+1/1/60+

EPFL

1

Enseignants: G. Maatouk, C.D. Petrescu
Informatique et Calcul Scientifique (ICS) - CMS
11 janvier 2023
Durée : 105 minutes

Dalton Joe

SCIPER : **987654**

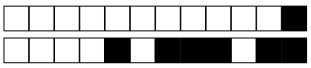
Attendez le début de l'épreuve avant de tourner la page. Ce document est imprimé recto-verso, il contient 10 questions sur 12 pages, les dernières pouvant être vides. Ne pas dégrafer.

- Posez votre **carte d'étudiant.e** sur la table.
- Le seul matériel autorisé est :
  - Un formulaire personnel consistant d'**une feuille A4 recto verso manuscrite**,
  - Le polycopié officiel de la première partie du cours. Ce document peut être annoté (avec des notes concernant la première partie du cours "Programmation Python") mais pas complété avec des feuilles supplémentaires ou d'autres éléments ajoutés d'une façon quelconque au support imprimé.
- L'utilisation d'une **calculatrice** et de tout **outil électronique** est **interdite** pendant l'épreuve.
- Pour les questions à **choix unique**, on comptera :
  les points indiqués si la réponse est correcte,
  0 point si il n'y a aucune ou plus d'une réponse inscrite,
  0 point si la réponse est incorrecte.
- Les algorithmes demandés sont à écrire sous forme de fonctions Python.
- Utilisez un **stylo** à encre **noire ou bleu foncé** et effacez proprement avec du **correcteur blanc** si nécessaire.
- Répondez dans l'espace prévu (**aucune** feuille supplémentaire ne sera fournie).
- Les brouillons ne sont pas à rendre: ils ne seront pas corrigés.

Respectez les consignes suivantes | Observe this guidelines | Beachten Sie bitte die unten stehenden Richtlinien

choisir une réponse | select an answer Antwort auswählen	ne PAS choisir une réponse | NOT select an answer NICHT Antwort auswählen	Corriger une réponse | Correct an answer Antwort korrigieren

ce qu'il ne faut **PAS** faire | what should **NOT** be done | was man **NICHT** tun sollte



Première partie, questions à choix unique

Pour chaque énoncé proposé, une ou plusieurs questions sont posées. Pour chaque question, marquer la case correspondante à la réponse correcte sans faire de ratures. Il n'y a qu'**une seule** réponse correcte par question.

Enoncé

On donne la fonction suivante:

```
def f(n):
    somme = 0
    i = n
    while i > 1:
        i //= 2
        for j in range(n):
            somme += 1
    return somme
```

Question 1 (2 points)
Laquelle des affirmations suivantes est vraie?

- ☐ $f(8) = f(9)$
- ☐ $f(4) = 4 \cdot f(2)$
- ☐ $f(4) = 2 \cdot f(2)$
- ☐ $f(8) = f(7)$

Question 2 (2 points)
Pour la valeur n en entrée, soit $T(n)$ le temps de parcours de la fonction f. Laquelles des affirmations suivantes est vraie?

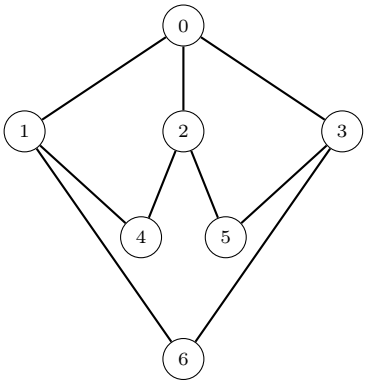
- ☐ $T(n) = \Theta(\log_2(n))$
- ☐ $T(n) = \Omega(n \log_2(n))$ mais $T(n)$ n'est pas $\Theta(n \log_2(n))$
- ☐ $T(n) = \Theta(n + \log_2(n))$
- ☐ $T(n) = \Theta(n \log_2(n))$

Enoncé

On reproduit ci-dessous l'algorithme `BFS_arbre` vu en cours, et on donne un graphe G.

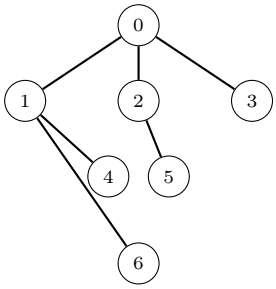
```
from collections import deque

def BFS_arbre(G,s):
    n = len(G)
    a_parcourir = deque([s])
    vu = [0 for u in range(n)]
    vu[s] = 1
    T = {u:[] for u in range(n)}
    while a_parcourir:
        sommet = a_parcourir.popleft()
        for u in G[sommet]:
            if not vu[u]:
                a_parcourir.append(u)
                vu[u] = 1
                T[sommet].append(u)
                T[u].append(sommet)
    return T
```

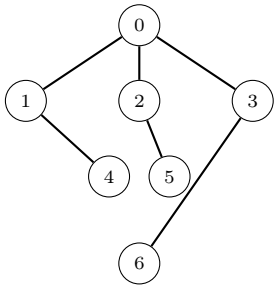


G

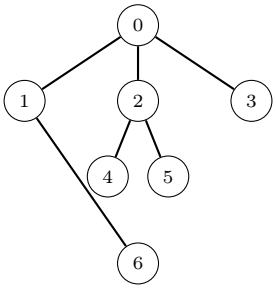
On donne quatre arbres couvrants de G: T_1, T_2, T_3 et T_4.



