Tris (version tronc commun)

C. Charignon

Trier un tableau est un exercice classique et très formateur. De nombreuses méthodes existent, dont trois sont officiellement au programme, qui ont chacune leurs avantages et inconvénients.

Table des matières

Première partie

Cours

1 Révision : recherche dichotomique dans un tableau trié

Un des principaux intérêts d'avoir trié un tableau est d'être capable d'y rechercher très rapidement un élément. L'algorithme de dichotomie a été vu en première année, révisons-le.

2 Critères d'appréciation d'un tri

Bien sûr le principal critère pour juger l'efficacité d'un tri est sa complexité. On étudiera la complexité au pire des cas, bien que la complexité moyenne puisse être plus parlante (mais plus compliquée à calculer...). En général, on étudiera la complexité en nombre de comparaisons, mais il faudra quand même essayer de garder à l'esprit les autres opérations (lectures/écritures dans le tableau notamment).

On se posera également la question de la complexité en espace.

Un point de vocabulaire : une méthode de tri sera dite « en place » lorsque toutes les opérations sont effectuées au sein du tableau de départ. Ainsi la complexité en espace sera minimale dans ce cas.

En général, une telle méthode fonctionnera par effet de bord, elle est donc de type `list -> None`.

Par exemple, calculons la médiane d'un tableau.

Avec un tri par effet de bord :

```
def mediane1(T):
    triEnPlace(T) #triEnPlace ne renvoie rien
    return(T[len(T)//2])
```

Avec un tri qui renvoie un nouveau tableau trié :

```
def mediane2(T):
    S = triPasEnPlace(T) #triPasEnPlace renvoie un nouveau tableau
    return( S[len(T)//2])
```

La première version peut être source de piège : le tableau fourni pas l'utilisateur a été trié à son insu !

On pourrait préférer ceci :

```
import copy
def mediane3(T)
    """calcule la mediane en utilisant triEnPlace,
    mais sans modifier le tableau fourni en entrée."""
    S=copy.deepcopy(T)
    triEnPlace(S)
    return(S[len(S)//2])
```

3 Un premier tri

Lorsque je demande à un étudiant au hasard de me proposer une méthode pour trier un tableau, plus de neuf fois sur dix il me propose l'algorithme suivant : on cherche le plus grand élément et on le place à la fin du tableau. Puis on cherche le deuxième plus grand et on le met en avant dernière position. Etc. Formalisé, ceci donne :

Soit t un tableau et n sa longueur.

Pour tout $i \in [\![0, n[\![$:

Chercher le plus grand élément de $t[0 : n-i]$ et le placer en case $n-1-i$.

Remarquons déjà que ce tri est en place, il n'occupe donc quasiment aucune mémoire supplémentaire.

En ce qui concerne sa complexité : Pour tout $i \in [\![0, n[\![$ la recherche du plus grand élément dans $t[0 : n-i]$ nécessite $n-i-1$ comparaisons. Donc le nombre total de comparaisons nécessaires est $\sum_{i=0}^{n-1}(n-i-1)$ qui vaut $\sum_{k=0}^{n-1} k$, soit encore $\frac{n(n-1)}{2}$, et ce dans tous les cas. On a donc une complexité en $O(n^2)$.

Nous ne programmerons pas ce tri dans le cours, mais nous allons voir si nous pouvons faire mieux.

4 Tri par insertion

Le tri par insertion est le tri du joueur de carte : la main gauche du joueur contient les cartes déjà triées, et sa main droite les cartes à trier. Donc initialement la main gauche est vide et toutes les cartes sont dans la main droite. On prend chaque carte de la main droite l'une après l'autre, et on l'insère à la bonne place dans la main gauche.

Il est relativement facile de programmer cet algorithme en place : nous allons utiliser un seul tableau ; sa partie gauche sera la main gauche du joueur de carte, et sa partie droite la main droite. Une variable `i` nous indiquera où est la séparation entre les deux mains.

4.1 L'algorithme principal

Précisons l'idée indiquée ci-dessus, c'est-à-dire explicitons les invariants de boucle !

On va utiliser une variable i entière telle qu'à chaque instant `t[0:i]` soit trié. Ainsi, à chaque étape il suffira de prendre l'élément `t[i]` et de l'insérer à sa place dans `t[0:i]`.

Commençons par écrire le programme principal ; nous verrons l'insertion ensuite.

N.B. Écrire la fonction principale avant la fonction auxiliaire permet de déterminer exactement les spécifications de la fonction auxiliaire dont nous aurons besoin, avant de rédiger celle-ci. C'est souvent le plus efficace.

```
1 def triInsertionEnPlace(t):
2     for i in range (len (t)):
3         # Invariant de boucle : t[0:i] est trié et contient les mêmes éléments qu'au début de
      ↪ la procédure.
4         insereEnPlace(t,i)
```

On notera que du moment que notre procédure auxiliaire permet bien de respecter l'invariant de boucle, la démonstration de la correction de `triInsertionEnPlace` est immédiate. En effet, en sortie de boucle, si je note n la longueur du tableau `t` passé en argument, on a par l'invariant de boucle que `t[0:n]`, c'est-à-dire `t` est trié et contient les mêmes éléments qu'au début.

4.2 Programmer l'insertion

Pour insérer un élément dans un tableau, il faut décaler tous les éléments suivants d'une case vers la droite.

Rédigeons maintenant l'insertion. En écrivant le programme principal, nous avons pu constater que le rôle précis de cette procédure est de prendre en entrée un tableau `t` et un indice `i` tel que `t[0:i]` est trié, et d'insérer l'élément `t[i]` à sa place.

Voici une première procédure, chargée de d'effectuer les déplacements nécessaires.

```
1 def décale(t, a, b):
2     """ Procédure qui décale les éléments de t[a:b] d'une case vers la droite, et transfère t[
      ↪ b] en t[a]."""
3     tmp=t[b]
4     for i in range(b, a, -1):
5         t[i]=t[i-1]
6     t[a]=tmp
```

N.B. Il faut être attentif pour réussir ce programme... Prêtez bien attention à la position des -1. Faites des dessins !

