TD 5 : Algorithmes gloutons

Exercice 1 – *Interval partitionning*

Maintenant, on va s'intéresser à un problème plus proche de l'exemple des emplois du temps qu'on avait décrit en cours en discutant d'algorithmes gloutons :

Interval partitionning

Entrée : Un ensemble d'intervalles ouverts $I = \{]a_i, b_i[\}_{0 \leq i \leq n-1}$. $]a_i, b_i[$ est appelée la requête numéro i, $deb(i) = a_i$ la date de début de la requête i et $fin(i) = b_i$ la fin de la requête i. On suppose que pour tout $i \in [\![0, n-1]\!]$, $deb(i) < fin(i)$.

Sortie : une plus petite[a] partition de I en $S_1, \dots, S_k$ telle que chaque partie S_i contient des requêtes compatibles.

[a]. ici, plus petite veut dire partition contenant le moins de parties de I

On peut voir ce problème comme un problème d'affectation de ressources : on veut trouver une répartition de requêtes à des ressources telles que chaque ressource ne peut traiter qu'une requête à la fois.
On va introduire quelques définitions pour décrire et résoudre ce problème.

Définition 1 – Clique d'intervalles

Une *clique d'intervalles* est un ensemble d'intervalles X deux à deux non disjoints : pour tous $r_1, r_2 \in X$ distincts, on a $r_1 \cap r_2 \neq \emptyset$. On note $\chi(X)$ le cardinal de la clique la plus grande de X.

▶ **Question 1** Montrer qu'une solution du problème de partitionnement d'intervalles (pas forcément une solution optimale) est de cardinal supérieur ou égal à $\chi(I)$.

On considère la fonction AlgoGlouton pour résoudre ce problème.

▶ **Question 2** Montrer proprement que la solution renvoyée par cet algorithme est bien une solution du problème (potentiellement pas la plus petite).

▶ **Question 3** Proposer un ensemble d'intervalles I pour lequel la solution proposée par l'algorithme glouton n'est pas optimale.

▶ **Question 4** Supposons maintenant que l'on a préalablement trié les requêtes par ordre de début croissant : $deb(0) \leq deb(1) \leq \dots \leq deb(n-1)$. Montrer que la solution renvoyée est optimale.

Pseudo-code

Requiert : On note $I = \{r_0, r_1, \dots r_{n-1}\}$ avec $n = |X|$.
Fonction AlgoGlouton(I)
  On affecte la requête r_0 sur la ressource 0.
  $d \leftarrow 1$ ▷ *contient le nombre de ressources $0 \dots d-1$ déjà utilisées.*
  Pour $i = 1$ à $n-1$, **faire**
    on cherche $0 \leq k \leq d-1$ la ressource la plus petite pouvant accueillir r_i
    Si si un tel k existe **alors**
      Ajouter la requête r_i à la ressource k
    Sinon
      Ajouter la requête r_i à une nouvelle ressource d
      $d \leftarrow d+1$
  retourne la partition de I des requêtes dans chaque ressource $0 \dots d-1$.

On peut l'implémenter avec une complexité en $\mathcal{O}(n \log n)$ en utilisant la structure de tas min pour stocker les dates de fin des dernières requêtes ajoutées à chaque ressource. On verra cette structure de donnée bientôt. En attendant :

★ **Question 5** Implémenter cet algorithme. Le format des requêtes et le langage utilisé est libre. On attend une complexité au plus en $\mathcal{O}(n^2)$.

Exercice 2 – Le problème de rendu de monnaie

C'est le cas le plus classique d'algorithme glouton qui ne renvoie pas systématiquement de solutions optimales : supposons que vous disposez d'un système monétaire $S = \{a_0, a_1, \dots, a_{p-1}\}$ où chaque a_i correspond à une quantité d'argent représentable par une pièce / un billet dans ce système monétaire. On suppose que $S \subseteq \mathbb{N}$, et que les a_i sont triés dans l'ordre croissant.
Soit $n \in \mathbb{N}$: l'objectif de cet exercice est de déterminer un algorithme permettant de décomposer une quantité d'argent n en le moins de pièces et billets possible. On l'énonce formellement comme suis :

Problème du rendu de monnaie

Entrée : Un système monétaire $S = \{a_0, a_1, \dots, a_{p-1}\}$, et un entier $n \in \mathbb{N}$.

