Examen Terminal UE Programmation et Algorithmes

L1 MPCI

5 mai 2023 - Durée: 3h

Lorsque l'on vous demande d'écrire de décrire ou de donner un algorithme cela signifiera toujours en donner un pseudo-code, justifier de son exactitude et de sa complexité

On rappelle qu'aucun document, ni équipement électrique ou électronique n'est autorisé.

Les exercices :
— sont au nombre de 3 ;
— valent chacun *à peu près* le même nombre de points (de l'ordre de 6/8/8) ;
— sont de difficulté équivalente ;
— sont indépendants ;
— leur début est plus facile que leur fin.

Rendez des copies séparées pour chaque exercice, ceci vous permettra de reprendre les exercices au cours de l'examen sans perdre le correcteur.

Exercice 1 – Programmation objet : structure d'ensemble

On appellera *tableau* une collection d'éléments de taille fixe (comme les tuples), dont on peut modifier les éléments (comme les listes). On supposera qu'il existe une classe `Tableau` permettant de stocker de tels tableaux. On dispose aussi d'une fonction `crée_tableau(n)` qui prend un entier `n` et renvoie un tableau de taille `n`. On peut ensuite accéder en temps constant au $\texttt{i}^{\text{ième}}$ élément d'un tableau `t`, en lecture et en écriture, grâce à la notation `t[i]`. On accède à la taille du tableau `t` via `len(t)`.

On souhaite construire une structure de données pour manipuler des ensembles, c'est-à-dire des collections non-ordonnées et sans redondance, comme les ensembles en maths, et dont les éléments sont des entiers. On supposera qu'il n'existe pas de classe `set` en Python.

Nous proposons de créer une classe `Ensemble` contenant deux attributs : un tableau avec une certaine taille maximale pour stocker les éléments et le nombre d'éléments effectivement contenus dans l'ensemble (plus petit ou égal à la taille maximale).

Question 1.1 Écrivez une classe `Ensemble` contenant :

- un constructeur prenant en argument un entier `n` et qui crée un `Ensemble` vide de taille maximale `n` ;
- une méthode `contient(x)` qui renvoie `True` si l'entier `x` est un élément de l'ensemble et `False` sinon.
- une méthode `ajoute(x)` qui ajoute l'entier `x` dans l'ensemble s'il n'est pas déjà présent, en modifiant les attributs de façon appropriée, et ne renvoie rien.
- une méthode `retire(x)` qui retire l'entier `x` de l'ensemble s'il est présent, en modifiant les attributs de façon appropriée, ne fait rien sinon, et ne renvoie rien.
- une méthode `union(autre_ensemble)` qui renvoie un nouvel objet `Ensemble` résultat de l'union entre l'objet courant et de l'ensemble `autre_ensemble` donné en argument.
- une méthode `intersection(autre_ensemble)` qui a un comportement similaire pour obtenir l'intersection de l'objet courant et d'un autre ensemble donné en argument.

N'oubliez pas les complexités, à exprimer dans le pire cas, en fonction de la taille maximale des ensembles.

Question 1.2 Pour chacune des méthodes ci-dessus, y compris le constructeur, donnez le code de deux tests qui vous paraissent appropriés.

Question 1.3 Quel est le comportement de la méthode `ajoute(x)` lorsque l'ensemble est déjà plein (le nombre d'éléments est égal à la taille maximale) ? On ne demande pas de traiter ce cas.

Exercice 2 – Un problème d'optimisation

Une de vos cousines se marie et vous a demandé de faire le plan de table du repas de noces. Pour maximiser la convivialité du repas elle vous demande :

— de ne mettre à chaque table que des personnes qui s'entendent ;
— d'avoir un petit nombre de tables.

On ne demande pas que le nombre de tables soit minimum.

2.1 Modélisation

Un plan de table P est une structure de données contenant :

— une liste `NOMS` contenant le nom de tous les invités ;
— une liste `IC` d'incompatibilités où `IC[i]` contient un ensemble d'indices tel que si `j` est dans `IC[i]` alors `NOMS[i]` ne peut être à la même table que `NOMS[j]`.