Pour obtenir la procédure d'insertion, il nous suffit maintenant de rechercher la place où envoyer l'élément à insérer, puis à appeler la procédure `décale` :

```
1 def insereEnPlace(T,n):
2     """
```

```
    Précondition : T[0:n] est trié.
    Effet  : insère T[n] dans T[0:n], de sorte que T[0:n+1] soit trié.
    """
    i=n
    # On cherche la place de T[n]
    while i > 0 and T[i-1] > T[n] :
        i-=1
        # Invariant de boucle : ici les éléments de T[i:n] sont > T[n].
    # Sortie de boucle : T[i-1] <= T[n] et les éléments de T[i:n] sont > T[n]

    # et on ramène x à sa place
    décale(T,i,n)
```

Vous avez remarqué que je commence la recherce de la future place de `T[n]` par n, puis redescends (ou revient vers la gauche) vers 0. On aurait pu faire l'inverse : faire partir `i` de 0 et l'augmenter jusqu'à ce que `T[i]>= T[n]`. Cette dernière solution aurait optimisé le cas où le tableau est initialement trié à l'envers, alors que la solution retenue optimise le cas où le tableau est initialement trié à l'endroit. On verra dans la suite du cours que l'intérêt du tri par insertion est qu'il est efficace lorsque le tableau est presque trié, d'où l'intérêt d'optimiser ce cas là.

4.3 Complexité

La complexité spatiale est négligeable devant la taille du tableau initial puisque l'algorithme est en place. Voyons la complexité temporelle.

4.3.1 Complexité au pire

Complexité de la fonction intermédiaire : Soit t un tableau, et n sa longueur. Pour tout $i \in [\![0, n[\![$, l'appel à `insereEnPlace(t,i)` coûte entre 1 et $i-1$ comparaisons. Le minimum est atteint si $t[i]$ est le maximum de $t[0:i+1]$, et le maximum est atteint lorsque $t[i]$ est le minimum (ou le deuxième plus petit élément) de $t[0:i+1]$.

Le pire des cas est celui d'un tableau trié à l'envers. En effet, dans ce cas pour tout $i \in [\![0, n[\![$, `insereEnPlace(t,i)` est dans le pire cas, donc nécessite $i-1$ comparaisons. Ce qui nous fait un total de

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O_{n\to\infty}(n^2).$$

Remarque : On peut aussi facilement compter le nombre d'opérations sur le tableau : pour déplacer un élément de la place i à la place 0, il faut $i+1$ lectures dans le tableau et autant d'écritures ($i+2$ opérations de lectures écriture si on compte la variable temporaire utilisée), donc un total de $\sum_{i=1}^{n}(i+1) = \frac{n(2+n+1)}{2} = O(n^2)$ lectures-écritures.

Remarque : On peut prouver que la formule $\sum_{i=0}^{n} O_{i\to\infty}(u_i) = O_{n\to\infty}\left(\sum_{i=0}^{n} u_i\right)$ est valable lorsque la série $\sum_i u_i$ diverge et est à termes positifs, ce qui est le cas ici. On pourrait alors rédiger plus simplement ainsi :

« Pour tout $i \in [\![0, n[\![$, l'insertion de `T[i]` nécessite $O(i)$ opérations de lecture écriture. Donc la complexité totale est $\sum_{i=0}^{n-1} O_{i\to\infty}(i)$, ce qui fait $O_{n\to\infty}(n^2)$.

4.3.2 Complexité au mieux

5 Partition - fusion

5.1 Principe

On passe maintenant à une méthode basée sur la stratégie « diviser pour régner ». Profitons-en pour rappeler cette stratégie :

1. (« Diviser ») Diviser le problème en problèmes plus petits ;
2. Résoudre récursivement chaque petit problème ;
3. (« Régner ») Rassembler les solutions des petits problèmes en une solution du problème initial.

Ici, cela va donner :

1. Couper le tableau en deux parties de longueurs égales ou presque ;
2. Trier récursivement chaque partie,
3. Rassembler les deux tableaux triés en un tableau trié. Cette étape s'appellera la « fusion » .

L'idée de départ est plus astucieuse que pour le tri par insertion, mais la mise en œuvre concrète est plus simple.

5.2 Programmation

Le point clef est la fonction de fusion. Celle doit prendre en entrée deux tableaux triés et renvoyer le tableau trié contenant l'union des contenus de ces deux tableaux.

```
def fusion(t1,t2):
    """ Précondition : t1 et t2 sont triés.
        Sortie : fusion de t1 et t2 """
    res=[] #pour contenir le résultat
    i1=0   #indice du prochain élément de t1 à  prendre
    i2=0   # "      "    "        "     " t2  "   "

    while i1<len(t1) and i2 < len(t2) :
        if t1[i1] < t2[i2]:
            res.append(t1[i1])
            i1+=1
        else:
            res.append(t2[i2])
            i2+=1
    # sortie de boucle:
    # Un des deux tableaux a été utilisé en totalité
    # Reste à rajouter le reste de l'autre tableau
    res.extend(t2[i2:])
    res.extend(t1[i1:])

    return res
```

On déduit alors le programme principal. J'ai fait apparaître distinctement les trois étapes. Ne pas oublier un cas d'arrêt, puisqu'il s'agit d'une fonction récursive.

```
def triFusion(t):
    """ Fonction qui renvoie un nouveau tableau contenant les éléments de t, mais dans l'ordre
    ↪ ."""

    n=len(t)

    if n<=1:
        return t
    else:
        # 1) On divise le tableau en deux
        t1, t2 = t[0:n//2], t[n//2:n]

        # 2) On trie chaque morceau
        t1Trié = triFusion(t1)
        t2Trié = triFusion(t2)

        # 3) On fusionne les deux moitiés de tableau triées
        return fusion(t1Trié, t2Trié)
```

Version en place : voir TD, exercice 9.

5.3 Complexité temporelle

Avant de commencer, un petit lemme, utile lorsqu'on découpe un entier en deux entiers :

Lemme 5.1. *Soit $n \in \mathbb{N}$. Alors :*

$$n = \left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n+1}{2} \right\rfloor$$

Ainsi, lorsqu'on divise une liste de longueur l en deux partie égales à plus ou moins 1 près, un morceau sera de longueur $\left\lfloor \frac{n}{2} \right\rfloor$ et l'autre de longueur $\left\lfloor \frac{n+1}{2} \right\rfloor$.