Sortie : Un p-uplet $(x_0, \dots, x_{p-1})$ tel que :
- $\sum_{i=0}^{p-1} x_i a_i = n$
- $\sum_{i=0}^{p-1} x_i$ est minimal

La stratégie gloutonne tentant de déterminer un tel p-uplet pour $n \in \mathbb{N}$ est de choisir la plus grande pièce $a_k \leq n$, puis décomposer récursivement $n - a_k$ en continuant de suivre la même stratégie gloutonne.

▶ **Question 1** Expliciter cette stratégie en pseudo-code. On supposera que S est représenté un tableau strictement croissant d'entiers, et la sortie également par un tableau.

▶ **Question 2** Déterminer sa complexité temporelle.

★ **Question 3** Implémenter cette stratégie par une fonction C de prototype `int* rendu_de_monnaie_glouton(int* systeme, int p)`.

Les systèmes monétaires tels que cette stratégie produit une solution optimale sont dits *canoniques*. C'est le cas de toutes les monnaies actuelles. Dans le cas général, vous verrez peut-être en MPI que le problème du rendu de monnaie est NP-complet.

▶ **Question 4** Montrer que le système monétaire des euros $\{1, 2, 5, 10, 20, 50, 100, 200, 500\}$ est canonique.

★ **Question 5** Dans *Harry Potter et la chambre des secrets*, Hagrid dit à Harry :

> The gold ones are Galleons. Seventeen silver Sickles to a Galleon and twenty-nine Knuts to a Sickle, it's easy enough. [a]

Déterminer le système monétaire du monde d'Harry Potter, et déterminer s'il est canonique.

★ **Question 6** Implémenter par la programmation dynamique une fonction résolvant le problème de rendu de monnaie, même quand le système n'est pas canonique.

[a] Aucune idée d'où j'ai mis mon exemplaire, désolé pour l'anglais !

Exercice 3 – Je suis en retard

Un petit retour aux problèmes de requêtes et d'affectation de tâches.
Supposons que l'on dispose d'un ensemble de n requêtes noté X, une requête x étant décrite par une durée d'exécution $e(x) \in \mathbb{N}$ et un horaire de fin souhaité $f(x) \in \mathbb{N}$.
L'objectif va être d'exécuter toutes ces requêtes en minimisant le retard total : en notant $t(x)$ le temps auquel on termine la requête x, on note le retard associé $r(x) = \max(0, t(x) - h(x))$. On cherche donc à minimiser $r = \sum_{x \in X} r(x)$. On considère que l'on peut commencer à exécuter des requêtes à partir du temps 0, et que l'on ne peut exécuter qu'une requête à la fois.
On va choisir la stratégie gloutonne dans laquelle on va exécuter les requêtes par horaire de fin souhaité croissant, en ignorant la durée de la requête : le reste de l'exercice cherche à montrer que cette stratégie construit une solution optimale.
Soit une solution du problème $\langle x_0, x_1, \dots, x_{n-1} \rangle$. Notons $d(x_i)$ la date à laquelle on commence à exécuter la requête i.

▶ **Question 1** On dit que la solution est sans temps mort si pour tout $i \in [\![0, n-2]\!]$, on a $d(x_i) + e(x_i) = d(x_{i+1})$ (on commence à exécuter la requête x_{i+1} dès qu'on a fini d'exécuter la requête x_i). Montrer que l'on peut se limiter à étudier les solutions sans temps mort.

▶ **Question 2** Est-ce qu'une solution optimale au problème est forcément sans temps mort ?

On considère maintenant que notre solution est sans temps mort.

▶ **Question 3** Supposons que la solution $\langle x_0, x_1, \dots, x_{n-1} \rangle$ est sans temps mort et qu'il existe $i \in [\![0, n-2]\!]$ tel que $f(x_i) > f(x_{i+1})$. Montrer que la solution $\langle x_0, x_1, \dots, x_{i-1}, x_{i+1}, x_i, x_{i+2}, \dots, x_{n-1} \rangle$ a un plus petit retard que $\langle x_0, x_1, \dots, x_{n-1} \rangle$.

▶ **Question 4** Conclure.

★ **Question 5** L'implémenter. Vous êtes libres de la représentation et du langage.