allez écrire le pseudocode du tri à bulles et de la recherche linéaire. Ensuite, vous analyserez la complexité dans le pire des cas des différents algorithmes à comparer. Pour analyser la complexité des opérations sur les listes (accès à un élément, l'insertion d'un élément, etc.), vous utiliserez les complexités des listes Python telles que données dans le chapitre 9 du cours (dans la section « évaluer la complexité des algorithmes »).

1 Donnez un pseudocode pour le tri à bulles. Analysez sa complexité en notation grand-O en fonction du nombre n d'éléments à trier. **Aide** : quel est le nombre maximum de parcours effectués ? Quelle est la complexité d'un de ces parcours ?

2 Comment classeriez-vous les quatre algorithmes de tri (sélection, insertion, fusion, bulles) en ce qui concerne l'efficacité ?

3 Prouvez que la recherche dichotomique sur une séquence de n éléments triés possède une complexité dans le pire des cas en $O(\log_2 n)$.

4 Pour trouver un élément dans une séquence (triée ou non) de n éléments, la recherche linéaire consiste à parcourir linéairement la séquence (de gauche à droite) jusqu'à trouver éventuellement l'élément recherché. Quelle est la complexité dans le pire des cas de cet algorithme ?

> **Important** : avant de passer à la suite du contrat (sur machine), vous devez montrer votre pseudocode pour le tri à bulles à un assistant. Cependant, l'assistant ne commentera pas les complexités théoriques que vous avez obtenues dans un premier temps : vous allez d'abord, sur machine, tenter de valider vous-même vos analyses grâce aux temps CPU.

2.2 À réaliser sur machine

Les 3 algorithmes de tri vus au cours et la fonction `dicho_search` sont déjà implémentés dans le module `sort` disponible sur Moodle. Le module `umons_cpu` qui a été également présenté au cours est lui aussi disponible sur Moodle.

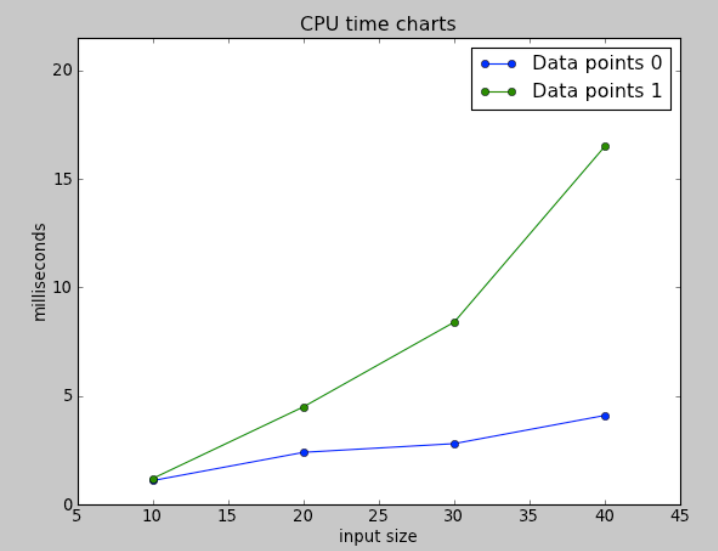
Quand le module `sort` est lancé en mode script, il affiche progressivement un tableau contenant des temps CPU moyens pour trier une liste de n éléments selon trois scénarios :

- *t1* : la liste contient les nombres de 0 à $n-1$ triés par ordre croissant ;
- *t2* : la liste contient les nombres de n à 1 triés par ordre décroissant ;
- *t3* : la liste contient n nombres aléatoires (tirés au hasard entre 0 et n).

Pour chacun de ces scénarios — et chacun des trois tris (`sel` = sélection, `ins` = insertion et `mer` = fusion) — le temps d'exécution moyen (exprimé en millisecondes CPU) est donné. Chaque temps CPU est calculé en réalisant 3 séries de 10 tris et en prenant la meilleure moyenne parmi ces 3 séries (cf. explication donnée au cours).

5 Implémentez le tri à bulle.

6 Écrivez un script permettant d'illustrer et de comparer l'efficacité des (quatre) différents algorithmes de tris via des tableaux de temps CPU et pour différents scénarios, mais également par d'autres moyens laissés à votre discrétion. Vous pouvez par exemple consulter et modifier le script `displayCpu.py` qui permet de produire facilement une figure comme celle représentée ci-dessous.



7 Interprétez vos résultats et confrontez ceux-ci aux complexités théoriques. Retrouvez-vous une croissance des temps CPU dans le pire des cas qui est en accord avec vos analyses ?

8 Comparez votre meilleur algorithme de tri avec la méthode `sort` « built-in » des listes de la libraire standard Python : qui fait mieux ? Pour réaliser cette comparaison, il vous suffit d'utiliser la fonction `python_sort(t)` du module `sort`.

9 Implémentez la recherche linéaire. Ensuite, utilisez le module `umons_cpu` pour comparer les temps d'exécution de la recherche linéaire et de la recherche dichotomique dans une liste triée d'entiers (il suffit d'utiliser `range` pour créer une liste d'entiers triée). Affichez un tableau avec les temps CPU en fonction d'une taille croissante pour les listes et selon 2 scénarios : l'élément à trouver n'est pas présent, l'élément à trouver est présent (mais sa position est aléatoire).

3 Exercices complémentaires

★☆☆ 10 Avec la recherche dichotomique, combien d'itérations seront nécessaires au maximum pour déterminer si un nombre est présent dans une liste triée contenant 1 million d'éléments ?

★★☆ 11 Implémentez les deux algorithmes du calcul de a^n (exposant) vus au cours et comparez les temps d'exécution.

★★☆ 12 Déterminez un invariant de boucle pour la fonction `dicho_search` telle que présentée à la page 2 et prouvez qu'il s'agit bien d'un invariant de boucle. Prouvez ensuite que la fonction `dicho_search` est correcte (preuve d'arrêt et d'exactitude).

★★★ 13 Déterminez des invariants de boucle pour votre implémentation du tri en bulles. Prouvez qu'il s'agit bien d'invariants de boucle. Prouvez ensuite que votre implémentation est correcte (preuve d'arrêt et d'exactitude).

Exam 14 (novembre 2009) Problème 4 : recherche dichotomique (inclus pour référence, mais cette question est déjà reprise dans le contenu de cette série).

Exam 15 (janvier 2010) Problème 1 : preuves d'exactitudes et calcul de complexité de 3 fonctions.

Exam 16 (juin 2010) Problème 2 : pattern matching.

Exam 17 (aout 2010) Problème 3 : preuves et calcul de complexité.

Exam 18 (novembre 2010) Problème 1 : calculs de complexité.

Exam 19 (novembre 2010) Problème 2 : preuve d'exactitude.

Exam 20 (janvier 2011) Problème 2 : calculs de complexité.

Exam 21 (janvier 2011) Problème 3 : Opérateurs sur les entiers (contient des questions sur les invariants).

Exam 22 (juin 2011) Problème 4 : calcul de complexité d'une fonction récursive.

Exam 23 (juin 2012) Problème 1 : multiplication d'entiers (preuves et invariants de boucles).