TP 11 - ALGORITHMES DE TRI

Info1.Algo1 - 2022-2023 Semestre Impair

Versions itératives des algorithmes (sur tableaux)

Exercice 1 - Tri par sélection (itératif)

Dans le fichier **ex01__tri__selection__iteratif.py**, compléter le corps de la fonction `tri_selection` de façon à implémenter l'algorithme de **tri par sélection** vu en cours.

Contraintes :

- Ne modifier le tableau qu'à l'aide de la **fonction `permuter`** fournie, afin de garantir que les éléments du tableau ont été conservés.
- Utiliser le **modèle de code** suivant pour la boucle principale :

```
assert True, 'Pre-condition'
...
assert est_trie(tab,i), 'Invariant'
while ...:
  ...
  assert est_trie(tab,i), 'Invariant'
assert est_trie(tab,len(tab)), 'Post-condition'
```

où `est_trie` est la fonction auxiliaire qui accepte en paramètres un tableau `tab` ainsi qu'un entier `i` et vérifie que la tranche `tab[:i]` est triée.

Exercice 2 - Tri par insertion (itératif)

Dans le fichier **ex02__tri__insertion__iteratif.py**, compléter le corps de la fonction `tri_insertion` de façon à implémenter l'algorithme de **tri par insertion** vu en cours.

Contraintes :

- Ne modifier le tableau qu'à l'aide de la **fonction `permuter`** fournie, afin de garantir que les éléments du tableau ont été conservés.
- Utiliser le **modèle de code** suivant pour la boucle principale :

```
assert True, 'Pre-condition'
```

```
...
assert est_trie(tab,i), 'Invariant'
while ...:
  ...
  assert est_trie(tab,i), 'Invariant'
assert est_trie(tab,len(tab)), 'Post-condition'
```

où `est_trie` est la fonction auxiliaire qui accepte en paramètres un tableau `tab` ainsi qu'un entier `i` et vérifie que la tranche `tab[:i]` est triée.

Versions récursives des algorithmes (sur listes chaînées)

Exercice 3 - Tri par sélection (récursif)

1) Dans le fichier **ex03_tri_selection_recursif.py**, écrire la fonction récursive `extraire_minimum` qui accepte en paramètre une **liste chaînée** non vide, **extrait** le minimum de cette liste chaînée et retourne un tuple constitué du minimum trouvé ainsi que de la liste chaînée ainsi **modifiée**.

> **Attention ! : On conserve l'ordre des éléments dans la liste privée du minimum**.

Exemple : Si la liste chaînée `liste` est `(7,(1,(6,(2,None))))`, alors le minimum extrait est `1` et la liste modifiée est `(7,(6,(2,None)))`.

Indications :

- *Cas d'arrêt* : la liste a un seul élement.
  - C'est le minimum, et on retourne ce minimum accompagné de la liste vide.
- *Cas récursif* : la queue de liste n'est pas vide, on peut donc extraire récursivement son minimum.
  - Si la tête de liste est plus petite que le minimum trouvé, on a trouvé le minimum, donc on retourne l'élement en tête accompagné de la queue de la liste initiale.
  - Sinon, le minimum est celui trouvé récursivement et on ajoute la tête à la liste privée du minimum.

2) Compléter la fonction récursive `tri_selection`, qui accepte en paramètre une **liste chaînée** et retourne cette liste une fois triée.

Indications :

Pour son cas récursif, la fonction `tri_selection` :

- Extrait le minimum accompagné de la liste privée du minimum.
- Trie récursivement la liste privée du minimum.

- Retourne la liste construite avec le minimum trouvé et la list récursivement triée.

Exercice 4 - Tri par insertion (récursif)

1) Dans le fichier **ex04__tri__insertion__recursif.py**, compléter le corps de la fonction récursive `inserer_dans_liste_triee` qui accepte en paramètres :

- Une liste chaînée `liste` triée par **ordre croissant**.
- Une `valeur`.

La fonction insère cette nouvelle valeur dans la liste chaînée, de telle façon que la liste reste triée par ordre croissant.