Démonstration :

- Si n est impair : soit $k \in \mathbb{N}$ tel que $n = 2k$, alors :

$$\frac{n}{2} + \left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{2k}{2} \right\rfloor + \left\lfloor \frac{2k+1}{2} \right\rfloor = k + k = n$$

- Si n est impair : soit $k \in \mathbb{N}$ tel que $n = 2k+1$, alors :

$$\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n+1}{2} \right\rfloor = \left\lfloor \frac{2k+1}{2} \right\rfloor + \left\lfloor \frac{2k+2}{2} \right\rfloor = k + k + 1 = n$$

□

Comptons le nombre de comparaisons. On commence par les fonctions auxiliaires :

- Aucune comparaison dans `partition`.
- On constate que le nombre de comparaisons dans `fusion l1 l2` est au plus la longueur de $l1$ + longueur de $l2$.

Passons à la fonction principale. Notons, pour tout $n \in \mathbb{N}$, C_n le nombre *maximal* de comparaisons dans l'appel de `tri` sur une liste de longueur *au plus* l. On constate alors que pour tout $n \in \mathbb{N}$:

$$C_n \leqslant n + C_{\left\lfloor \frac{n}{2} \right\rfloor} + C_{\left\lfloor \frac{n+1}{2} \right\rfloor}$$

N.B. Relation de récurrence typique lors de l'étude de la complexité d'un algorithme de type "diviser pour régner".

Cas d'une puissance de 2 : la relation est plus simple si n est une puissance de 2. En effet pour tout $k \in \mathbb{N}$:

$$C_{2^k} \leqslant 2^k + 2C_{2^{k-1}}$$

Et en divisant par 2^k :

$$\frac{C_{2^k}}{2^k} \leqslant 1 + \frac{C_{2^{k-1}}}{2^{k-1}}$$

Et la suite $\left(\frac{C_{2^k}}{2^k}\right)_{n\in\mathbb{N}}$ est arithmétique de raison 1. Son premier terme est $\frac{C_1}{1} = 0$ (le cas d'une liste de longueur 1 est un des cas de base), on obtient $\forall k \in \mathbb{N}$, $\frac{C_{2^k}}{2^k} \leqslant k$, donc :

$$C_{2^k} \leqslant k2^k.$$

Remarque : En gros, si on pouvait remplacer ci-dessus k par $\log_2(n)$, on obtiendrait : $C_n \leqslant n\log_2(n)$
Mais ce n'est pas possible car $\log_2(n)$ n'est en général pas entier !

Pour faire le calcul propre, nous allons encadrer n entre deux puissances de 2 : Soit $k \in \mathbb{N}$ tel que :

$$2^k \leq n < 2^{k+1} \tag{1}$$

En fait, cet entier k vérifie $k \leqslant \log_2(n) < k+1$, donc $k = \left\lfloor \log_2(n) \right\rfloor$.

Lemme 5.2. *La suite C est croissante.*

Démonstration : C'est du au fait qu'on a défini C_n comme étant la complexité maximale pour des listes de longueur **au plus** n. □

En utilisant le lemme, il vient :

$$C_{2^k} \leq C_n \leq C_{2^{k+1}}$$
$$\text{donc } C_n \leq (k+1)2^{k+1}$$

Or, d'après 1, on déduit $k \leq \log_2(n)$ donc :

$$C_n \leq (\log_2(n)+1)2^{\log_2(n)+1}$$
$$\leq C_n \leq 2n(\log_2(n)+1)$$

D'où on tire $C_n = O_{n\to\infty}\left(n\log(n)\right)$.

Ainsi, le tri fusion a une complexité de l'ordre de $n\log_(n)$. En particulier, cette complexité est négligeable devant la complexité au pire du tri par insertion.

Remarque : On peut démontrer que $O\left(n\log(n)\right)$ est la complexité optimale pour le tri d'une liste de longueur n.

5.4 Complexité spatiale

6 Tri rapide, ou de Hoare

Le principe de ce tri est très proche du tri fusion : on partage le tableau en deux, on tri séparément chaque moitié, puis on rassemble les deux parties. La différence est qu'on ne partage pas le tableau en deux moitiés de même longueur : on choisit un élément appelé pivot (le plus simple est de choisir le premier élément du tableau) on sépare le tableau en deux parties : les éléments inférieurs au pivot, et les élément supérieurs au pivot.

Pour rassembler les deux morceaux une fois chacun d'eux triés une simple concaténation suffit, puisque les éléments du premier morceaux sont tous inférieurs à ceux du deuxième morceau.

N.B. Le pivot doit être retiré de la liste à trier, sans quoi on risque de faire planter le programme dans le cas où le pivot serait le maximum ou le minimum de la liste.

6.1 Version naïve

Commençons par la segmentation :

```
def segmente(t, pivot):
    """ Renvoie deux tableaux : les éléments <= pivot, et les éléments > pivot."""


    t1, t2= [], []
    for i in range(t):
        if t[i] <= pivot:
            t1.append(t[i])
        else:
            t2.append(t[i])
    return t1, t2
```

Bonus : avec la boucle for spéciale Python qui permet de se passer des indices

```
def segmente(t, pivot):
    """ Renvoie deux tableaux : les éléments <= pivot, et les éléments > pivot."""
    t1, t2= [], []
    for x in t:
        if x <= pivot:
            t1.append(x)
        else:
            t2.append(x)
    return t1, t2
```

Puis le programme principal :

```
def triSeg(t):
    if len(t) <2 : #cas d'arrêt
        return t

    else:
        pivot=t.pop() #On prend le dernier venu comme pivot
        t1, t2 =segmente(t, pivot)
        return triSeg(t1) + [pivot]+   triSeg(t2)
```

Remarque : Utilisation de mémoire. Cas où le tableau est toujours partagé en deux, puis cas le pire.

