université de BORDEAUX	Année Universitaire 2013/2014 DST de Printemps **Parcours :** Licence LIMI201 & LIMI211 **Code UE :** J1MI2013 **Épreuve :** Algorithmes et Programmes **Date :** Lundi 16 juin 2014 **Heure :** 16 heures 30 **Durée :** 2 heures Documents : non autorisés Épreuve de M. Alain Griffault	Collège Sciences et Technologies

SUJET + CORRIGE

Avertissement

- **À chaque question, vous pouvez au choix répondre par un algorithme ou bien par un programme python.**
- Les indentations des fonctions écrites en `Python` doivent être respectées.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Question	Points	Score
Le tri par tas	28	
Suite de Padovan	12	
Total:	40	

Exercice 1 : Le tri par tas (28 points)

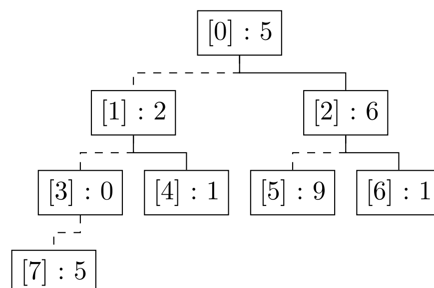
Le but de l'exercice est l'écriture d'un algorithme de tri de tableaux basé sur la notion de *tas*.

Les questions se suivent logiquement, mais beaucoup sont indépendantes.

(a) Des fonctions élémentaires pour se déplacer dans un tableau.

indice	0	1	2	3	4	5	6	7
T[indice]	5	2	6	0	1	9	1	5

(a) Vision tabulaire



(b) Vision arborescente

Figure 1 – Deux vues différentes d'un même tableau

La figure 1 montre qu'un même tableau peut être *dessiné* avec des cases *contiguës*, ou bien avec des cases *dispersées* dans une *arborescence*. Avec la vue *contiguë*, on utilise généralement une variable `i` qui *parcourt* les indices du tableau. Avec la vue *arborescente*, on peut évidemment utiliser une variable `i` qui *parcourt* les indices du tableau, mais on utilise également trois fonctions qui permettent de suivre les *liens bidirectionnels (réels ou virtuels) de l'arborescence* :

- `gauche(indice)` représente les liens *pointillé du haut vers le bas* de l'arborescence.
  Par exemple, dans la figure 1b, `gauche(1)=3`, `gauche(4)=9` et `gauche(2)=5`.
- `droite(indice)` représente les liens *en trait plein du haut vers le bas* de l'arborescence.
  Par exemple, dans la figure 1b, `droite(1)=4`, `droite(3)=8` et `droite(0)=2`.
- `pere(indice)` représente les liens *du bas vers le haut* de l'arborescence.
  Par exemple, dans la figure 1b, `pere(4)=1`, `pere(7)=3` et `pere(2)=0`. Par contre `pere(0)` n'est pas défini, et sa valeur (`null`,`-1`,`0`,...) importe peu car jamais utilisée dans cet exercice.

Voici un algorithme et un programme `Python`, de complexités en temps et en espace de $\Theta(1)$, possibles pour la fonction `gauche(indice)`.

```
Gauche(i){
  retourner 2*i+1;
}
```

```python
def gauche(i):
    return 2*i+1
```

i. (1 point) Écrire un programme `Python` ou un algorithme `droite(i)` qui retourne un entier `d` tel qu'il existe un lien *en trait plein du haut vers le bas* reliant les indices `i` à `d`.
Les complexités en temps et en espace doivent être en $\Theta(1)$.