Exemple : Si la liste chaînée est `(3,(4,(6,None)))` et la valeur est `5`, la liste retournée est `(3,(4,(5,(6,None))))`.

Indications :

- Cas d'arrêt :
  - Si la liste est vide : on retourne une nouvelle liste contenant uniquement la valeur à ajouter.
  - Si la valeur à insérer est inférieure à la tête de la liste : on l'ajoute en tête.
- Cas récursif :
  - On retourne une liste avec la tête qui n'a pas bougé et la queue de liste dans lequel on a positionné la valeur à insérer.

2) Compléter la fonction récursive `tri_insertion`, qui accepte en paramètre une **liste chaînée** et retourne cette liste une fois triée.

Indications :

Le **cas récursif** de cette fonction procède aux étapes suivantes :

- **Tri par insertion** de la queue de liste.
- **Insertion** de la tête de liste dans la queue ainsi triée.

Exercice 5 - Tri fusion (récursif)

1) Dans le fichier **ex05__tri__fusion__recursif.py**, écrire la fonction récursive `partager_liste` qui accepte en paramètre une variable `liste` et retourne deux listes chaînées `liste1` et `liste2` composées chacune d'un élément sur deux de `liste`.

Indications :

- Le partage en deux se fait selon un principe semblable à celui de la fermeture éclair.
- Le premier élément de `liste1` doit être identique au premier élément de `liste`.

Autrement dit : `tete(liste1)==tete(liste)`.

Exemple : Si `liste` est `(1,(6,(8,(3,None))))`, alors les deux listes `liste1` et `liste2` sont respectivement `(1,(8,None))` et `(6,(3,None))`.

2) Écrire la fonction récursive `fusionner_listes_triees` qui accepte en paramètre deux listes chaînées `liste1` et `liste2` triées en ordre croissant et retourne la liste chaînée `liste` triée elle aussi en ordre croissant et résultat de la fusion des listes chaînées `liste1` et `liste2` .

Exemple : Si les deux listes chaînées `liste1` et `liste2` sont respectivement `(1,(6,None))` et `(3,(8,None))`, alors la liste chaînée `liste` est `(1,(3,(6,(8,None))))`.

3) Écrire la fonction récursive `tri_fusion` qui accepte en paramètre une liste chaînée `liste` et retourne cette liste une fois triée par l'algorithme de **tri fusion**.

Rappel : Le principe du tri fusion est basé sur la stratégie **diviser pour régner**. Étant donnée une liste d'éléments à trier, le principe de l'algorithme est le suivant :

- **Partage** de la liste en deux sous-liste.
- **Tri fusion** de chacune des deux sous-listes afin d'obtenir deux sous-listes triées.
- **Fusion** des deux sous-listes triées en une seule liste triée.

Exercice 6 - Tri rapide (récursif)

L'objectif de cet exercice est de mettre en oeuvre l'algorithme de **tri rapide** sur une liste chaînée. Le **tri rapide** adopte la stratégie **diviser pour régner** : étant donnée une liste d'éléments à trier, le principe de l'algorithme est le suivant :

- **Partition** de la liste en utilisant la méthode du pivot : on obtient deux sous-listes (valeurs inférieures/supérieures au pivot).
- **Tri rapide** de chacune des deux sous-listes : on obtient deux sous-listes triées.
- **Concaténation** des deux sous-listes ainsi triées en une seule liste triée (on intercale le pivot entre les deux sous-listes triées).

Le tri rapide est un algorithme de tri très utilisé pour sa relative simplicité et sa rapidité.

1) Dans le fichier **ex06__tri__rapide__recursif.py**, écrire la fonction récursive `partitionner_pivot` qui accepte en paramètre une variable `liste` ainsi qu'une valeur de `pivot` et retourne deux listes chaînées `liste_inf` et `liste_sup` composées respectivement des éléments de `liste` qui sont inférieurs ou égaux à `pivot` et de ceux qui sont strictement supérieurs à `pivot`.