6.2 Complexité au pire

Comptons les comparaisons au pire dans le tri rapide.

Soit T un tableau et n sa longueur.

L'opération de segmentation nécessite n comparaisons. Puis il y a les deux appels récursifs.

Le pire des cas est obtenu lorsque l'opération de segmentation conduit à un sous-tableau de longueur 1, et un autre de longueur $n-1$. C'est le cas par exemple lorsque le tableau est déjà trié, ou déjà trié à l'envers.

Plaçons-nous dans ce cas, et notons alors pour tout $k \in \mathbb{N}$ C_k la complexité pour trier k éléments. La relation de récurrence obtenue est :

$$\forall k \in \mathbb{N}, C_k \leqslant k + C_{k-1}$$

Donc au pire des cas, $C_n = \sum_{k=1}^{n} k = \frac{n(n-1)}{2} = O_{n\to\infty}(n^2)$.

Remarque : Pour remédier à ce problème, une stratégie est de choisir comme pivot non pas le premier élément venu mais la médiane du tableau. Mais ceci nécessite de savoir calculer une médiane rapidement... **cf exercice :** 17, 19

Remarque : Le meilleur des cas est celui où le tableau se trouve divisé à chaque étape en deux sous-tableaux de longueur identique ou presque. Dans ce cas, la relation de récurrence est $C_n = C_{\lfloor n/2 \rfloor} + C_{\lfloor \frac{n-1}{2} \rfloor} + n - 1$. C'est quasiment la même que celle du tri fusion, on aura donc $C_n = O(n \log n)$.

Ainsi, l'implémentation naïve du tri par segmentation que nous avons écrite n'a pas d'intérêt face au tri fusion.

6.3 En place

L'intérêt essentiel du tri rapide est le fait qu'il peut être efficacement implémenté en place. C'est pourquoi il est fréquemment utilisé.

Le point délicat est la partition : une fois choisi un pivot, nous devons placer à gauche les éléments inférieurs au pivot, et à droite ceux supérieurs au pivot, le pivot lui-même arrivant entre les deux.

A ce moment le pivot sera à sa place, il n'y aura plus qu'à trier récursivement la partie gauche et la partie droite. Pas d'opération de fusion à faire, puisque tout est déjà en place.

Remarque : Nous ne savons pas à priori combien il y aura d'éléments à gauche et à droite du pivot, nous ne pouvons donc pas savoir la place finale du pivot.

Enfin le but est de réaliser la partition en $O(n)$ comparaisons. En effet, intuitivement, il suffit de comparer le pivot une fois à chacun des autres éléments.

6.3.1 La fonction principale

Commençons par le programme principal. Nous allons effectuer la segmentation, puis ré-appeler le programme sur les deux morceaux obtenus. Puisque nous voulons travailler en place, nous ne pouvons pas créer deux nouveaux petits tableaux sur lesquels ré-appeler notre programme ! Pour résoudre ce problème, nous allons toujours transmettre le tableau entier lors des appels récursifs, mais nous ajouterons deux indices qui indiquent quelle partie du tableau nous traitons.

Ainsi, nous écrirons une procédure `triEntre` prenant en entrée un tableau `t` et deux indices `deb` et `fin`, qui s'occupe de trier `t[deb:fin]`.

On constate qu'on a besoin de savoir où arrive le pivot lors de la segmentation. Ainsi, le programme chargé de la segmentation devra renvoyer la position finale du pivot.

Remarque : Exemple typique de situation où on a bien fait de commencer par la fonction principale, afin de savoir exactement quelle fonction auxiliaire il nous faudra.

```
def triEntre(t,deb,fin):
    """ Trie en place t[deb:fin]."""
    if fin - deb >1:
        iPivot = segmenteEntre(t, deb, fin, deb)
        triEntre(t, deb, iPivot)
        triEntre(t, iPivot+1, fin) # Le pivot est déjà en place. Ne pas le remettre dans l'
    ↪ appel récursif
```

```
7
8     # sinon : rien à faire!
```

6.3.2 Segmentation

On passe alors au programme de segmentation. Ce programme devra à la fois modifier le tableau fourni, et renvoyer la position finale du pivot : on peut dire qu'il s'agit à la fois d'une fonction et d'une procédure.

Pour commencer, voici une procédure pour permuter deux cases d'un tableau :

```
1 def permute(t,i,j):
2      """échange les cases i et j dans le tableau T"""
3      temp=t[i]
4      t[i]=t[j]
5      t[j]=temp
6
7 # ou alors, version pythonesque:
8 def permute(t,i,j):
9     t[i], t[j] = t[j], t[i]
```

Et enfin, la segmentation proprement dite. C'est le point clef de l'algorithme. Il convient d'être soigneux sur les invariants de boucle, et de faire un dessin pour les avoir sous les yeux.

```
1 def segmenteEntre(t,deb,fin, iPivot):
2     """ Effectue la segmentation en place de t[deb:fin]
3      *et* renvoie la position finale du pivot."""
4
5     pivot=t[iPivot]
6     transpose(t,deb, iPivot) # On met le pivot au début
7
8     i= deb+1
9     j= fin
10
11    while  i< j :
12        # invariant de boucle:
13        #   - le pivot est dans t[deb]
14        #   - les éléments t[deb:i] sont <= pivot
15        #   - les éléments de t[j:fin] sont > pivot
16        if t[i] > pivot:
17            transpose(t,i,j-1)
18            j-=1
19        else:
20            i+=1
21
22    # sortie de boucle:
23    # i==j
24    # et par l'invariant:
25    # les éléments de t[deb:i] sont <= pivot
26    # Les éléments de t[i:fin] sont > pivot
27    transpose(t, deb, i-1)
28    return i-1
```

7 En résumé, utilité de chaque tri

Pour schématiser :

- Le tri fusion a une bonne complexité dans tous les cas. Son défaut est de consommer de la mémoire. C'est en outre celui des trois le plus facile à programmer.
- Le tri rapide a une mauvaise complexité au pire. Par contre sa complexité moyenne est bonne. De plus il n'utilise pas de mémoire supplémentaire car il est en place. Les tests montrent qu'il est en moyenne plus rapide que le tri fusion.

