Algorithmique - ROB3 - TP3
Algorithmes gloutons

L'objet de ce TP est d'implémenter des algorithmes gloutons pour divers problèmes d'ordonnancement. Un problème d'ordonnancement consiste à planifier la réalisation de n tâches sur une ou plusieurs machines. Une tâche j est caractérisée par sa date de début $d[j]$ et sa date de fin $f[j]$ (autrement dit, c'est un intervalle de temps). Chaque machine ne peut réaliser qu'une tâche à la fois (elle peut toutefois réaliser une tâche i et une tâche j pour lesquelles $f[i] = d[j]$). L'objectif de l'ordonnancement peut être de maximiser le nombre de tâches sur une machine, ou encore de minimiser le nombre de machines nécessaires pour réaliser toutes les tâches.

Exercice 1 (Génération aléatoire des instances)

Coder un générateur aléatoire d'instances. Il s'agira de générer n intervalles aléatoires correspondant aux différentes tâches, sous la forme de deux tableaux d et f, où $d[j] \in \mathbb{N}$ est la date de début de la tâche j et $f[j] \in \mathbb{N}$ est la date de fin de la tâche j. Le générateur prendra en paramètre une date de fin $F_{\max}$ au plus tard d'une tâche (c'est-à-dire une borne supérieure sur la date de fin d'une tâche : $\forall j,\ f[j] \leq F_{\max}$). Les intervalles seront de longueurs variées.

Exercice 2 (Ordonnancement d'un nombre maximum de tâches sur une machine)

Dans cet exercice, on s'intéresse à déterminer le nombre maximum de tâches mutuellement compatibles (problème vu en cours sous le nom de "sélection d'intervalles"), c'est-à-dire dont les intervalles de réalisation ne se chevauchent pas (hormis éventuellement aux extrémités).

Q 2.1 Une approche récursive de résolution par "force brute" s'écrit comme ci-dessous, pour des tâches j supposées triées par valeurs croissantes de $f[j]$, avec **Calcule-OPT**(n) comme appel initial et $\text{der}(j) = \max\{i : i < j \text{ et } i \text{ compatible avec } j\}$. Coder l'algorithme et indiquer sa complexité.

```
Calcule-OPT(j)
si j = 0 alors
    retourner 0
sinon
    retourner max{1 + Calcule-OPT(der[j]), Calcule-OPT(j − 1)}
fin si
```

Q 2.2 L'algorithme glouton vu en cours s'écrit comme suit, où *dispo* est la date de disponibilité de la machine étant donné les tâches déjà sélectionnées et k est le nombre courant de tâches sélectionnées :

```
Trier les tâches j par valeurs croissantes de f[j]
dispo ← 0 ; k ← 0
pour j = 1 à n faire
    si d[j] ≥ dispo alors
        k ← k + 1
        dispo ← f[j]
    fin si
fin pour
retourner k
```

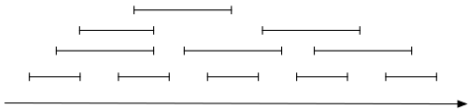
Coder cet algorithme. Indiquer sa complexité.

Q 2.3 Comparer expérimentalement le gain de temps obtenu en exécutant les deux algorithmes sur les mêmes instances générées aléatoirement (en faisant croître progressivement n).

Q 2.4 En utilisant un tableau T de n booléens avec $T[j] = 1$ si la tâche j est sélectionnée, et $T[j] = 0$ sinon, adapter l'algorithme glouton précédent pour qu'il retourne l'ensemble des tâches sélectionnées.

Exercice 3 (Ordonnancement de tâches sur un nombre minimum de machines)

Dans cet exercice, on considère le problème d'ordonnancement où l'on vise à déterminer le nombre minimum de machines identiques à mobiliser pour réaliser l'ensemble des n tâches. Comme les machines sont identiques, n'importe quelle tâche peut bien sûr être réalisée sur n'importe quelle machine. Voici un exemple d'ordonnancement sur 4 machines pour un ensemble de 11 tâches :



Q 3.1 Une approche gloutonne pour le problème d'ordonnancement étudié ici va consister à trier les tâches dans un ordre encore à définir, puis, pour chaque tâche j dans l'ordre considéré, à l'assigner sur une machine disponible à l'instant $d[j]$. Si il n'y a pas de machine disponible, on ajoute une nouvelle machine. Cette approche peut se formaliser sous la forme de l'algorithme qui suit, où p est un compteur du nombre de machines mobilisées et $dispo[i]$ indique la date de disponibilité de la machine i (date de fin de la dernière tâche assignée à la machine i).