Exemple : Si la liste chaînée `liste` est `(8,(2,(7,(9,None))))` et que le `pivot` est `7` , alors `liste_inf` et `liste_sup` sont respectivement `(2,(7,None))`

et `(8,(9,None))`.

2) Écrire la fonction récursive `concatener_listes` qui accepte en paramètre deux listes chaînées `liste1` et `liste2` et retourne la liste chaînée `liste`, résultat de la concaténation des listes chaînées `liste1` et `liste2` .

Exemple : Si les deux listes chaînées `liste1` et `liste2` sont respectivement `(8,(4,None))` et `(7,(1,None))`, alors la liste chaînée `liste` est `(8,(4,(7,(1,None))))`.

3) Écrire la fonction récursive `tri_rapide` qui accepte en paramètre une liste chaînée `liste` et retourne cette liste une fois triée par l'algorithme de **tri rapide**.

> **Attention !** Dans la mise en oeuvre de la fonction `tri_rapide`, on choisit la **tête de liste** comme pivot. Afin de garantir la terminaison de la récursion, la partition selon le pivot ne doit pas être effectuée sur la liste toute entière mais **uniquement sur la queue de liste**.

Mesures de complexité

Exercice 07 - Tri par sélection (complexité)

1) Dans le fichier **ex07_tri_selection_complexite.py**, compléter le corps de la fonction `tri_selection` de façon à implémenter l'algorithme de **tri par sélection** vu en cours.

> Dans cet exercice, afin de ne pas fausser la mesure de complexité, **ne pas vérifier de propriété (pré-, post-condition ou invariant) à l'aide d'assertion**.

2) a) Compléter le programme afin de mesurer le temps d'exécution de l'appel `tri_selection(tab)`, où `tab` est un tableau aléatoire de taille `n` fourni par l'appel `permutation_aleatoire(n)`. Choisir les valeurs de `n` selon des puissances croissantes de 10. **Relever les temps d'exécution** obtenus.

b) Comment évolue le temps d'exécution de l'appel `tri_selection(tab)` lorsque la taille `n` du tableau passé en paramètre est multipliée par `10` ? **Interpréter**.

Exercice 08 - Tri par insertion (complexité)

1) Dans le fichier **ex08_tri_insertion_complexite.py**, compléter le corps de la fonction `tri_insertion` de façon à implémenter l'algorithme de **tri par insertion** vu en cours.

> Dans cet exercice, afin de ne pas fausser la mesure de complexité, **ne pas vérifier de propriété (pré-, post-condition ou invariant) à l'aide d'assertion**.

2) a) Compléter le programme afin de mesurer le temps d'exécution de l'appel `tri_insertion(tab)`, où `tab` est un tableau aléatoire de taille `n` fourni par

l'appel `permutation_aleatoire(n)`. Choisir les valeurs de `n` selon des puissances croissantes de 10. **Relever les temps d'exécution** obtenus.

b) Comment évolue le temps d'exécution de l'appel `tri_insertion(tab)` lorsque la taille `n` du tableau passé en paramètre est multipliée par `10` ? **Interpréter**.

Exercice 09 - Tri fusion (complexité)

1) Dans le fichier **ex09__tri__fusion__complexite.py**, compléter le corps des fonctions `partager_liste`, `fusionner_listes_triees` et `tri_fusion`.

2) On souhaite dans cette question analyser expérimentalement la **complexité** en temps de l'algorithme de tri fusion.

Pour cela, on donne dans le fichier **ex09__tri__fusion__complexite.py** la fonction `creer_liste_aleatoire` qui accepte en paramètre une `longueur` et retourne une liste chaînée composée des entiers de `0` à `longueur-1` et mélangée de façon aléatoire.

a) Écrire un **programme principal** qui, pour des entiers `n` allant de `100` à `995` :

- Génère une liste aléatoire de longueur `n`.
- Mesure le temps (en secondes) nécessaire pour réaliser le tri fusion de cette liste.
- Affiche l'entier `n` et le temps mesuré.

b) Que se passe-t-il si l'on tente d'augmenter la valeur de `n` au-delà de `1000` ? Interpréter.

