ALGORITHMES GLOUTONS

Rendu de TD

Les réponses aux exercices sont à rendre sur la boîte de dépôt `moodle`, en respectant les consignes suivantes :

- ☐ un fichier `README.txt` qui décrit votre avancement : exercices terminés, en cours, difficultés éventuelles rencontrées, ...
- ☐ quand c'est demandé, fournir un document texte de nom `exercice_num.txt` avec les réponses de l'exercice numéro num ;
- ☐ les fichiers sources Java **commentés avec votre/vos noms en en-tête** doivent utiliser le codage UTF-8 et respecter les noms de l'énoncé ;
- ☐ chaque programme doit proposer dans la fonction `main()` des démonstrations en mode silencieux (aucune entrée/sortie interactive), les tests peuvent être fournis à part sous forme de modules `Junit 5`.

---

1 Ordonnancement d'intervalles (interval scheduling)

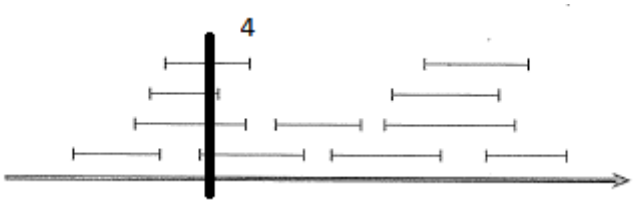

FIGURE 1 – Exemple d'ensemble de requêtes

1. Compléter le programme Java `IntervalTreeSet` qui traite des ensembles R d'intervalles (de type `Interval`) et propose les quatre algorithmes gloutons étudiés en cours pour calculer un sous-ensemble de requêtes valide avec la plus grande cardinalité possible.

   Les noms des méthodes correspondant aux quatre algorithmes sont :

   (a) `getStartEarlierFirst` ou la version récursive `getStartEarlierRec`

   (b) `getSmallerFirst`

   (c) `getFinishEarlierFirst`

   (d) `getLessConflictsFirst`

   Les méthodes 1 et 2 sont données à titre d'exemple. Vous devez écrire le corps des méthodes 3 et 4 (dans cet ordre).
2. Donner des tests significatifs des quatre méthodes (et une figure représentant les intervalles pour chaque jeu d'essai).
3. Écrire la méthode `intersectionMax` qui renvoie la taille du plus grand conflit pour un ensemble R donné. Dans la figure 1, cette taille est 4.

2 Sac à dos (0-1 knapsack problem)(suite TD2)

Un voleur cherche à dévaliser un magasin. Comme il a mal au dos, il ne peut pas porter plus qu'un poids maximum `poidsMax` mais il cherche cependant à emporter le butin de plus grande valeur possible. Le butin est choisi parmi les `Articles` du `Magasin`, pris en 0 ou 1 exemplaire.

On considère la classe `Magasin` donnée dans le TD2

- Pour l'instance de problème du magasin bio (constructeur par défaut de la classe `Magasin`), quelle est la valeur optimale du butin qu'il emportera pour des valeurs de poids maximum de 30, 311 et 1000 ?
- Compléter la méthode `private int volerAux(int poidsMax, int j)` de la classe `Magasin` qui calcule la valeur maximale possible du butin par une recherche de type « force brute ».
- Donner une 2^e version de cette méthode qui renvoie la liste des articles retenus en plus de la valeur pour le butin optimal. Utiliser la classe `SacADos` qui implémente une liste et non un tableau.
- Dans la classe `SacADos`, écrire les deux méthodes gloutonnes `biggestValueFirst` et `bestRatioValueWeightFirst` décrites en cours.
- Donner des tests significatifs des deux méthodes et les comparer. En particulier, vous devez proposer des instances qui donnent des solutions optimales et d'autres qui sont au contraire très éloignées de la solution optimale. Les tests seront aussi décrits dans `exercice_2.md`.

3 Ordonnancement de tâches (parallel machine scheduling problem)

On dispose de M machines pour exécuter un ensemble de T tâches. Sur l'exemple de la figure 2 on veut exécuter $T = 8$ tâches numérotées de 1 à 8 sur $M = 3$ machines M_1, M_2, et M_3. Les tâches ont des durées respectives de 4, 2, 4, 5, 2, 2, 2, et 3 unités de temps. Un ordonnancement consiste à choisir pour chaque tâche la machine sur laquelle elle doit s'exécuter. Une tâche ne peut être interrompue (elle est exécutée en une seule fois) et une machine exécute une seule tâche à la fois. Une fois que l'on a décidé un ordonnancement, on mesure la date de fin d'exécution de toutes les tâches (la date de celle qui termine en dernier) que l'on appelle *Makespan*. L'objectif est de minimiser le *Makespan*.

Les trois ordonnancements ci-dessus donnent respectivement (de haut en bas) des valeurs de *Makespan* égales à 9, 10 et 8.

Toutes les réponses aux questions suivantes seront écrites dans le fichier `exercice_3.md`.

