Tris, complexité et dichotomie

Activité 04

Partie 1 : de l'ordre dans les ordinateurs

Un **algorithme de tri** est un algorithme qui sert à trier (dans l'ordre croissant, généralement) un tableau d'éléments. Il en existe de très nombreux : l'objectif de ce TP est d'aborder les algorithmes les plus classiques, et de les comparer.

Exercice 1 : tri par sélection

L'algorithme du **tri par sélection** est le tri le plus classique à coder.

1. Prendre 10 cartes numérotées de 1 à 10, les mélanger, et les disposer en ligne devant soi.
2. Exécuter l'algorithme suivant :

> mettre un stylo à gauche de la carte la plus à gauche
> **tant que** *le stylo n'est pas tout à droite des cartes* **:**
> ⎸ échanger la carte directement à droite du stylo avec la carte la plus petite parmi celles à droite du stylo (ne pas toucher les autres cartes)
> ⎸ déplacer le stylo d'une carte à droite

3. Réécrire le pseudo-code pour qu'il s'applique à un tableau d'entiers, et non à des cartes avec un stylo.
4. Écrire une fonction `indice_minimum(tab,i)` qui renvoie l'indice de l'élément le plus petit **à droite** de l'indice `i`, **indice `i` inclus**.
5. Écrire une fonction `tri_selection(tab) : list -> list` qui renvoie la version triée de `tab` selon l'algorithme du tri par sélection.

Exercice 2 : tri par insertion

1. Prendre 10 cartes numérotées de 1 à 10, les mélanger, et les disposer en ligne devant soi.
2. Exécuter l'algorithme suivant :

> prendre toutes les mains dans la main gauche
> **tant que** *la main gauche n'est pas vide* **:**
> ⎸ prendre la carte la plus à droite de la main gauche
> ⎸ la mettre dans la main droite de sorte à avoir toujours une main droite triée (en **insérant** la carte correctement dans la main)

3. Intuitivement, c'est très facile à faire avec des cartes : la traduction en tableau est sensiblement plus pénible. Remélanger vos cartes, et exécuter cette version de l'algorithme du tri par insertion :

> mettre un stylo tout à droite des cartes
> **tant que** *le stylo n'est pas tout à gauche des cartes* **:**
> ⎸ décaler le stylo d'un cran à gauche
> ⎸ décaler la carte ajoutée à la droite du stylo en l'échangeant de proche en proche jusqu'à ce qu'elle soit plus petite que sa voisine de droite

4. Réécrire le pseudo-code pour qu'il s'applique à un tableau d'entiers, et non à des cartes avec un stylo.
5. Écrire une fonction `insertion(tab,i)` qui prend un tableau `tab` et un indice `i`, et insère `tab[i]` dans `tab[i:]`, en remontant `tab[i]` le long de la tranche jusqu'à ce que la case soit plus petite que la case à sa droite.

Par exemple : considérons `tab = [5,3,1,2,4,6]`, et exécutons `insertion(tab,1)`. Comme `i = 1`, on ne modifie pas la case d'indice 0. Puis on observe que 3 > 1, donc on échange les deux cases, et on se retrouve avec `[5,1,3,2,4,6]` ; on observe maintenant que 3 > 2, donc on échange les deux cases, et on se retrouve avec `[5,1,2,3,4,6]` ; on observe que 3 < 4, et on arrête nos opérations. En particulier, on ne compare pas 3 et 6.

6. Écrire une fonction `tri_insertion(tab) : list -> list` qui renvoie la version triée de `tab` selon l'algorithme du tri par insertion.

Exercice 3

Désormais, comparons les deux programmes : lequel est le plus rapide ? On va commencer par apprendre comment tracer des courbes en Python.

1. Pour tracer des courbes, nous allons utiliser la bibliothèque `matplotlib.pyplot`. Dans le fichier, importer la bibliothèque sous l'alias `plt` à l'aide de la commande `import matplotlib.pyplot as plt`.
2. Pour tracer une courbe, il faut un tableau des abscisses et un tableau des ordonnées (les deux tableaux doivent absolument être de mêmes longueurs). Construire deux tableaux :
   — un tableau `X` contenant les entiers de 1 à 100 ;
   — un tableau `Y` contenant les racines carrées des entiers de 1 à 100.

Bonus. Pour construire `X`, on aurait pu utiliser la construction des tableaux **par compréhension**, propre à Python : `X = [i for i in range(1,101)]`. Trouver comment construire `Y` par compréhension.

3. Pour terminer, recopier le script suivant :

```
plt.figure() # initie la création d'une figure

plt.plot(X,Y) # associe les abscisses et les ordonnées

plt.show() # ouvre la fenêtre graphique
```

Pour pouvoir comparer nos algorithmes, nous allons procéder aux étapes suivantes : nous allons construire des tableaux à `n` éléments (pour `n` entre 1 et 200), les mélanger, et chronométrer le temps mis par les deux algorithmes pour trier les tableaux en question.

Exercice 4

Il existe une bibliothèque très classique pour générer du hasard, la bibliothèque `random`.

1. Importer `random` sous l'alias `rd`.
2. Tester plusieurs fois la fonction `rd.randint(0,10)` : que fait-elle ?
3. Créer un tableau `tab` contenant les entiers de 1 à 10, et tester la commande `rd.shuffle(t)` : que fait-elle ? Donner la signature de `rd.shuffle`.
4. L'algorithme derrière `rd.shuffle` est plutôt simple et efficace : il s'agit du mélange de Yates-Fisher. Implémenter le code suivant en Python.