c) On souhaite enregistrer les données mesurées dans un fichier CSV (**Comma-Separated Values**). Pour cela, adapter le code écrit à question **a** afin qu'il suive le patron de code suivant :

```
with open('tri_fusion_complexite.csv','w') as f:
  f.write('n,sec\n')
  for n in ...:
    # Génération d'une liste aléatoire de longueur n
    # Mesure du temps d'exécution
    temps = ...
    f.write('{},{}\n'.format(n,temps))
```

Lors de l'exécution, le programme doit alors créer le fichier **tri__fusion__complexite.csv** dans le répertoire où il s'exécute. Vérifier dans un éditeur de texte brut que ce fichier est formaté de la façon suivante :

```
n,sec
100,0.0006110668182373047
101,0.0005900859832763672
[...]
994,0.010184288024902344
```

```
995,0.00917673110961914
```

Pour plus d'informations au sujet du format CSV ainsi généré, vous pouvez consulter la page :

https://fr.wikipedia.org/wiki/Comma-separated_values

d) Ouvrir le fichier **tri_fusion_complexite.csv** avec un tableur (*LibreOffice Calc*, *Microsoft Excel* ou *Google Sheets*, selon disponibilité sur la machine utilisée). Les valeurs de `n` ainsi que les temps mesurés doivent apparaître dans les colonnes **A** et **B** du tableur.

Créer un graphique dont les abscisses sont les valeurs de `n` et les ordonnées sont les temps mesurés. La courbe observée correspond-elle à la courbe théorique prévue pour le tri fusion ?

Variantes et autres algorithmes

Exercice 10 - Tri à bulles

1) Dans le fichier **ex10_tri_bulles.py**, compléter le corps des fonctions `tri_bulles_etape` et `tri_bulles`.

2) a) Compléter le programme afin de mesurer le temps d'exécution de l'appel `tri_bulles(tab)`, où `tab` est un tableau aléatoire de taille `n` fourni par l'appel `permutation_aleatoire(n)`. Choisir les valeurs de `n` selon des puissances croissantes de 10. **Relever les temps d'exécution** obtenus.

b) Comment évolue le temps d'exécution de l'appel `tri_bulles(tab)` lorsque la taille `n` du tableau passé en paramètre est multipliée par `10` ?

c) Afin de confirmer par le calcul ce résultat expérimental, exprimer en fonction de `i` le nombre d'itérations effectuées dans la fonction `tri_bulles_etape`. En déduire la complexité en temps de l'algorithme de tri à bulles.

Exercice 11 - Tri cocktail

Le **tri cocktail** est une variante du tri à bulles :

- Lors d'un **tri à bulles**, on effectue tous les parcours du tableau de gauche à droite.
- Lors d'un **tri cocktail**, on effectue alternativement un parcours du tableau de gauche à droite (sens **aller**), puis un parcours de droite à gauche (sens **retour**), etc.

Pour chacun des passages, la zone restant à traiter est délimitée par deux indices : `i_gauche` et `i_droite`.

> **Attention !** : Afin d'être cohérent avec la notation Python déjà utilisée dans d'autres algorithmes de tri, l'indice `i_gauche` est **in-**

clus dans la zone restant à traiter, tandis que l'indice `i_droite` est **exclus**. Cette zone restant à traiter correspond donc à la tranche `tab[i_gauche:i_droite]`.

Lors des parcours **aller** (de gauche à droite), le **plus grand élément** de la zone restant à traiter est poussé **vers la droite**.

Lors des parcours **retour** (de droite à gauche), le **plus petit élément** de la zone restant à traiter est poussé **vers la gauche**.