Solution:

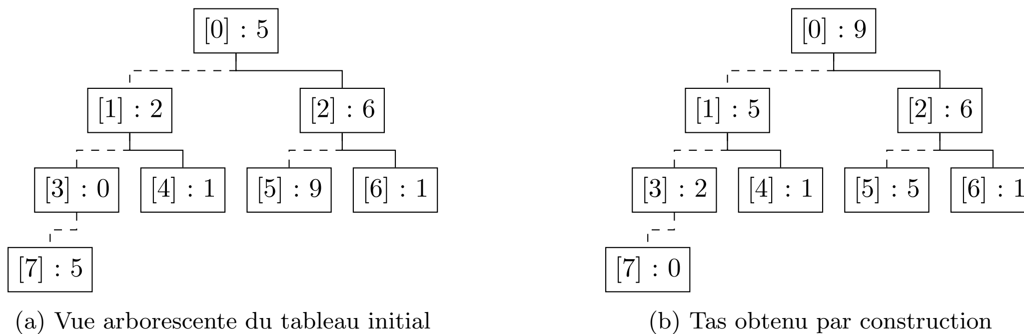
```python
def droite(i):
    return 2*(i+1)
```

ii. (1 point) Écrire un programme `Python` ou un algorithme `pere(i)` qui retourne un entier `p` tel qu'il existe un lien *du bas vers le haut* reliant les indices `i` à `p`.
Les complexités en temps et en espace doivent être en $\Theta(1)$.

Solution:

```python
def pere(i):
    return (i-1)//2
```

(b) Construction d'un tas à partir d'un tableau.
Définition 1 *Un tas est un tableau d'entiers tel que pour tous les indices* `i` *strictement positifs, la valeur de* `T[i]` *est inférieure ou égale à celle de* `T[pere(i)]`
Le but de cette partie de l'exercice est d'effectuer la transformation représentée par la figure 2.



(a) Vue arborescente du tableau initial

(b) Tas obtenu par construction

FIGURE 2 – Construction d'un tas

i. (3 points) Écrire un programme `Python` ou un algorithme `estunTas(T)` qui retourne `Vrai` si le tableau `T` est un tas, `Faux` sinon.
La complexité en temps doit être en $\Omega(1), \mathcal{O}(n)$ avec $n = longueur(T)$, et celle en espace doit être en $\Theta(1)$.

Solution:

```python
def estUnTas(T):
    for i in range(1,len(T)):
        if T[pere(i)]<T[i]:
            return False
    return True
```

ii. (3 points) Avec les hypothèses `0≤i<limite` et `limite≤longueur(T)`, écrire un programme `Python` ou un algorithme `maximum(T,i,limite)` qui retourne :

$$\text{le plus petit entier } \texttt{iMax} \text{ tel que } \begin{cases} (\texttt{iMax} < \texttt{limite}) \wedge (\texttt{iMax} \in \{\texttt{i}, \texttt{gauche(i)}, \texttt{droite(i)}\}) \\ \texttt{T[iMax]} \geq \texttt{T[i]} \\ \texttt{gauche(i)} < \texttt{limite} \Rightarrow \texttt{T[iMax]} \geq \texttt{T[gauche(i)]} \\ \texttt{droite(i)} < \texttt{limite} \Rightarrow \texttt{T[iMax]} \geq \texttt{T[droite(i)]} \end{cases}$$

En d'autres termes, `maximum(T,i,limite)` retourne l'indice (inférieur à `limite`) de la plus grande des trois valeurs `T[i]`, `T[gauche(i)]` et `T[droite(i)]`. En cas de valeurs égales, le plus petit indice est retourné. Par exemple sur la figure 2a, `maximum(T,0,8)=2`, `maximum(T,2,8)=5`, `maximum(T,3,8)=7` et `maximum(T,3,7)=3`.

Les complexités en temps et en espace doivent être en $\Theta(1)$.

