Corrigé du TD 9

Exercice 1. *En python, une liste n'est pas une liste chaînée mais ressemble plus à un tableau qui est géré dynamiquement (de la mémoire est allouée lorsque l'on ajoute des éléments). On notera n la longueur d'une liste* `L`*. Nous aurons parfois besoin d'un second paramètre $k \leq n$ pour désigner la longueur d'une seconde liste* `L2`*. D'après vous, quel est le coût des opérations suivantes ?*

Les listes sont implémentées en python comme des tableaux de taille variable. En particulier on peut accéder à un élément donné par son indice directement, sans parcourir la liste. La longueur de la liste est également accessible directement.

créer une liste vide	`L = []`	$\Theta(1)$
trouver la longueur	`l = len(L)`	$\Theta(1)$
lire un élément	`x = L[i]`	$\Theta(1)$
lire une « tranche »	`L2 = L[i:i+k]`	$\Theta(k)$
modifier un élément	`L[i] = 0`	$\Theta(1)$
ajouter un élément à la fin	`L.append(x)`	$\Theta(1)$
ajouter plusieurs éléments à la fin	`L.extend(L2)`	$\Theta(k)$
ajouter un élément « au milieu »	`L.insert(i,x)`	$\Theta(n)$
enlever le dernier élément	`L.pop()`	$\Theta(1)$
enlever un élément « au milieu »	`L.pop(i)`	$\Theta(n)$
trouver la valeur minimale	`x = min(L)`	$\Theta(n)$
tester si une valeur apparaît	`x in L`	$\Theta(n)$
enlever la première occurence d'une valeur	`L.remove(x)`	$\Theta(n)$
copier la liste	`L.copy()`	$\Theta(n)$

Les opérations de modification de la liste « au milieu » sont coûteuses car il faut décaller tous les éléments suivants. Lorsqu'on ajoute un élément en fin de liste et que l'espace alloué est atteint, python doit allouer un nouvel espace plus grand et recopier toute la liste dans le nouvel espace, ce qui a une complexité $\Theta(n)$. Cependant, la politique d'allocation d'espace utilisée fait que cet événement se produit « rarement », en moyenne l'assertion en fin de liste reste en $\Theta(1)$.

En utilisant les données ci-dessus, déterminer la complexité des fonctions ci-dessous :

```
def sommeIt(T):
    s = 0
    for i in range(len(T)):
        s = s + T[i]
    return s

def sommeRec1(T):
    if len(T) == 0: return 0
    x = T.pop() # dernier élément
    return x + sommeRec1(T)

def sommeRec2(T):
    if len(T) == 0: return 0
    x = T.pop(0) # premier élément
    return x + sommeRec2(T)
```

```
def double1(T):
    n = len(T)
    i = 0
    while i < 2*n:
        T.insert(i,T[i])
        i += 2

def double2(T):
    S = []
    for i in range(len(T)):
        S.append(T[i])
        S.append(T[i])
    return S
```

Voici les complexités en fonction de la taille n de `T` :

- `sommeIt(T)` : $\Theta(n)$ car la boucle est exécutée n fois et chaque exécution de la boucle a la complexité $\Theta(1)$
- `sommeRec1(T)` : $\Theta(n)$ car à chaque appel récursif, la taille du tableau diminue de 1, il y a donc n (ou $n+1$) appels récursifs, et la complexité de chaque appel est en $\Theta(1)$ (car c'est le cas de `len(T)` et `T.pop()`)
- `sommeRec2(T)` : $\Theta(n^2)$ car, de même il y a n appels récursifs, mais cette fois la complexité de chaque appel est $\Theta(n)$ à cause du `T.pop(0)`.
  Plus précisément, la complexité du premier appel est de l'ordre de n, celle du deuxième appel de l'ordre de $n-1$, ..., jusqu'au dernier appel dont la complexité est de l'ordre de 1 car la tableau n'a alors plus qu'un élément. Mais on a $1+2+3+\cdots+n = n(n+1)/2$ qui bien est dans la classe $\Theta(n^2)$.

— `double1(T)` : $\Theta(n^2)$ car la boucle est exécutée n fois (i augmente de 2 en 2) et il y a des insertions « au milieu » qui ont une complexité $\Theta(n)$.
— `double2(T)` : $\Theta(n)$, à nouveau la boucle est exécutée n fois, mais son exécution n'utilise que des opérations en $\Theta(1)$ (ajout en fin de tableau).

Remarque : `double2(T)` a une complexité meilleure que `double1(T)`, par contre elle utilise plus d'espace en mémoire car on crée un nouveau tableau. On a parfois besoin de faire des compromis entre la complexité en temps (celle à laquelle on s'intéresse dans l'exercice) et la complexité en espace.

Exercice 2.

Question 1. *Rappeler comment est calculé le coût d'un algorithme de tri.*

On compte le nombre de comparaisons entre éléments du tableau. C'est une simplification qui est criticable : même si le coût de la comparaison est supérieur au coût de lecture et d'écriture, si un algorithme impose par exemple de décaler de grandes parties d'un tableau cela ne sera pas pris en compte. Dans la question 3 on considère par exemple que l'insertion en « milieu » de tableau a un coût constant, ce qui ne correspond pas à ce qui a été dit à l'exercice précédent...