Le schéma ci-dessous montre le déroulé de l'algorithme sur le tableau `[7,4,2,9,6,1]`, la zone restant à traiter est grisée :

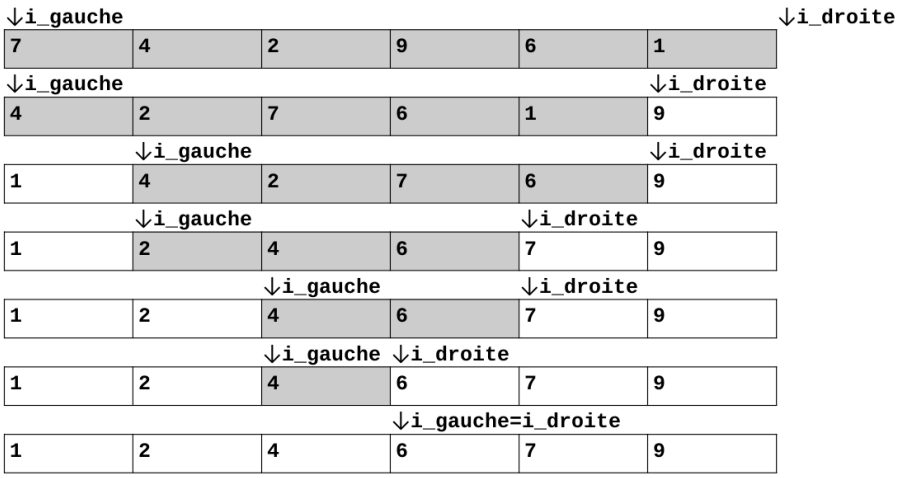

Figure 1: déroulé sur le tableau `[7,4,2,9,6,1]`

1) a) Dans le fichier **ex11_tri_cocktail.py**, compléter les fonctions `tri_cocktail_aller` et `tri_cocktail_retour` qui permettent d'effectuer les parcours **aller** et **retour**.

b) Compléter la fonction `tri_cocktail` qui alterne les appels aux fonction `tri_cocktail_aller` et `tri_cocktail_retour` afin de réaliser le tri du tableau.

2) a) Compléter le programme afin de mesurer le temps d'exécution de l'appel `tri_cocktail(tab)`, où `tab` est un tableau aléatoire de taille `n` fourni par l'appel`permutation_aleatoire(n)`. Choisir les valeurs de `n` selon des puissances croissantes de 10. **Relever les temps d'exécution** obtenus.

b) Comment évolue le temps d'exécution de l'appel `tri_cocktail(tab)` lorsque la taille `n` du tableau passé en paramètre est multipliée par `10` ?

c) Afin de confirmer par le calcul ce résultat expérimental, exprimer en fonction de `i_gauche` et `i_droite` le nombre d'itérations effectuées dans la fonction

`tri_cocktail_aller` ainsi que dans la fonction `tri_cocktail_retour`. En déduire la complexité en temps de l'algorithme de tri cocktail.

Exercice 12 - Indices

Dans le fichier **ex12_indices_tri.py**, compléter la fonction `indices_tri` qui accepte en entrée un `tableau` d'entiers de taille `n` et retourne un tableau `indices` permettant de savoir l'ordre des indices dans lesquels les éléments de `tableau` sont triés.

Ainsi, lorsque `i` va de `0` à `n-1`, les valeurs `tableau[indices[i]]` doivent être croissantes.

Exemple : Si `tableau` vaut `[10,7,9,6,8]`, les valeurs sont, dans l'ordre :

- `6`, d'indice `3`
- `7`, d'indice `1`
- `8`, d'indice `4`
- `9`, d'indice `2`
- `10`, d'indice `0`

Le tableau `indices` retourné est donc `[3,1,4,2,0]`

Contraintes :

- Le tableau `indices` doit être créé et utilisé en respectant l'**abstraction tableau**.
- Le `tableau` passé en paramètre **ne doit être ni copié ni modifié**.

Méthode conseillée :

Reprendre l'algorithme de **tri par insertion** avec les modifications suivantes :

- Rassembler l'algorithme de tri dans une seule fonction.
- Le tableau `indices` est initialisé avec les indices initiaux `0`,`1`,`2`,..., `n-1`.
- Les permutations sont effectuées sur `indices` et non sur `tableau`.
- On accède à une valeur du tableau grâce à l'accès indirect `tableau[indices[i]]`.