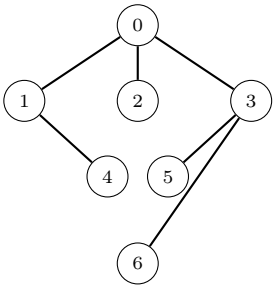
T_1



T_2



T_3



T_4

Question 3 (2 points)

Lequel des quatre arbres **ne peut pas** avoir été produit par `BFS_arbre(G, 0)`?

- ☐ T_2
- ☐ T_3
- ☐ T_4
- ☐ T_1

Question 4 (2 points)

On reproduit ci-dessous les algorithmes `tri_par_selection` et `tri_par_insertion`, où on a rajouté des instructions `print(L)` pour illuster le comportement des algorithmes.

```
def tri_par_selection(L):
    n = len(L)
    for i in range(n):
        m = L[i]
        m_index = i
        for j in range(i+1,n):
            if L[j] < m:
                m = L[j]
                m_index = j
        L[i], L[m_index] = L[m_index], L[i]
        print(L)
```

```
def tri_par_insertion(L):
    n = len(L)
    for i in range(n):
        j = i
        while j > 0 and L[j] < L[j-1]:
                L[j], L[j-1] = L[j-1], L[j]
                j -= 1
        print(L)
```

Pour quelle liste `L` les appels `tri_par_insertion(L)` et `tri_par_selection(L)` produisent-ils le même affichage?

- ☐ `L = [1, 5, 4, 3, 2, 6]`
- ☐ `L = [6, 5, 4, 3, 2, 1]`
- ☐ Aucune des listes proposées ne conduit au même affichage pour les deux appels
- ☐ `L = [2, 1, 4, 3, 6, 5]`

Question 5 (2 points)

On reproduit ci-dessous les algorithmes `tri_par_fusion` et `fusion` donnés en cours, où on a rajouté l'instruction `print(L)` après l'appel à la fonction `fusion` dans la fonction `tri_par_fusion`.

```
def tri_par_fusion(L, bas, haut):
    if haut - bas > 0:
        milieu = (bas + haut)//2
        tri_par_fusion(L, bas, milieu)
        tri_par_fusion(L, milieu+1, haut)
        fusion(L, bas, milieu, haut)
        print(L)
```

```
def fusion(L, bas, milieu, haut):
    L1 = L[bas:milieu+1]
    L2 = L[milieu+1:haut+1]
    L1.append(float('inf'))
    L2.append(float('inf'))
    L1_index = 0
    L2_index = 0
    for i in range(bas, haut+1):
        if L1[L1_index] <= L2[L2_index]:
            L[i] = L1[L1_index]
            L1_index += 1
        else:
            L[i] = L2[L2_index]
            L2_index += 1
```

Qu'affichent les deux instructions suivantes?

```
L = [6, 5, 4, 3, 2, 1]
tri_par_fusion(L, 0, len(L)-1)
```

- ☐
```
[6, 4, 5, 3, 1, 2]
[4, 5, 6, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

- ☐
```
[5, 6, 4, 2, 3, 1]
[4, 5, 6, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

- ☐
```
[6, 4, 5, 3, 2, 1]
[4, 5, 6, 3, 2, 1]
[4, 5, 6, 3, 1, 2]
[4, 5, 6, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

- ☐
```
[5, 6, 4, 3, 2, 1]
[4, 5, 6, 3, 2, 1]
[4, 5, 6, 2, 3, 1]
[4, 5, 6, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
```

Enoncé

On reproduit ci-dessous l'algorithme de recherche binaire vu en cours, où on a rajouté l'instruction `print(milieu, end=' ')` dans la boucle `while` pour illuster le comportement de l'algorithme.