Données : un entier n
créer un tableau tab des n premiers entiers
pour $i = 0$ **à** $n - 2$ **:**
  $j \leftarrow$ indice aléatoire entre i et $n - 1$
  échanger les cases `tab[i]` et `tab[j]`
retourner tab

Cet algorithme produit tous les tableaux possibles avec probabilité égale.

Exercice 5

Enfin, nous allons mesurer le temps mis par nos algorithmes de tri pour trier nos tableaux. Pour cela, nous allons devoir mesurer des temps d'exécution : on a besoin d'une nouvelle bibliothèque !

1. Importer la bibliothèque `time`.
2. Tester la fonction `perf_counter_ns()` : cette fonction mesure le temps sur votre machine.

Le protocole pour construire l'ordonnée d'un point à tracer sera le suivant :

---

générer un tableau de longueur n
en créer une copie
initialiser une variable t_0 avec `perf_counter_ns`
trier le tableau original avec le tri sélection
mesurer dans une variable $t_{\text{sélection}}$ le temps mis par le tri par sélection : il s'agit du temps actuel (mesuré par `perf_counter_ns()`) auquel on soustrait la valeur retenue dans t_0
répéter pour mesurer le temps mis par le tri par insertion sur la copie du tableau précédent
répéter ce processus 20 fois
retourner *le temps mis moyen mis par les deux tris sur les 20 tentatives dans deux tableaux séparés*

---

3. Utiliser ce protocole pour mesurer et tracer les deux courbes. Pour tracer deux courbes sur une même figure, il suffit de faire deux `plot`, un par courbe : `plot(X,Y)` puis `plot(X,Z)`.
4. Quel tri est le plus rapide ? Est-ce particulièrement remarquable ?

Partie 2 : par la recherche, pour la recherche

Un **algorithme de recherche** est un algorithme qui sert à savoir si un élément précis se trouve dans un tableau. Par exemple, est-ce que 4 est un élément de `[1,3,5,6]` ?

Exercice 6

L'algorithme de **recherche séquentielle** est l'algorithme le plus classique possible pour résoudre ce problème : il consiste à parcourir toutes les cases du tableau pour vérifier si l'une des cases vaut ce qu'on cherche.

1. Écrire une fonction `recherche_sequentielle(tab,elem)` renvoyant `True` si `elem` est dans `tab`.
2. Analysons l'algorithme en question lors de son exécution :
   a) Combien de comparaisons effectue-t-on au minimum ? À quel cas cela correspond-il ?
   b) Combien de comparaisons effectue-t-on au maximum ? À quel cas cela correspond-il ?
   c) On suppose que l'élément est dans le tableau. Combien de comparaisons effectue-t-on en moyenne ?

Exercice 7

L'algorithme de **recherche dichotomique** permet de chercher efficacement un élément dans un tableau trié.

1. Prendre un dictionnaire, et chercher à l'aide de l'algorithme de recherche séquentielle le mot « aberration ». Cette méthode serait-il elle efficace pour chercher le mot « logarithme » ?
2. Chercher le mot « logarithme » dans votre dictionnaire de manière plus efficace. Essayer de le faire sans utiliser les marques extérieures permettant de repérer l'alphabet : vos seules opérations devraient être "ouvrir une page" et "lire un mot sur une page".
3. La recherche dichotomique consiste en l'algorithme suivant :

Données : un tableau tab et un élément $elem$
tant que *on n'a pas trouvé* $elem$ *et* tab *contient encore des éléments* **:**
    regarder au milieu de tab
    si $elem$ *se trouve au milieu de* tab **:**
        retourner *vrai*
    sinon, si *le milieu est plus grand que* $elem$ **:**
        continuer la recherche dans la partie gauche de tab
    sinon
        continuer la recherche dans la partie droite de tab
si on est sorti du « tant que », c'est que $elem$ n'est pas dans tab :
retourner *faux*

Tester la recherche dichotomique pour trouver le mot « logarithme » dans le dictionnaire.

4. Cet algorithme fonctionnerait-il si le dictionnaire n'était pas trié dans l'ordre lexicographique ?
5. Écrire `recherche_dicho(tab, elem)` qui renvoie `True` si `elem` est un élément de `tab` et `False` sinon.
6. Tester l'algorithme dans les cas suivants :
    — l'élément cherché est à l'une des extrêmités du tableau ;
    — l'élément cherché n'est pas dans le tableau ;
    — le tableau n'est pas trié (que se passe-t-il à ce moment-là ?).

Exercice 8

Pour terminer, on souhaiterait comparer la recherche séquentielle et la recherche dichotomique.
Adapter le protocole de l'exercice 5 pour comparer les deux algorithmes. Cette fois, pour une longueur donnée n, le tableau sera inchangé à chaque nouvelle tentative : c'est l'élément à chercher qui sera choisi au hasard (parmi les éléments du tableau).
Tester la comparaison sur des tableaux de taille maximale 1000. Lequel de ces algorithmes est le plus efficace ?

Pour la prochaine fois :

— On considère un string composé exclusivement des caractère `-`, puis de (au moins un) caractères `*` (par exemple `"---***"`, `"-----*"` ou `"****"`). Écrire une fonction `premiere_etoile(str)` qui renvoie l'indice de la première étoile de `str` à l'aide de la recherche dichotomique.
— Python connaît déjà l'ordre lexicographique : `"gauss" > "euler"` renvoie `False`, par exemple. Écrire une fonction `trier_classe(tab)` qui trie un tableau de couples `(nom,prenom)` dans l'ordre alphabétique des noms, puis des prénoms.
— On considère une nouvelle relation d'ordre sur les strings : plus un string contient la lettre `"a"`, plus il est grand. Écrire une fonction `tri_bizarre(tab)` qui trie un tableau de strings dans l'ordre croissant selon cette étrange relation d'ordre.