Atelier d'informatique

Épisode IV : Algorithmes de tri

25 février 2017

Introduction

Dans ce chapitre, on s'intéresse au problème ultra-classique du tri des éléments d'une liste, quand ils sont comparables entre eux : des nombres pour l'ordre usuel sur $\mathbb{R}$, ou des chaînes de caractères pour l'ordre lexicographique (l'ordre des mots dans un dictionnaire).

Introduction

Dans ce chapitre, on s'intéresse au problème ultra-classique du tri des éléments d'une liste, quand ils sont comparables entre eux : des nombres pour l'ordre usuel sur $\mathbb{R}$, ou des chaînes de caractères pour l'ordre lexicographique (l'ordre des mots dans un dictionnaire).

Les algorithmes de tri, suites d'instructions permettant de résoudre ce problème, sont multiples et constituent des sujets d'étude classique pour l'informaticien en herbe :

Introduction

Dans ce chapitre, on s'intéresse au problème ultra-classique du tri des éléments d'une liste, quand ils sont comparables entre eux : des nombres pour l'ordre usuel sur $\mathbb{R}$, ou des chaînes de caractères pour l'ordre lexicographique (l'ordre des mots dans un dictionnaire).

Les algorithmes de tri, suites d'instructions permettant de résoudre ce problème, sont multiples et constituent des sujets d'étude classique pour l'informaticien en herbe : efficacité,

Introduction

Dans ce chapitre, on s'intéresse au problème ultra-classique du tri des éléments d'une liste, quand ils sont comparables entre eux : des nombres pour l'ordre usuel sur $\mathbb{R}$, ou des chaînes de caractères pour l'ordre lexicographique (l'ordre des mots dans un dictionnaire).

Les algorithmes de tri, suites d'instructions permettant de résoudre ce problème, sont multiples et constituent des sujets d'étude classique pour l'informaticien en herbe : efficacité, complexité temporelle (temps d'exécution),

Introduction

Dans ce chapitre, on s'intéresse au problème ultra-classique du tri des éléments d'une liste, quand ils sont comparables entre eux : des nombres pour l'ordre usuel sur $\mathbb{R}$, ou des chaînes de caractères pour l'ordre lexicographique (l'ordre des mots dans un dictionnaire).

Les algorithmes de tri, suites d'instructions permettant de résoudre ce problème, sont multiples et constituent des sujets d'étude classique pour l'informaticien en herbe : efficacité, complexité temporelle (temps d'exécution), stabilité (est-ce que le temps d'exécution augmente beaucoup si je modifie un petit peu ma liste ?).

Introduction

Dans ce chapitre, on s'intéresse au problème ultra-classique du tri des éléments d'une liste, quand ils sont comparables entre eux : des nombres pour l'ordre usuel sur $\mathbb{R}$, ou des chaînes de caractères pour l'ordre lexicographique (l'ordre des mots dans un dictionnaire).

Les algorithmes de tri, suites d'instructions permettant de résoudre ce problème, sont multiples et constituent des sujets d'étude classique pour l'informaticien en herbe : efficacité, complexité temporelle (temps d'exécution), stabilité (est-ce que le temps d'exécution augmente beaucoup si je modifie un petit peu ma liste ?).

On verra dans ce chapitre quelques algorithmes simples, que l'on essaiera de programmer en langage Python.

Tri par sélection

On va commencer par l'algorithme de tri le plus évident de tous. Étant donnée une liste `l`, on sélectionne son plus grand élément, que l'on place à la fin. Maintenant, on recommence sur ce qui reste de la liste. Et ainsi de suite.

Tri par sélection

On va commencer par l'algorithme de tri le plus évident de tous. Étant donnée une liste `l`, on sélectionne son plus grand élément, que l'on place à la fin. Maintenant, on recommence sur ce qui reste de la liste. Et ainsi de suite.

Mais ce n'est pas un tri très efficace au niveau du nombre de calculs à effectuer. En fait, on peut montrer qu'étant donnée une liste l de N éléments, l'algorithme de tri par sélection fait $O(N^2)$ opérations (le nombre d'opérations est au plus CN^2 avec $C > 0$).

Tri par sélection

Implémentation

On va procéder par étapes.