- Le tri par insertion a a priori une mauvaise complexité. Cependant, il sera utile lorsque le tableau est presque trié (justement quand le tri rapide est mauvais). C'est une situation qui peut arriver assez vite : si on maintient un tableau trié et qu'on le modifie un peu de temps en temps (typiquement : base de donnée). **cf exercice :** 13 En outre, il est en place.

D'autres méthodes de tris élémentaires figurent dans la feuille de TD :

- le tri bulle
  - « If you know what bubble sort is, wipe it from your mind ; if you don't know, make a point of never finding out ! » (Numerical Recipes in C : The Art of Scientific Computing)
  - « bubble sort seems to have nothing to recommend it » (D. Knuth, The art of computer programming, vol 3)
  - https://www.youtube.com/watch?v=k4RRi_ntQc8
- le tri par extraction **cf exercice :** 10

Deuxième partie

Exercices

Exercices : tris

1 Divers

Exercice 1. * Tester si un tableau est trié

Écrire un prédicat prenant en entrée un tableau t et indiquant si celui-ci est trié.

Exercice 2. * Permutation circulaire

1. Écrire une fonction `permuteDroite` prenant en arguments un tableau T, deux entiers i et j et qui effectue une permutation circulaire sur $T[i:j+1]$: les éléments $T[i], ..., T[j-1]$ sont décalé vers la droite, et $T[j]$ arrive en $T[i]$.

   On modifiera T en place.
2. Écrire la fonction similaire `permuteGauche`.
3. Écrire une fonction `transpose` qui prend en arguments un tableau T, deux entiers i et j et qui échange les contenus de $T[i]$ et $T[j]$.
4. Réécrire les fonctions `permuteDroite` et `permuteGauche` à l'aide de `transpose`. Quelle méthode est la plus efficace ?

Exercice 3. * Inverser l'ordre des éléments d'un tableau

Écrire une fonction pour inverser l'ordre des éléments d'un tableau. On fera une première version qui crée un autre tableau, et une seconde version en place.

Calculer la complexité de ces deux fonctions.

Exercice 4. * Mélange de Knuth

Le mélange de Knuth permet de mélanger un tableau, en faisant en sorte que toutes les permutations possibles soient équiprobables. Voici le principe : soit t un tableau et n sa longueur. Pour tout $i \in [\![0, n[\![$, on choisit un entier $j \in [\![i, n[\![$ avec probabilité uniforme et on échange $t[i]$ et $t[j]$.

1. Programmer cet algorithme.
2. (***) Vérifier que toutes les permutations sont équiprobables.

2 Applications

Exercice 5. ** Distance minimale entre deux éléments d'une liste

On désire écrire une fonction qui prend en entrée une liste d'entiers et qui renvoie la distance minimale entre deux éléments de cette liste.

1. Quelle serait la complexité de la méthode naïve ?
2. Écrire une fonction qui commence par trier la liste, et donner sa complexité.

Exercice 6. **! Dédoublonnage

Soit `t` un tableau, contenant éventuellement des éléments en double. On désire créer un nouveau tableau `t'` qui contiendra les mêmes éléments que `t` mais où chaque élément apparaît une seule fois.

1. Écrire une première version utilisant la fonction « `in` » de Python. Quelle est sa complexité ?
2. On propose une autre stratégie : on commence par trier le tableau. Écrire la fonction correspondante, et donner sa complexité. Quelle condition doit vérifier la complexité du tri employé pour que cette méthode vaille le coup ?

Exercice 7. * Élément dans un maximum d'intervalles**

Un intervalle sera représenté par un couple `(debut, fin)`. Le but est d'écrire une fonction prenant en entrée une liste d'intervalles et de déterminer un élément présent dans un maximum d'intervalles.

Une fois appliqué un tri, on pourra répondre à la question en $O(n)$, si n est le nombre d'intervalles. Si vous ne trouvez pas, une deuxième indication est présente à la fin de la feuille de TD.

3 Tri fusion

Exercice 8. * ! Nombre d'inversions**

Soit t un tableau, n sa longueur, $a_0, ..., a_{l-1}$ ses éléments dans l'ordre. On rappelle qu'une inversion de t est un couple $(i, j) \in [\![0, n-1]\!]^2$ tel que $i < j$ et $a_i > a_j$. Le but de l'exercice est de calculer le nombre d'inversions dans une liste.

Remarque : Le nombre d'inversion peut donner une mesure du « désordonnement » du tableau. On rappelle aussi que la signature de t vaut $(-1)^{\text{nombre d'inversions de l}}$.

1. *Méthode naïve :* Écrire une fonction `plusPetitQue` qui calcule le nombre d'éléments dans t qui sont inférieurs à x. En déduire une fonction calculant le nombre d'inversions dans une liste.

   Calculer la complexité de cette fonction.

2. *Méthode diviser pour régner :* On peut calculer le nombre d'inversions de manière plus efficace en reprenant la structure du tri fusion.

   Le principe est le suivant : soit $t1$ la première moitié de t et $t2$ la seconde moitié. Lors de la fusion de $t1$ avec $t2$, à chaque fois qu'on insère un élément venant de $t2$, il était en inversion avec tous les éléments précédemment inséré venant de $t1$.

   Programmer cette méthode, puis donner l'ordre de grandeur du nombre de comparaisons effectuées.

Exercice 9. ** Tri fusion en place

1. Écrire une version en place du tri fusion. On utilisera une fonction auxiliaire `triEntre` comme pour le tri par segmentation.
2. Une petite optimisation : on peut facilement tester si par hasard à l'issue de la partition et du tri des deux morceaux, les éléments de gauche sont déjà tous plus petits que les éléments de droite, et dans ce cas éviter la fusion.

   Apporter cette amélioration. Dans quels cas sera-ce utile ?

4 Autres tris

Exercice 10. ** Tri par extraction

Le principe du tri par extraction est le suivant : on recherche le maximum du tableau, et on le place à la bonne place. Puis on recherche le maximum dans le restant du tableau, etc...

Programmer cette méthode de tri, en place.