Question 2. *Redonner l'algorithme du tri sélection et calculer son coût exact. On pourra utiliser une procédure* `taille(T)` *qui renvoie la taille d'un tableau avec coût* 1*, et une procédure* `indexMin(T,a,b)` *qui renvoie l'indice d'un élément minimal du tableau entre les indices* `a` *et* `b` *inclus, avec coût* $b - a$*. Y a-t-il une différence entre la complexité dans le pire des cas et la complexité en moyenne ? Donner la classe* Θ *de cet algorithme.*

Voici l'algorithme vu au TP 5 :

```
triSelection(T : tableau d'entiers, debut : entier, fin : entier)
    j, n : entier
    n = taille(T)
    pour i de 0 à n - 1 faire
        j = indexMin(T, i, n - 1)
        T[i],T[j] = T[j],T[i]
```

La boucle est exécutée n fois, et les coûts d'exécutions de la boucle sont successivement proportionnels à $n-1$, $n-2$, $n-3$, ..., jusqu'à 1 (car $b - a = n - 1 - i$ diminue lorsque i augmente). Comme à l'exercice précédent, quand on additionne ces coûts on aboutit au coût exact $n(n-1)/2$ et à la classe de complexité $\Theta(n^2)$. On voit que le calcul du coût exact ne dépend pas des valeurs qui figurent dans le tableau passé en entrée : le coût est le même dans tous les cas.

Question 3. *Redonner l'algorithme du tri insertion. On pourra utiliser une procédre* `insère(T,i,x)` *qui insère une valeur* `x` *dans un tableau* `T` *à la position* `i` *avec coût* 1*. Calculer le coût de l'algorithme sur un tableau déjà trié en ordre croissant ? et sur un tableau trié en ordre décroissant ? Quel est le coût dans le pire des cas ?*

Voici l'algorithme, vu au TP 1 mais écrit ici de manière plus concise :

```
triInsertion(T : tableau d'entiers) : tableau d'entiers
    S : tableau d'entiers ; S = []
    j : entier
    # on parcourt le tableau en entrée
    pour i de 0 à taille(T) - 1 faire
        # on parcourt le nouveau tableau
        # jusqu'à trouver l'endroit où insérer T[i]
        j = 0
        tant que j < i et S[j] < T[i] faire
            j = j + 1
        insère(S, j, T[i])
    retourner S
```

Les comparaisons faisant intervenir les éléments de `T` on lieu uniquement dans le test d'arrêt de la boucle `tant que`. Dans le pire des cas, cette boucle s'exécute jusqu'à `j = i-1` et il y a donc $i-1$ comparaisons pour une valeur de i fixée. Ensuite dans la boucle principale i varie de 0 à $n-1$ donc, comme à la question précédente, on a $n(n-1)/2$ comparaisons dans le pire des cas. Le pire des cas est réalisé quand l'élément encours d'insertion est systématiquement plus grand que les éléments déjà insérés, c'est le cas si le tableau `T` est déjà classé en ordre croissant. Inversement le meilleur des cas est réalisé si `T` est en ordre décroissant : lors des insertions de `T[i]` est systématiquement plus petit

que les éléments déjà insérés donc la boucle `tant que` s'arrête tout de suite et il y a une seule comparaison pour chaque valeur de `i`.

C'est donc un cas où le coût exact dépend du tableau donnée en entrée, cela est dû au fait qu'il y a une condition d'arrêt qui dépend des valeurs qui figurent dans le tableau. Dans la question suivante on détaille le calcul de la complexité en moyenne. Le corrigé sera envoyé plus tard.

Question 4. *On veut maintenant calculer la complexité en moyenne de l'algorithme de tri insertion. On considère le tableau aléatoire* $\mathtt{T} = [\sigma(1), \ldots, \sigma(n)]$ *où* σ *est une permutation prise au hasard suivant la distribution uniforme.*

1. *Lorsqu'on réalise l'insertion de l'élément* $\sigma(k)$ *dans le début du nouveal tableau trié, quel est le nombre* c_k *de comparaisons effectuées ? On pourra donner une réponse à* ± 1 *près.*
2. *Montrer que la complexité* C *de l'algorithme, exécuté sur le tableau* `T`*, est égale à* $\sum_{1 \leq l < k \leq n} \delta_{\sigma(l) < \sigma(k)}$*, à* n *unités près. On note ici* $\delta_{a<b} = 1$ *si* $a < b$*,* 0 *sinon.*
3. *Montrer que la probabilité* $P(\sigma(l) < \sigma(k))$*, pour* $k \neq l$ *fixés, est égale à* $\frac{1}{2}$*.*
4. *En déduire un équivalent de la complexité en moyenne lorsque* n *est grand.*

Question 5. *Redonner l'algorithme du tri fusion. Montrer que l'on a*

$$CF(n) = F(n) + CF\left(\left\lfloor \tfrac{n}{2} \right\rfloor\right) + CF\left(\left\lceil \tfrac{n}{2} \right\rceil\right),$$

où $CF(n)$ *désigne le coût du tri fusion pour* n *entiers et* $F(n)$ *désigne le coût de la fusion de deux tableaux triés de taille* n_1 *et* n_2 *tels que* $n_1 + n_2 = n$*. Montrer que l'on a* $F(n) \leq n - 1$*. Nous supposerons pour les calculs que nous avons toujours* $F(n) = n - 1$*.*

Question 6. *Nous allons maintenant calculer* $CF(n)$ *dans le cas où* $n = 2^k$*. Calculer* $\lfloor n/2 \rfloor$ *et* $\lceil n/2 \rceil$ *en fonction de* k*. Que peut-on dire de la taille des sous-tableaux intervenant dans l'algorithme de tri fusion ? Pour tout* $i \in \{0, \ldots, k\}$*, posons* $c_i = CF(2i)$ *et* $f_i = F(2i)$*. Montrer par récurrence que l'on a* $c_i = (i-1)2^i + 1$*. En déduire* $CF(n)$ *pour* $n = 2^k$*. Quelle est la classe* Θ *du tri fusion ?*