Tri par sélection

Implémentation

On va procéder par étapes.

D'abord, une fonction auxiliaire qui va nous servir pour calculer le maximum d'une liste passée en argument :

Tri par sélection

Implémentation

On va procéder par étapes.

D'abord, une fonction auxiliaire qui va nous servir pour calculer le maximum d'une liste passée en argument :

Exercice 1 (Maximum d'une liste)

Définir une fonction `indice_max` qui prend en argument une liste `l` et renvoie l'indice où trouver son plus grand élément, appelé *maximum*.

Question bonus Si N est le nombre d'éléments de la liste `l`, combien d'opérations (comparaisons, affectations de variables) effectue votre programme ?

Tri par sélection

Implémentation

Maintenant, on écrit le code complet :

Tri par sélection

Implémentation

Maintenant, on écrit le code complet :

```
def tri_selec(l):
    N = len(l)
```

Tri par sélection

Implémentation

Maintenant, on écrit le code complet :

```
def tri_selec(l):
    N = len(l)
    for k in range(N):
```

Tri par sélection

Implémentation

Maintenant, on écrit le code complet :

```
def tri_selec(l):
    N = len(l)
    for k in range(N):
        # On chope l'indice du maximum
        # de la sous-liste allant
        # des incides 0 à N-k-1
        imax = indice_max(l)
```

Tri par sélection

Implémentation

Maintenant, on écrit le code complet :

```
def tri_selec(l):
    N = len(l)
    for k in range(N):
        # On chope l'indice du maximum
        # de la sous-liste allant
        # des incides 0 à N-k-1
        imax = indice_max(l)

        # On échange avec le dernier
        # élément de cette sous-liste
        l[N-k-1], l[imax] = l[imax],l[N-k-1]
```

Question On note N la longueur de la liste `l`. Combien d'opérations fait-on au total ? N'oubliez pas que l'appel à `indice_max(l[N-k-1])` effectue $2(N-k)+1$ opérations !

Tri par sélection

Performances

Alors, on peut montrer mathématiquement que l'algorithme fonctionne parfaitement et triera n'importe quelle liste qu'on lui passe.

Tri par sélection

Performances

Alors, on peut montrer mathématiquement que l'algorithme fonctionne parfaitement et triera n'importe quelle liste qu'on lui passe.

Dans l'exercice suivant, on évalue le temps pris pour effectuer un tri par sélection :

Tri par sélection

Performances

Alors, on peut montrer mathématiquement que l'algorithme fonctionne parfaitement et triera n'importe quelle liste qu'on lui passe.

Dans l'exercice suivant, on évalue le temps pris pour effectuer un tri par sélection :

Exercice 2

Importez les fonctions `time` (détermine l'heure qu'il est) et `randint` (génère des entiers au hasard) via

```
from time import time
from random import randint
```

Exercice 2 (suite)

Définissez la fonction suivante, qui évalue le temps pris par le tri par sélection en fonction du nombre d'éléments dans la liste :

```
def chrono_selec(N):
    # liste de N entiers pris
    # au hasard entre 0 et 10
    l = [randint(0,10) for i in range(N)]
    debut = time()
    l_triee = tri_selec(l)
    temps = time() - debut
    return temps
```

et testez-la pour plusieurs valeurs de N.

Tri par sélection

On désire trier une liste `l` passée en argument de la manière suivante : on part du début de la liste, et on remonte en échangeant chaque élément avec son successeur s'il est plus grand que lui, de sorte qu'à chaque passage, on ait placé les plus grands éléments à la fin de la liste dans l'ordre croissant.

Dans le pire des cas, cet algorithme effectue $O(N^2)$ opérations, avec $N = \mathsf{len}(l)$.

Tri par sélection

Implémentation

En Python, cela donne le code suivant :

```
def tri_bulle(l, debug=False):
N = len(l)
    def remontee(i):
        """Remonte l'élément d'indice i
        tant que c'est possible"""
        if i < N-1:
        if l[i+1] < l[i]:
            l[i+1],l[i] = l[i], l[i+1]
            remontee(i+1)
        else:
            remontee(i+1)
for i in range(N):
    remontee(0)
    if debug: print(l)
```