

Année universitaire : 2019 / 2020

DIU Enseigner l’Informatique au Lycée Devoir maison
UE 2 – Algorithmique Date : 30 avril 2020

Durée estimée : 2h

|  |
| --- |
| Si vous souhaitez composer sur une feuille papier, travaillez au brouillon d’abord de sorte à rendre une copie propre et assurez-vous de la lisibilité du scan ou des photos. Il sera tenu compte de la présentation et de la clarté de vos réponses. Ce devoir est individuel. Vous avez le droit à vos notes de cours, TD, TP ainsi qu’à un éditeur Python. |

|  |  |
| --- | --- |
| **NOM** :  | **PRENOM** :  |

**Exercice 1 : Complexité des algorithmes**

On considère la fonction suivante réalisant la fusion de deux listes triées passées en paramètres. La fonction retourne la liste fusionnée elle-même triée.

|  |
| --- |
|  **def** fusion(liste1,liste2) :1 i1,i2 = 0,02 resultat = []3 **while** i1 < len(liste1) **and** i2 < len(liste2) :4 **if** liste1[i1] < liste2[i2] :5 resultat.append(liste1[i1])6 i1 += 17 **else** :8 resultat.append(liste2[i2])9 i2 += 110 resultat += liste1[i1:]11 resultat += liste2[i2:]12 **return** resultat |

Recopiez cette fonction dans un éditeur Python.

**Question 1.1 :** Donnez le code du programme principal permettant de récupérer dans une variable f la liste fusionnée des deux listes liste1 = [-2, 0, 1, 5] et liste 2 = [-1, 0, 6, 7].

|  |
| --- |
|  |

**Question 1.2 :** Que vaut f ?

|  |
| --- |
|  |

**Question 1.3 :** A quoi servent les instructions des lignes 10 et 11 ?

|  |
| --- |
|  |

**Question 1.4 :** A partir d’une analyse rapide de l’algorithme, comptez le nombre d’ajouts effectués dans la liste **resultat** en fonction des tailles $n\_{1}$ et $n\_{2}$ des deux listes passées en paramètres. Justifiez brièvement votre réponse.

|  |
| --- |
|  |

Ajoutez dans le code une variable **compteur** qui est incrémentée à chaque ajout d’un élément dans **resultat** et, à la fin de la fonction, affichez la valeur de **compteur** ainsi que la taille des deux listes.

**Question 1.5 :** Quelles instructions avez-vous ajoutées et à quelles lignes ?

|  |
| --- |
|  |

Vérifiez votre réponse à la question 1.4 en faisant quelques appels à la fonction **fusion** avec des listes de différentes tailles.

**Question 1.6 :** En conclure sur la complexité temporelle dans le pire des cas, en notation $O$, de la fonction **fusion** en fonction des tailles des deux listes.

|  |
| --- |
|  |

**Question 1.7** : Donnez la fonction récursive Python de tri par fusion qui prend en paramètre la liste à trier et qui retourne la liste triée. Vous pourrez utiliser la fonction **fusion**.

|  |
| --- |
|  |

**Question 1.8** : Déterminez et justifiez, par la méthode du Master Theorem, la complexité de cette fonction de tri.

Pour rappel, dans cette méthode, $a$ est le nombre de sous-problèmes, $b$ est le facteur de réduction de la taille des sous-problèmes et $f(n)$ décrit le coût de la décomposition et recomposition. Si le coût des calculs liés à la récursivité domine celui des calculs liés à la décomposition et recomposition, alors la complexité est en $Θ(n^{log\_{b}a})$. Si c’est l’inverse, la complexité est en $Θ(f(n))$. En cas d’égalité des coûts, la complexité est en $Θ(n^{log\_{b}a}log\_{b}n)$.

|  |
| --- |
|  |

**Exercice 2 : Algorithme glouton – Empaquetage**

On dispose d'un ensemble E de $n$ objets où chaque objet possède un poids $p\_{i}$ ($i\in [1,n]$). On dispose également de boîtes de capacité $C$. On souhaite placer les n objets dans les boîtes en respectant les capacités (la somme des poids des objets dans une boîte ne doit pas dépasser sa capacité) et en utilisant le moins possible de boîtes.