On suppose de plus que la relation d'incompatibilité est symétrique (si `j` est dans `IC[i]` alors `i` est dans `IC[j]`). Par exemple si les invités sont : *"tata Guillemette"*, *"cousin Valentin"*, *"tonton Julien"*, *"papy François"* et *"soeur Manon"* et que les relations sont :

— *"tata Guillemette"* aime bien tout le monde
— *"papy François"* n'aime personne à part *"tata Guillemette"*
— *"cousin Valentin"* ne supporte pas *"soeur Manon"*

On a la structure de données suivante (où `{...}` représente des ensembles) :

1. `NOMS = ["tonton Julien", "papy François", "tata Guillemette", "cousin Valentin", "soeur Manon"]`
2. `IC = [{1}, {0, 3, 4}, {}, {1, 4}, {3, 1} ]`

Question 2.1.1 En supposant que vous avez en votre possession la structure d'**Ensemble** de l'exercice 1, créez une classe `PlanDeTable` avec les méthodes suivantes (n'oubliez pas d'en donner les complexités) :

— un constructeur de la structure à partir d'une liste de noms. On considère qu'initialement il n'y a pas d'incompatibilité ;
— une méthode `compatible(i, j)` qui prend deux indices en paramètres et rend `True` si `NOMS[i]` et `NOMS[j]` sont compatibles et `False` sinon ;
— une méthode `ajoute_incompatibilité(i, j)` qui prend deux indices en paramètres et ajoute une incompatibilité entre `NOMS[i]` et `NOMS[j]` si elle n'existe pas déjà ;
— une méthode `affiche_incompatibilités()` qui affiche toutes les incompatibilités **une unique fois**.

Pour l'exemple précédent, `affiche_incompatibilités()` pourra par exemple afficher :

```
tonton Julien – papy François
cousin Valentin – soeur Manon
cousin Valentin – papy François
papy François – soeur Manon
```

2.2 Plan de table valide

Résoudre le problème revient à trouver un plan de table (chaque invité est associé à une table unique) valide (deux invités à une même table ne doivent pas avoir d'incompatibilité), c'est-à-dire créer une liste `tables` telle que :

— chaque élément de `tables` est un ensemble d'indices tel que si $i \in \text{tables}[k]$ alors l'invité `NOMS[i]` est placé à la table numéro k
— pour tout indice $i \geq 0$ strictement plus petit que le nombre d'invités, il existe un unique k tel que $i \in \text{tables}[k]$
— si $i, j \in \text{tables}[k]$ alors $j \notin \text{IC}[i]$

Question 2.2.1 Montrez que quelles que soient les incompatibilités et le nombre d'invités, il existe un plan de table valide.

Question 2.2.2 Justifiez que pour l'exemple, le nombre minium de tables est 3.

Question 2.2.3 Combien de solutions à 3 tables différentes existe-il ?

Question 2.2.4 Montrez que si l'on supprime l'incompatibilité entre *"papy François"* et *"soeur Manon"* dans l'exemple alors il existe une solution à 2 tables.

Question 2.2.5 Ajoutez à la classe `PlanDeTable` une méthode `est_valide(tables)` prenant en paramètre une liste `tables`. Cette méthode rend `True` si `tables` est un plan de table valide et `False` sinon.

2.3 Cas particulier des relations anti-transitives

Trouvons le nombre minimum de tables d'un plan de table valide pour un cas particulier de plan de table. On se place dans le cas où la relation d'incompatibilité est **anti-transitive**, c'est-à-dire que si l'invité A est incompatible avec l'invité B et l'invité B incompatible avec l'invité C, alors l'invité A est **compatible** avec l'invité C.

Question 2.3.1 Ajoutez une méthode `est_AT()` sans paramètre à la classe `PlanDeTable`. Cette méthode rend `True` si la relation d'incompatibilité est anti-transitive et `False` sinon.

Question 2.3.2 Démontrez que s'il existe une solution à 2 tables alors la relation d'incompatibilité est anti-transitive.

Question 2.3.3 Démontrez que si la relation d'incompatibilité est anti-transitive alors il existe une solution à 2 tables.

Question 2.3.4 Déduire de la question précédente un algorithme permettant de rendre un plan de table valide à deux tables lorsque la relation d'incompatibilité est anti-transitive.