```
def recherche_binaire(L, x):
    n = len(L)
    bas = 0
    haut = n-1

    while haut >= bas:
        milieu = (bas + haut)//2
        print(milieu, end=' ')
        if L[milieu] == x:
            return milieu
        elif L[milieu] > x:
            haut = milieu - 1
        else:
            bas = milieu + 1
```

On définit la liste `L = [10, 12, 14, 16, 18, 20, 22, 24]`.

Question 6 (2 points)
Quelle paire d'appels produit le même affichage?

- ☐ `recherche_binaire(L, 11)` et `recherche_binaire(L, 13)`
- ☐ `recherche_binaire(L, 13)` et `recherche_binaire(L, 15)`
- ☐ `recherche_binaire(L, 15)` et `recherche_binaire(L, 16)`
- ☐ `recherche_binaire(L, 12)` et `recherche_binaire(L, 13)`

Question 7 (1 point)
Quel affichage est produit par l'appel `recherche_binaire(L, 13)` ?

- ☐ `3 1`
- ☐ `3 1 2 1`
- ☐ `3 2 1`
- ☐ `3 1 2`

Deuxième partie, questions de type ouvert

Répondre dans l'espace dédié. Laisser libres les cases à cocher : elles sont réservées au correcteur.

Question 8: *Cette question est notée sur 5 points.*

.5 .5 .5 .5 .5

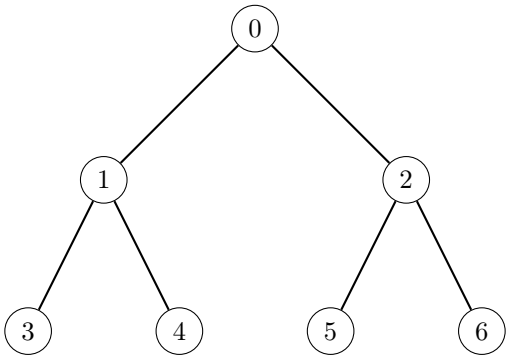
0 1 2 3 4 5

On donne ci-dessous deux variantes de l'algorithme DFS vu en cours, qui diffèrent uniquement par la position de l'instruction `print(s)`.

```
def DFS_pre(G, s):
    print(s)
    vu[s] = 1
    for u in G[s]:
        if not vu[u]:
            DFS_pre(G, u)
```

```
def DFS_post(G, s):
    vu[s] = 1
    for u in G[s]:
        if not vu[u]:
            DFS_post(G, u)
    print(s)
```

On donne le graphe G ci-dessous:



(a) Donnez une représentation par listes d'adjacence du graphe G, sous forme d'une ligne de code Python commençant par "`G = `" .

(b) Qu'affichent les instructions suivantes (où la première ligne doit être remplacée par la ligne de code que vous avez fournie au point (a))?

```
G = # representation par listes d'adjacence de G
vu = [0 for u in G]
DFS_pre(G, 0)
```

(c) Qu'affichent les instructions suivantes (où la première ligne doit être remplacée par la ligne de code que vous avez fournie au point (a))?

```
G = # representation par listes d'adjacence de G
vu = [0 for u in G]
DFS_post(G, 0)
```

Question 9: *Cette question est notée sur 6 points.*

Ecrivez un algorithme `list_intersection` qui prend en entrée deux listes **triées** de nombres `L1` et `L2`, et retourne une liste `L` contenant les éléments présents dans les deux listes `L1` et `L2`. La liste `L` peut contenir des éléments répétés.
Par exemple:

- `list_intersection([1, 2, 2], [2, 2, 3])` doit retourner `[2]` (ou `[2, 2]`...)
- `list_intersection([1, 2], [3, 4, 5])` doit retourner `[]` .

Soit n la somme des tailles des deux listes `L1` et `L2`: votre algorithme doit avoir un temps de parcours linéaire en n. **Justifiez brièvement ce temps de parcours**.

Votre code **ne peut pas** utiliser des objets de type `set`, ni de listes en compréhension.

Bonus (1 point): toujours sans utiliser d'objets de type `set`, faites en sorte que la liste `L` ne contienne aucune répétition. Par exemple, `list_intersection([1, 2, 2], [2, 2, 3])` doit retourner `[2]`.

Remarque: un algorithme correct mais moins efficace que ce qui est demandé obtiendra 2 points.

Question 10: *Cette question est notée sur 6 points.*

.5 .5 .5 .5 .5 .5

0 1 2 3 4 5 6

Ecrivez un algorithme **récursif** `rec_reverse` qui prend en entrée une chaîne de caractères `s` et retourne la chaîne de caractères `s` à l'envers, c'est-à-dire la chaîne de caractères constituée des mêmes caractères que `s` mais dans l'ordre inverse de leur apparition dans `s`.

Par exemple, l'appel `rec_reverse("abc")` doit retourner la chaîne de caractères `"cba"`, `rec_reverse("a")` doit retourner `"a"` et `rec_reverse("")` doit retourner la chaîne vide.

Votre algorithme ne doit contenir **aucune boucle**. Il ne doit pas utiliser de slicing de pas négatif (du genre `s[::-1]`), ni la fonction `reversed()` (qui n'a pas été vue en cours), ni la méthode de liste `reverse()`.