Solution:

```
def maximum(T,i,limite):
    assert(0<=i and i<limite and limite<=len(T))
    iMax = i
    g = gauche(i)
    d = droite(i)
    # maximum entre T[i], T[g] et T[d] avec g et d < limite
    if g<limite and T[g]>T[iMax]:
        iMax = g
    if d<limite and T[d]>T[iMax]:
        iMax = d
    return iMax
```

iii. (1 point) Soit la fonction récursive `Python` suivante

```
def entasserRecursif(T,i,limite):
    iMax = maximum(T,i,limite)
    if iMax!=i:
        echange(T,i,iMax)
        entasserRecursif(T,iMax,limite)
```

avec

```
def echange(T,i,j):
    aux = T[i]
    T[i] = T[j]
    T[j] = aux
```

Compléter l'arborescence avec les valeurs du tableau après l'appel `entasserRecursif(T,0,8)`

Solution :

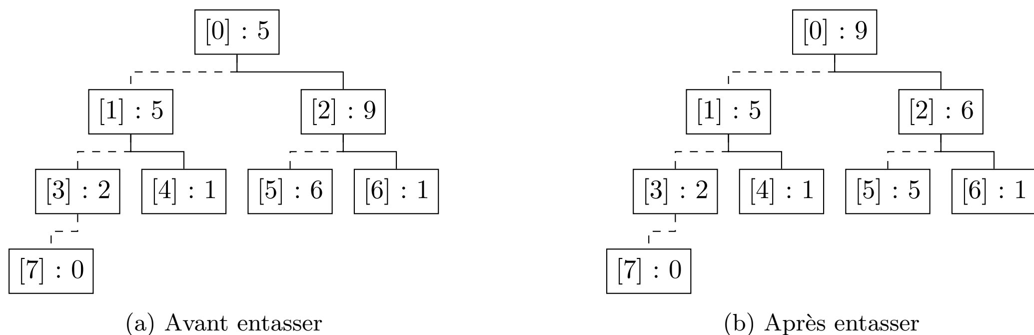


Figure 3 – Entasser(T,0,8)

iv. (3 points) La fonction `entasserRecursif(T,i,limite)` est récursive terminale.
Écrire un programme `Python` ou un algorithme itératif `entasser(T,i,limite)` équivalent.

Solution:

```
def entasser(T,i,limite):
    iMax = maximum(T,i,limite)
    while iMax!=i:
        echange(T,i,iMax)
        i = iMax
        iMax = maximum(T,i,limite)
```

v. (1 point) Donner les complexités en temps et en espace (meilleur des cas et pire des cas) de votre algorithme `entasser(T,i,limite)`.

Solution:
Les complexités sont :
– en temps en $\Omega(1), \mathcal{O}(log_2(n))$ avec $n = longueur(T)$,
– en espace en $\Theta(1)$.

vi. (4 points) L'algorithme `entasser(T,i,limite)` échange des valeurs du tableau de haut en bas, *en suivant une branche de l'arborescence*. Cela a pour effet de faire *descendre* des petites valeurs, et de faire *monter* les grandes valeurs. Il est donc possible de construire un tas, en itérant cet algorithme sur les indices décroissants du tableau.

En utilisant `entasserRecursif(T,i,limite)` ou `entasser(T,i,limite)`, écrire un programme `Python` ou un algorithme `construireTas(T)` qui transforme un tableau en un tas.

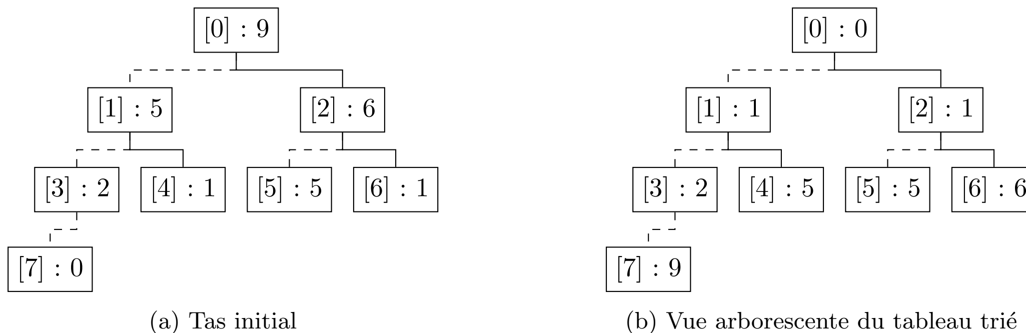
Solution:

```
def construireTas(T):
    #for i in range(len(T)-1,-1,-1): peut etre optimise en
    for i in range((len(T)-1)//2,-1,-1):
        entasser(T,i,len(T))
```

vii. (1 point) Donner les complexités en temps et en espace (meilleur des cas et pire des cas) de votre algorithme `construireTas(T)`.