Exercice 11. ** Tri bulle

https://www.youtube.com/watch?v=koMpGeZpu4Q

Pour trier un tableau de n éléments, on effectuera n parcours de ce tableau. A chaque passage, si on rencontre un indice i tel que $t[i] > t[i+1]$ on permute $t[i]$ et $t[i+1]$.

On se convainc aisément qu'après le premier passage, le maximum est arrivé à sa place, après le deuxième passage le deuxième élément aussi, etc...

1. Programmer le tri bulle sur listes ou sur tableau.
2. Quelle est sa complexité au pire ? Au mieux ?

5 Tri par insertion

Exercice 12. *! Complexité du tri par insertion en nombre d'opérations sur les tableaux

Compter le nombre maximal d'écritures dans le tableau pour le tri par insertion.

Exercice 13. ** Complexité du tri par insertion sur un tableau presque trié

Pour tout $k \in \mathbb{N}$ on dit qu'un tableau T est k-presque trié lorsque :

- Ses éléments sont à au plus k places de leur place finale dans le tableau trié
- ou il y a au plus k éléments non triés.

Démontrer que la complexité de tri par insertion d'un tableau k-presque trié est $O(kn)$ (et donc $O_{n\to\infty}(n)$ si on fixe k).

Exercice 14. * ! Insertion dichotomique**

On peut optimiser le tri par insertion en recherchant l'emplacement où insérer l'élément par dichotomie.

1. Programmer cette amélioration.
2. Calculer la complexité en nombre de comparaisons.
3. Le tri par insertion dichotomique est assez peu utilisé en pratique. Pouvez-vous deviner pourquoi ?

6 Tri rapide ou de Hoare

Exercice 15. ** Une optimisation simple

Pour réduire les probabilités de tomber sur un mauvais cas lors de la segmentation, voici une manière de choisir le pivot : on tire au hasard trois élément, et on choisit l'élément médian de ces trois. Programmer cette optimisation.

Exercice 16. ** Insertion lorsqu'il y a peu d'éléments

Écrire une fonction pour comparer le temps d'exécution du tri par insertion et du tri rapide pour des tableaux de petite taille. Estimer à partir de quelle taille de tableau le tri rapide est meilleur que le tri par insertion.

En déduire une optimisation du tri rapide.

Exercice 17. **! Calcul de médiane

Pour simplifier, on conviendra que la médiane d'un tableau T de longueur n est l'élément `T[N//2]`. Ce qui signifie que dans le cas où n est pair, on choisit l'élément "du dessus". (Un autre choix possible serait `(T[n//2]+T[n//2-1])/2`.)

1. Écrire une fonction pour calculer la médiane d'un tableau. Quelle est sa complexité ?

   Une méthode plus efficace consiste à adapter le tri rapide. On garde la partie « segmentation ». Mais ensuite une fois qu'on dispose des deux sous-tableaux, on peut facilement déterminer dans lequel de ces sous-tableaux il faudra rechercher la médiane, ce qui permettra de ré-appeler récursivement la fonction.

   Cependant, la médiane du tableau de départ ne sera pas la médiane du sous-tableau dans lequel on poursuit la recherche : le calcul de médiane passe mal à la récursivité. C'est pourquoi on commencera par écrire une fonction récursive `iemeElement` *prenant en argument un tableau T et un entier $i \in [\![0, len(T)-1]\!]$, deux entiers deb et fin et qui renvoie le i-ème plus petit élément de t* en supposant qu'il est dans $T[deb : fin]$.

2. Programmer cette fonction `iemeElement`.
3. Comment utiliser `eiemeElement` pour calculer le minimum ou le maximum d'un tableau ? Est-ce judicieux ?
4. Quelle est la complexité au pire de cette méthode ?

Exercice 18. * Complexité moyenne du tri rapide**

On notera, pour tout $n \in \mathbb{N}$, C_n la complexité moyenne du tri rapide sur un tableau à n éléments, en nombre de comparaisons entre éléments du tableau. Le but de l'exercice est de déterminer un équivalent de C_n.

1. Préciser C_0 et C_1.
2. Soit $n \in [\![2, \infty[\![$. Quelle est la complexité de la segmentation pour un tableau (ou un sous-tableau en pratique) à n éléments ?
3. Soit $n \in [\![2, \infty[\![$. On suppose que lors de la segmentation, toutes les places finales possibles du pivot sont équiprobables.
   (a) Justifier que :
   $$C_n = \frac{1}{n}\sum_{i=0}^{n-1}(C_i + C_{n-1-i}) + n - 1.$$
   (b) Puis que :
   $$C_n = \frac{2}{n}\sum_{i=0}^{n-1} C_i + n - 1.$$
4. Déduire que pour tout $n \in [\![2, \infty[\![$:
   $$C_n = C_{n-1} \times \frac{n+1}{n} + 2\frac{(n-1)}{n}.$$
5. On étudie alors la suite $\left(\frac{C_n}{n+1}\right)_{n\in\mathbb{N}}$. Démontrer que $\forall n \in \mathbb{N}^*$,
   $$\frac{C_n}{n+1} = \sum_{k=2}^{n} 2\frac{k-1}{k(k+1)}$$

6. Finalement, obtenir un équivalent de $\left(C_n\right)_{n\in\mathbb{N}}$. Il va sans dire que tout théorème mathématique utilisé devra être cité et ses hypothèses clairement vérifiées.

Exercice 19. ** Optimisation du tri rapide par la médiane des médianes (prolongement de l'exercice 17)**

Voici une méthode pour déterminer un pivot afin d'optimiser le tri rapide. Il nécessite un calcul de médiane, nous allons donc écrire simultanément une fonction de tri et une fonction de calcul de médiane. Ce sujet est à la base de plusieurs problèmes de concours...

L'idée est de calculer dans un premier temps une médiane approchée.

On divise le tableau en sous-tableaux de longueur au plus 5, et on calcule la médiane de chacun de ces sous-tableaux. Puis on calcule récursivement la (vraie) médiane de ce tableau des médianes.

Remarque : On procédera autant que possible en place : quand on dit "diviser le tableau" c'est purement abstrait. En réalité on utilisera une fonction `medianeEntre(T,deb,fin)` appelé avec les paramètres `deb, fin` tels que `fin - deb`$\leqslant 5$.