La première idée, reposant sur le principe des algorithmes gloutons, est de placer les objets dans la première boîte où c’est possible.

Prenons l’exemple suivant d’un ensemble E de $n=8$ objets et des boîtes de même capacité $C=11$.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Poids | 7 | 6 | 3 | 4 | 8 | 5 | 9 | 2 |

Prenons les objets dans l’ordre donné (gauche à droite) et ajoutons les dans la première boîte où c’est possible. Nous obtenons les boîtes suivantes :

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
| 3 |  | 4 |  | 2 |  |  |  |  |
| 7 |  | 6 |  | 8 |  | 5 |  | 9 |

L’objet de poids 7 a été mis dans la première boîte, puis le 6 dans la deuxième (car 6+7=13>11 donc impossible de le mettre aussi dans la première boîte). Ensuite on a pu mettre l’objet de poids 3 dans la première boîte, mais pas le 4 (7+3+4>11) qui a été ajouté dans la deuxième boîte (6+4$\leq $11). Et ainsi de suite jusqu’à ajouter le dernier objet. Le nombre de boîtes nécessaires est donc ici 5 (pour cet exemple et cette stratégie), mais cette solution est-elle optimale ?

La fonction Python **empaqueter** suivante calcule et retourne le nombre de boîtes nécessaires pour empaqueter les objets selon la stratégie présentée. La fonction prend en paramètre une liste **objets** qui contient les objets disponibles sous la forme [poids1, poids2, ... poidsn] et la capacité des boîtes **C** (même capacité pour toutes les boîtes).

 **def** empaqueter(objets,C) :

1 nb\_boites = 0

2 capacite\_restante = [C]

3 **for** objet in objets :

4 indice\_boite = 0

5 **while** indice\_boite <= nb\_boites **and** objet > capacite\_restante[indice\_boite] :

6 indice\_boite += 1

7 **if** indice\_boite == nb\_boites + 1 :

8 nb\_boites += 1

9 capacite\_restante.append(C)

10 capacite\_restante[indice\_boite] -= objet

11 **return** nb\_boites+1

**Question 2.1** : Expliquez brièvement le rôle de la boucle des lignes 5 et 6, et du +1 de la ligne 11.

|  |
| --- |
|  |

**Question 2.2** : Quelles sont les préconditions de cette fonction (c’est-à-dire dans quelles conditions fonctionne-t-elle correctement) ?

|  |
| --- |
|  |

Comme vous devez vous en douter, cette stratégie n’est pas optimale. Une autre stratégie, toujours gloutonne, consiste à ajouter les objets dans l’ordre de leur poids (les plus lourds en premier).

Sur l’exemple précédent, nous obtenons la solution suivante :

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |
| 2 |  | 3 |  | 4 |  | 5 |
| 9 |  | 8 |  | 7 |  | 6 |

Le premier objet à être ajouté est le 9 (de poids max) qui se place dans la première boîte, puis le 8 dans la deuxième (impossible de le mettre aussi dans la première car 9+8>11), etc. Cette stratégie est ici meilleure, elle trouve une solution avec 4 boîtes.

**Question 2.3** : Que faut-il modifier au code de la fonction **empaqueter** afin qu’elle suive cette nouvelle stratégie ?

|  |
| --- |
|  |

**Question 2.4** : Cette stratégie est-elle optimale ? Si oui, justifiez brièvement pourquoi. Si non, donnez un contre-exemple.

|  |
| --- |
|  |

**Exercice 3 : Correction des algorithmes**

La fonction itérative **factoriel** suivante calcule et retourne la valeur de n ! (factorielle de n), n étant passé en paramètre. Pour rappel, $n!=1×2×\cdots ×n$.

|  |
| --- |
| **def** factoriel(n) : res = 1 **for** i **in** range(2,n+1) : res = res \* i **return** res |

**Question 3.1** : Quelles sont les préconditions de cette fonction ?

|  |
| --- |
|  |

**Question 3.2** : Démontrez la terminaison de la fonction.

|  |
| --- |
|  |

**Question 3.3** : Donnez un invariant pour la boucle pour et démontrez la correction totale de la fonction.

|  |
| --- |
|  |