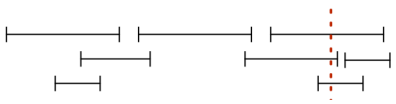
```
p ← 0
pour j = 1 à n faire
    si la tâche j peut être assignée à une machine déjà présente alors
        Soit i la première machine disponible
        dispo[i] ← f[j]
    sinon
        Ajouter une nouvelle machine p + 1
        dispo[p + 1] ← f[j]
        p ← p + 1
    fin si
fin pour
retourner p
```

Il s'agit maintenant de déterminer l'ordre dans lequel trier les tâches. On envisage les trois possibilités suivantes :

1. trier les tâches en ordre croissant des $f[j]$,
2. trier les tâches en ordre croissant des $d[j]$,
3. trier les tâches en ordre croissant des $f[j] - d[j]$.

Identifier des contre-exemples permettant de montrer que deux de ces trois alternatives ne conduisent pas à un ordonnancement optimal (autrement dit, qui ne minimise pas le nombre de machines mobilisées). L'alternative restante conduit à un ordonnancement optimal. Coder l'algorithme correspondant. Indiquer sa complexité.

Q 3.2 On appelle *profondeur* d'un ensemble d'intervalles le nombre maximum d'intervalles *ouverts* dont l'intersection commune est non-vide (pas d'intersection aux extrémités, par exemple $]1, 2[\cap]2, 3[= \emptyset$). Par exemple, sur la figure ci-dessous, la profondeur de l'ensemble d'intervalles est 3.



Pour concevoir un algorithme qui calcule la profondeur d'un ensemble d'intervalles, l'idée est de considérer toutes les extrémités des intervalles en ordre croissant (en cas d'égalité, priorité à une extrémité

finale sur une extrémité initiale). Il s'agit ensuite de parcourir cette liste en incrémentant un compteur p (initialisé à 0) de 1 pour chaque extrémité initiale rencontrée, et en décrémentant p de 1 pour chaque extrémité finale rencontrée. La valeur maximale qu'atteint p lors de ce parcours est la profondeur de l'ensemble d'intervalles. Par exemple, soit $d = [1, 3, 4, 5, 9, 10]$ et $f = [2, 8, 5, 7, 11, 12]$, où l'intervalle j a pour extrémité initiale $d[j]$ et pour extrémité finale $f[j]$. La profondeur est alors 2 :

date	extrémité	p
1	début	1
2	fin	0
3	début	1
4	début	2
5	fin	1
5	début	2
7	fin	1
8	fin	0
9	début	1
10	début	2
11	fin	1
12	fin	0

Pour un codage efficace, on ne créera pas une seule liste triée des extrémités, mais plutôt un liste triée des extrémités initiales et une liste triée des extrémités finales. On parcoura ensuite ces deux listes en utilisant la même approche que dans le tri fusion pour parcourir les extrémités dans l'ordre croissant. Coder cet algorithme. Indiquer sa complexité.

Q 3.3 Vérifier expérimentalement que le nombre de machines dans un ordonnancement optimal est toujours égal à la profondeur de l'ensemble d'intervalles caractérisant les tâches.

Exercice 4 (Le cadeau)

Rendez-vous sur le site www.codingame.com.

Une fois rendu(e) sur le site, réalisez les opérations suivantes :

- Ouvrez dans le menu latéral de gauche l'onglet "ENTRAÎNEMENT".
- Dans la section "PUZZLE CLASSIQUE - MOYEN" cherchez et cliquez sur le puzzle "THE GIFT".
- Cliquez sur le bouton "RÉSOUDRE" en haut à droite. Un éditeur de texte s'ouvre.
- Sélectionnez le langage de programmation C.

Pour résoudre le puzzle "The Gift", vous devez coder votre réponse dans l'éditeur de texte qui vient de s'ouvrir. La description du puzzle est présente à gauche de votre écran. La réponse attendue est un algorithme glouton que vous devez concevoir. Cet algorithme est-il correct pour toutes les instances possibles ?
Une fois votre réponse codée et le jeu de test passé avec succès, soumettez votre réponse sur le site.