1. Écrire une fonction `medianeDesMedianesEntre` prenant en arguments le tableau `T` et les indices `deb` et `fin` et renvoyant la médiane des médianes de `T[deb:fin]`. Pour la médiane de chaque sous-tableau de 5 éléments, on utilisera un algorithme de médiane non optimisé (en pratique, on choisit généralement celle basée sur le tri par insertion, plus rapide sur des tableaux de moins de 5 éléments.).

   Pour la médiane des médianes, dans la version finale du programme on utilisera (récursivement) la fonction de médiane optimisée qu'on va écrire. En attendant, pour faire des tests, on pourra utiliser une fonction non optimisée quelconque.
2. Quelle sera la complexité du tri rapide selon cette méthode dans le cas extrême où tous les éléments de T sont identiques ?

   Dans la suite, on suppose à l'opposé que les éléments de T sont deux à deux distincts.
3. Combien d'éléments au moins seront inférieurs au pivot renvoyé ? Et combien au moins seront supérieurs ?

   On supposera pour simplifier que la longueur de T est multiple de 10.
4. Écrire une fonction médiane récursive basée sur la méthode de l'exercice précédent mais utilisant comme pivot la "médiane approchée" calculée précédemment.
5. Notons pour tout $n \in \mathbb{N}^*$ $C(n)$ le nombre maximal de comparaisons pour calculer la médiane d'un tableau de taille n. Écrire la relation de récurrence vérifiée par la complexité de cette fonction. Montrer qu'il existe une constante c telle que pour tout $n \in \mathbb{N}$:

$$C(n) \leqslant C(\frac{n}{5}) + C(\frac{7}{10}n) + cn$$

   On supposera encore pour simplifier n multiple de 10.
6. En déduire que $\forall k \in \mathbb{N}$, $C(10^k) \leqslant 10c10^k$.
7. En déduire que la complexité du calcul de médiane ainsi programmé est linéaire. On admettra que C est croissante.
8. Finalement, écrire le tri rapide ainsi optimisé. Écrire la relation de récurrence vérifiée par la complexité au pire de votre fonction. Faire quelques tests pour comparer aux tris déjà vu en cours.
9. Commentaires sur la complexité mémoire ?

Remarque : Ceci est la méthode de tri a priori la plus efficace. Cependant, on s'en tient souvent au tri rapide simple en choisissant un pivot au hasard car pour la plupart des tableaux c'est tout aussi efficace et évite les calculs supplémentaires de médiane approchée. Une méthode plus sophistiquée consiste à essayer un tri rapide simple, puis à passer à l'utilisation de la médiane des médianes si on se rend compte qu'il est inefficace.

Quelques indications

2 1) Réfléchir au sens dans lequel parcourir `T[i:j]`.

3 Dans la version en place, attention à ne pas inverser deux fois les éléments, ce qui ferait revenir à l'ordre de départ.

7 On propose de créer une liste contenant toutes les bornes d'intervalles triées par ordre croissant, auxquelles on adjoindra un booléen `fin` qui indique s'il s'agit du début ou de la fin d'un intervalle. On pourra conclure en un seul parcours de cette liste.

10 On peut utiliser une boucle sur un compteur i, et maintenir l'invariant « les éléments de `t[i:]` sont à leur place finale ».
On peut écrire une fonction auxiliaire `maxDeb` qui prend en entrée un tableau `t` et un indice `fin` et qui renvoie l'indice du maximum de `t[:fin]`.

12 Le cas le pire est le même que pour le nombre de comparaisons.

14 3) étudier le nombre de lectures/écriture.

15 Écrire une fonction externe `choixPivot` prenant en entrée `t`, `deb`, et `fin` et renvoyant l'indice du pivot à utiliser.

17 Lisez attentivement les spécifications suggérées pour la fonction `iemeElement` ! Elles sont choisies pour que le codage soit le plus simple possible.

Quelques solutions

1

2

3

4 2) Demandez à votre professeur de mathématiques, et transmettez-moi sa réponse.

5

6

7

8

9

10

11

12 On trouve $\frac{n(n+1)}{2} = O_{n\to\infty}(n^2)$.

13

14

15

16

17 4) Le pire des cas n'a pas changé : celui d'un tableau trié ou trié à l'envers. Et dans ce cas, la complexité est toujours en $O(n^2)$.

18

1. $C_0 = C_1 = 0$: ce sont les cas d'arrêt.
2. Lorsqu'on appelle `segmente` sur une plage de n éléments d'un tableau, le pivot est comparé à tous les autres éléments, ce qui fait $n-1$ comparaisons entre éléments du tableau.
3. (a) Soit T un tableau, l sa longueur, soit `deb` $\in [\![0, n-1]\!]$, voyons ce qui se passe lorsqu'on effectue `triEntre(T,deb,deb+n)`.
   D'abord, on appelle `segmente(T,deb,deb+n)`, qui a une complexité de $n-1$. Notons i le nombre de cases après `deb` où arrive le pivot à l'issue de la segmentation. Donc $i \in [\![0, n-1]\!]$. (Le résultat renvoyé par`segmente(T,deb,deb+n)` est donc `deb+i`).
   On appelle ensuite `triEntre(T,deb,deb+i)` qui a pour complexité moyenne C_i, et `triEntre(T,deb+i+1,deb+n)` qui a pour complexité moyenne C_{n-i-1}.
   Au total, la complexité moyenne de `triEntre(T,deb,deb+n)` lorsque le pivot arrive en `deb+i` vaut $C_i + C_{n-i-1} + n - 1$.

   Maintenant, nous n'avons plus qu'à faire la moyenne de ces valeurs pour toutes les valeurs de i possible. Comme ces valeurs sont équiprobables par hypothèse, la probabilité de chacune est $\frac{1}{n}$. Et nous obtenons :

$$C_n = \sum_{i=0}^{n} \frac{1}{n} \left(C_i + C_{n-i-1} + n - 1\right) = \frac{1}{n} \sum_{i=0}^{n} \left(C_i + C_{n-i-1}\right) + n - 1.$$

   Remarque : Il s'agit dans le fond d'une sorte de formule des probabilités totales pour l'espérance.