1. À votre avis, quelles stratégies gloutonnes (comment a-t-on choisi l'ordre dans lequel les tâches ont été affectées aux machines) ont été utilisées pour les ordonnancements (h) haut et (m) milieu ? Quel algorithme pas forcément glouton pour (b) bas ?
2. Est-ce que l'ordonnancement (b) est optimal ? Pourquoi ?
3. Écrivez en Java un algorithme d'ordonnancement qui vous semble approprié (l'un de ceux que vous avez proposés en réponse à la question 1, ou un différent). Le code sera écrit dans la classe `Scheduler.java`.
4. Expliquez pourquoi cet algorithme peut être qualifié de « glouton ».
5. Expliquez dans quels cas votre algorithme résulte en un $Makespan$ optimal. Donnez des exemples du type de ceux de la figure 2.

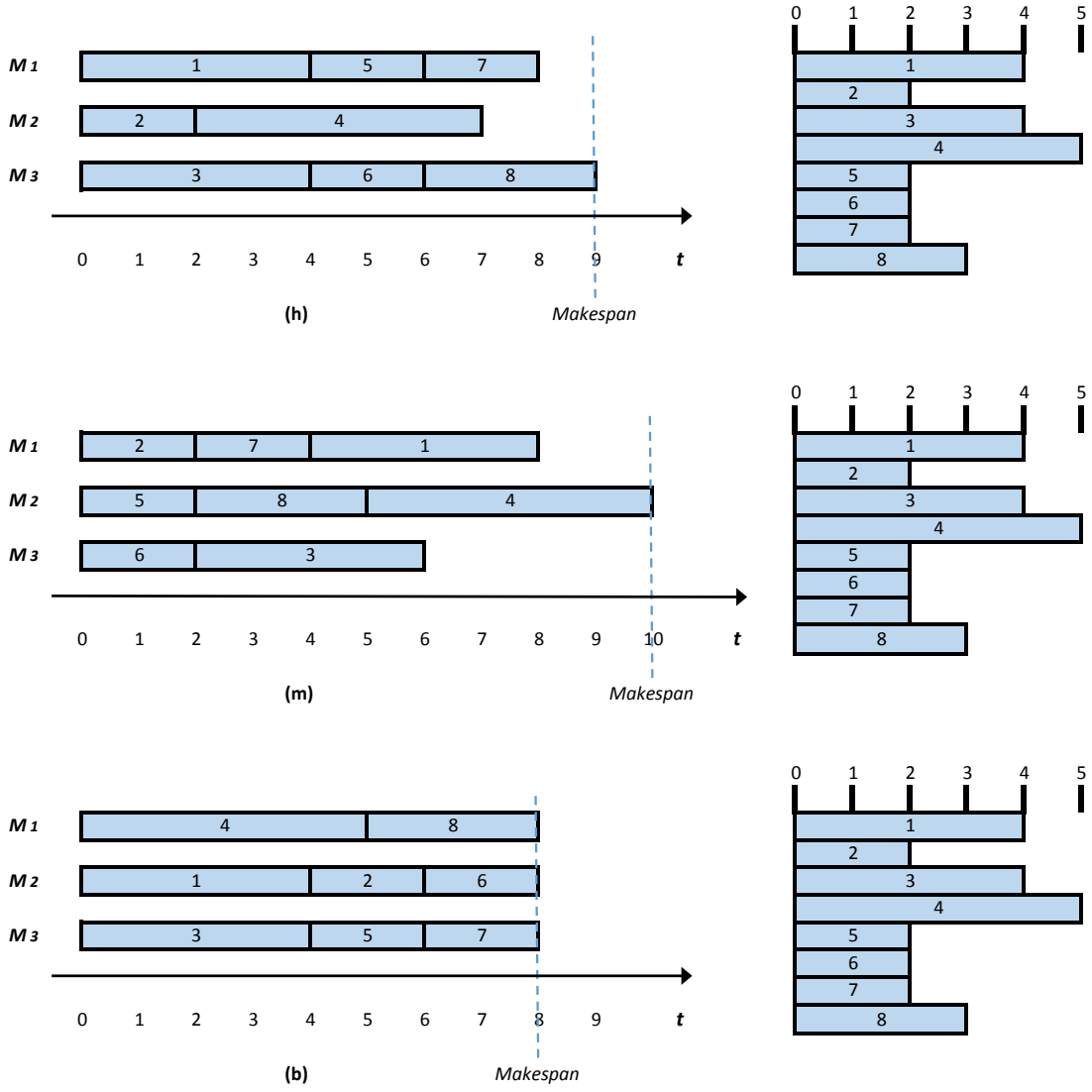

FIGURE 2 – Ordonnancement de 8 tâches sur 3 machines

6. Expliquez dans quels cas votre algorithme résulte en un $Makespan$ « mauvais ». Donnez des exemples du type de ceux de la figure 2.
7. Si on suppose maintenant que pour toutes les tâches la durée est soit 2, soit 4. Quel nouvel algorithme proposez-vous ?
8. Écrire une solution Brute Force qui traite les petites instances de problèmes.