Solution:
Si l'on utilise l'algorithme itératif pour *entasser*, les complexités sont :
– en temps en $\Omega(n)$ si `T` est déjà un tas, $\mathcal{O}(n \times log_2(n))$ avec $n = longueur(T)$,
– en espace en $\Theta(1)$.

(c) Tri d'un tas.
Le but de cette partie de l'exercice est d'effectuer la transformation représentée par la figure 4.



(a) Tas initial (b) Vue arborescente du tableau trié

FIGURE 4 – Tri d'un tas

i. (4 points) Dans un tas, la valeur maximale est à la *racine de l'arborescence*, donc en `T[0]`. Dans le tableau trié, cette valeur doit être en `T[len(T)-1]`. Il suffit donc d'échanger ces deux valeurs pour progresser vers la solution. Une fois cette échange fait, si l'on exclut la dernière valeur du tableau, le tas est peu changé. En fait `entasser(T,0,len(T)-1)` va créer un nouveau tas pour les valeurs du tableau dont les indices sont inférieurs à `limite = len(T)-1`. Il suffit donc d'itérer ces deux étapes (échange, entasser) pour trier un tas.

Écrire un programme `Python` ou un algorithme `trierTas(T)` qui transforme un tas en un tableau trié en ordre croissant.

Solution:

```
def trierTas(T):
    for i in range(len(T)-1,0,-1):
        echange(T,0,i)
        entasser(T,0,i)
```

ii. (1 point) Donner les complexités en temps et en espace (meilleur des cas et pire des cas) de votre algorithme `trierTas(T)`.

Solution:
Si l'on utilise l'algorithme itératif pour *entasser*, les complexités sont :
– en temps en $\Omega(n)$ si toutes les valeurs de `T` sont égales, $\mathcal{O}(n \times log_2(n))$ avec $n = longueur(T)$,
– en espace en $\Theta(1)$.

(d) Tri d'un tableau à l'aide de tas.

i. (3 points) Écrire un programme `Python` ou un algorithme `triParTas(T)` qui trie un tableau d'entiers `T` en construisant d'abord un tas, puis en le triant.

Solution:

```
def triParTas(T):
    construireTas(T)
    trierTas(T)
```

ii. (1 point) Donner les complexités en temps et en espace (meilleur des cas et pire des cas) de votre algorithme `triParTas(T)`.

Solution:
Si l'on utilise l'algorithme itératif pour *entasser*, les complexités sont :
– en temps en $\Omega(n)$ si toutes les valeurs de `T` sont égales, $\mathcal{O}(n \times log_2(n))$ avec $n = longueur(T)$,
– en espace en $\Theta(1)$.

iii. (1 point) Votre algorithme `triParTas(T)` effectue t-il un tri stable ? Justifier.

Solution:
Non, car il effectue des échanges de valeurs avec des indices non consécutifs.

Exercice 2 : Suite de Padovan (12 points)

La suite (ou fonction) de Padovan d'un entier naturel est définie par :

$$Padovan(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ 1 & \text{si } n = 2 \\ Padovan(n-2) + Padovan(n-3) & \text{sinon} \end{cases}$$