   (b) Changement d'indice $i \mapsto n-1-i$ dans la deuxième partie de la somme.
4. Soit $n \in [\![2, \infty[\![$. On commence par mettre de côté C_{n-1} dans la somme obtenue question précédente :

$$C_n = \frac{2}{n} \left(C_{n-1} + \sum_{i=0}^{n-2} C_i \right) + n - 1.$$

   Mais en appliquant la formule de la question précédente au rang $n-1$, il vient $C_{n-1} = \frac{2}{n-1} \sum_{i=0}^{n-2} C_i + n - 2$, d'où $\sum_{i=0}^{n-2} C_i = \frac{n-1}{2} \left(C_{n-1} - (n-2)\right)$. On remplace alors :

$$\begin{aligned} C_n =& \frac{2}{n} \left(C_{n-1} + \frac{n-1}{2} \left(C_{n-1} - (n-2)\right) \right) + n - 1 \\ =& C_{n-1} \times \frac{n+1}{n} + \frac{-(n-1)(n-2) + n(n-1)}{n} \\ =& C_{n-1} \times \frac{n+1}{n} + 2\frac{(n-1)}{n}. \end{aligned}$$

5. Pour tout $n \in [\![2, \infty[\![$,
$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + 2\frac{n-1}{n(n+1)}$$
d'où le résultat, sachant que $C_1 = 0$.

6. On a $2\dfrac{k-1}{k(k+1)} \underset{k\to\infty}{\sim} \dfrac{2}{k}$. Or la série de terme général $\left(\dfrac{2}{k}\right)_{k\in\mathbb{N}^*}$ est *à termes positifs* et *divergente* donc le théorème sur l'équivalent d'une série s'applique, et :
$$\frac{C_n}{n+1} \underset{x\to\infty}{\sim} \sum_{k=1}^{n} \frac{2}{k}.$$
On sait que la série harmonique $\sum_k \dfrac{1}{k}$ est équivalente à $\ln(n)$,d'où :
$$\frac{C_n}{n+1} \underset{n\to\infty}{\sim} 2\ln(n)$$
$$\text{donc } C_n \underset{n\to\infty}{\sim} 2(n+1)\ln(n)$$
$$\text{donc } C_n \underset{n\to\infty}{\sim} 2n\ln(n).$$
Ainsi, la complexité moyenne est quasi-linéaire.

19 N.B. Pour faire les calculs exacts, il faudrait mettre des parties entières partout...

1.
2.
3. Dans le cas où la longueur de T est multiple de 5 : notons n cette longueur, et notons M la médiane des médianes calculée. Il y a $n/5$ sous-tableaux. Parmi eux, $n/10$ ont une médiane inférieure à M. Et dans chacun de ces $n/10$ sous-tableaux, il y a trois éléments inférieurs à M (la médiane elle-même, et deux éléments plus petits).
Nous obtenons un total de $\dfrac{3n}{10}$ éléments inférieurs à M.
De même, il y a aussi $\dfrac{3n}{10}$ éléments supérieurs à M.
(On ne sait pas comment sont les $4n/10$ éléments restant.)
4.
5. Soit $n \in \mathbb{N}^*$. Appliquons la fonction à un tableau T de n éléments.
   - Pour commencer, on calcule chacune des médianes de sous-tableau de 5 éléments : ceci fait $\dfrac{n}{5}C_5$ comparaisons.
   - Déjà pour calculer la médiane de toutes ces médianes, il faut $C(n/5)$ comparaisons.
   - Ensuite, on segmente, en n comparaisons (la fonction segmente a été étudiée en cours, elle n'a pas changé).
   - On obtient alors deux sous-tableaux. Vues les questions précédentes on sait que chaque sous-tableau fait au moins $3n/10$ éléments et au plus $7n/10$.
     Le pire des cas est celui on devra poursuivre la recherche de la médiane dans le plus grand des deux sous-tableau, qui est au plus de longueur $7n/10$.
     Ainsi, la suite de la recherche de la médiane aura une complexité inférieure à $C(7n/10)$ (on suppose C croissante).
   - *Remarque :* Il y a encore une comparaison pour choisir le sous-tableau dans lequel poursuivre la recherche. On peut la négliger au sens où ce n'est pas la comparaison de deux éléments du tableau mais entre deux entiers directement disponibles (et petits). (ou alors augmenter c de 1 pour être rigoureux).

   D'où le résultat, pour $c = C_5/5 + 1$.
6. Récurrence forte : posons pour tout $n \in \mathbb{N}^*$, $\mathcal{P}(n) : C(n) \leqslant 10n$.
   - Pour un tableau de longueur 1, il n'y aucune comparaison entre éléments du tableau (une comparaison pour se rendre compte que la longueur est 1). D'où $\mathcal{P}(1)$.
   - Soit $n \in \mathbb{N}$. Supposons $\forall k \in [\![1, n]\!]$, $\mathcal{P}(k)$. Alors :
$$\begin{aligned} C(n+1) &\leqslant C\left(\frac{n+1}{5}\right) + C\left(7\frac{n+1}{10}\right) + c(n+1) \\ &\leqslant \frac{n+1}{5}10c + 7\frac{n+1}{10}10c + (n+1)c \\ &\leqslant 2c(n+1) + 7c(n+1) + c(n+1) \\ &\leqslant 10c(n+1) \end{aligned}$$
(par $\mathcal{P}(\left\lfloor \frac{n+1}{5} \right\rfloor)$ et $\mathcal{P}(\left\lfloor \frac{n+1}{5} \right\rfloor)$)

     D'où $\mathcal{P}(n+1)$.

Par récurrence forte, on a bien $\forall n \in \mathbb{N}^*$, $C(n) \leqslant 10cn$.

Ainsi, le calcul de la médiane se fait en temps linéaire dans tous les cas.

7.

8. La création du tableau des médianes à chaque étape nécessite un espace mémoire supplémentaire de $n/5$. On ne peut donc pas vraiment considérer qu'on a une méthode en place...