2.4 Cas général : algorithme glouton

On se propose d'écrire un algorithme glouton permettant de résoudre le problème dans le cas général (on ne suppose pas les relations anti-transitives). La structure de cet algorithme est la suivante :

1. créer une liste `ordre` contenant les indices de tous les convives ;
2. créer une liste vide `tables`
3. pour chaque élément `i` de `ordre`, ajouter `i` à la première table de `tables` possible (la première table ne contenant aucune de ses incompatibilités) si elle existe ou en créer une nouvelle sinon.

Question 2.4.1 Pourquoi l'algorithme précédent est-il glouton ?

Question 2.4.2 Démontrez qu'il donne bien une réponse au problème quel que soit `ordre`. Quel ordre utiliseriez-vous par défaut pour résoudre le problème ? Et pourquoi ?

Question 2.4.3 Ajoutez une méthode `résolution()` sans paramètre à la classe `PlanDeTable` qui est l'implémentation de l'algorithme glouton.

Question 2.4.4 Cet algorithme est efficace mais on va voir qu'il dépend fortement de la liste `ordre`. Montrez que l'algorithme peut rendre un nombre de tables strictement plus grand que 2 pour une relation anti-transitive.

Question 2.4.5 En utilisant la structure de l'algorithme glouton :
- démontrez que le nombre minimum de tables ne peut excéder le nombre maximum d'incompatibilités pour une personne plus 1 ;
- donnez un cas où cette borne est atteinte ;
- donnez un cas où on peut faire strictement mieux que cette borne.

Exercice 3 – Problème des 8 reines

Ce problème consiste à placer 8 reines sur un échiquier (possédant 8 lignes et 8 colonnes) sans qu'aucune reine ne puisse en prendre une autre. Une reine peut prendre toute pièce qui est sur sa ligne, sa colonne ou sur ses diagonales.

On modélise l'échiquier par une matrice `E` à n lignes et n colonnes ($n = 8$ pour un échiquier traditionnel) : `E[i][j]` correspond à la case à l'intersection de la ligne `i` et de la colonne `j`. Cette case est `True` si une reine y est placée et `False` sinon.

On vous demande de donner la complexité de vos algorithmes en fonction de cette taille n.

3.1 Étude préliminaire

Cette partie permet d'introduire quelques bases pour trouver une solution.

Question 3.1.1 Écrivez une fonction créant un échiquier de taille n (à n lignes et n colonnes) vide.

Question 3.1.2 Écrivez une fonction permettant de savoir si on peut placer une reine à la ligne `i` et la colonne `j` pour un échiquier donné. Elle rendra `True` si la reine peut être placée et `False` sinon.

Question 3.1.3 On peut cependant faire mieux que juste créer tous les échiquiers possibles. Montrez que l'on peut représenter le problème des n reines par une permutation de la liste `[0, ..., n-1]`.

Question 3.1.4 Écrivez un algorithme prenant en paramètre une permutation de `[0, ..., n-1]` et rendant `True` si la permutation est une solution du problème des 8 reines et `False` sinon.

3.2 Permutations

On cherche ici à construire toutes les permutations possibles.

Question 3.2.1 Créez un algorithme `supprime(L, i)` qui prend en paramètre une liste `L` et un indice `i` et qui rend une nouvelle liste contenant tous les éléments de `L` (dans le même ordre) sauf celui d'indice `i` (la liste `L` reste inchangée).

Question 3.2.2 Montrez que les permutations d'une liste `L` peuvent être obtenues à partir des permutations des listes `supprime(L, i)` avec `i` allant de 0 à `len(L) - 1`.

Question 3.2.3 Déduisez-en un algorithme récursif rendant toutes les permutations d'une liste passée en paramètre.

3.3 Résolution

On utilise les parties précédentes pour résoudre le problème des n reines.

Question 3.3.1 Déduisez des questions précédentes un algorithme permettant de résoudre le problème des n reines en examinant toutes les permutations.

Question 3.3.2 Montrez qu'il est inutile d'examiner toutes les permutations et qu'il suffit souvent de connaitre le début d'une permutation pour l'invalider. En déduire une méthode de résolution du problème n'examinant pas toutes les permutations.

Question 3.3.3 Que pouvez-vous dire des différences de complexité entre les deux méthodes de résolution ?