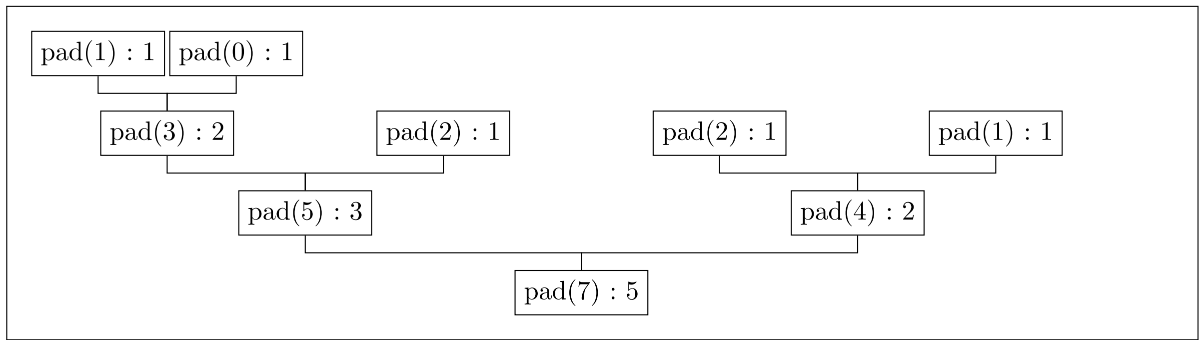
(a) i. (3 points) Écrire un programme `Python` ou un algorithme récursif `padovanRecursif(n)` qui retourne la valeur $Padovan(n)$.

Solution:

```
def padovanRecursif(n):
    assert(n>=0)
    if n<3:
        return 1
    else:
        return padovanRecursif(n-2) + padovanRecursif(n-3)
```

ii. (1 point) Dessiner l'arbre des appels récursifs lors de l'appel `padovanRecursif(7)`.

Solution:



iii. (1 point) En interprétant l'arbre des appels récursifs et notamment la longueur de ses branches, donner un encadrement du nombre d'appels récursifs, pour en déduire la complexité (temps et espace) de votre algorithme récursif.

Solution:
- La plus longue branche (la plus à gauche) de l'arbre des appels a une longueur de $n/2$. Le nombre d'appels est donc inférieur à $2^{n/2}$
- La plus courte branche (la plus à droite) de l'arbre des appels a une longueur de $n/3$. Le nombre d'appels est donc supérieur à $2^{n/3}$

La complexité :
- en temps est comprise entre $\Omega(\sqrt[3]{2}^n)$ et $\mathcal{O}(\sqrt{2}^n)$.
- en espace est de $\Theta(n)$.

non demandé : La complexité en temps est en $\Theta(\psi^n)$ avec ψ dénommé *le nombre plastique*, qui est l'unique solution réelle de $X^3 = X + 1$, et environ égal à $1,3247179572$.

(b) (4 points) Par analogie avec la fonction `Fibonacci(n)` vue en cours, il est possible de transformer cette fonction récursive en une fonction récursive terminale. Pour cela, il faut ajouter trois paramètres qui contiennent trois valeurs consécutives de la suite. A chaque appel récursif :
- deux sont modifiés par *décalage*,
- le troisième est obtenu par l'addition de deux parmi les trois.

Écrire un programme `Python` ou un algorithme récursif terminal `padovanRecTerminal(n,u,v,w)` qui retourne la valeur $Padovan(n)$, et préciser l'appel initial.

Solution:

```
def padovanRecTerminal(n, u=1, v=1, w=1):
    assert(n>=0)
    if n==0:
        return u
    elif n==1:
        return v
    elif n==2:
        return w
    else:
        return padovanRecTerminal(n-1,v,w,v+u)
```

(c) i. (2 points) Transformer *automatiquement* votre solution récursive terminale, afin d'obtenir un programme `Python` ou un algorithme itératif `padovanIteratif(n)` qui retourne la valeur $Padovan(n)$.

Solution:

```
def padovanIteratifAutomatique(n):
    assert(n>=0)
    u=1
    v=1
    w=1
    while True:
        if n==0:
            return u
        elif n==1:
            return v
```

```
    elif n==2:
        return w
    else:
        # n,u,v,w = n-1,v,w,v+u
        n = n-1
        aux = u
        u = v
        v = w
        w = u+aux
```

ii. (1 point) Donner les complexités en temps et en espace (meilleur des cas et pire des cas) de votre solution itérative.

Solution:
La complexité :
– en temps est en $\Theta(n)$,
– en espace est en $\Theta(1)$.