On travaillera en OCaml, depuis la machine virtuelle ClefAgreg2019 et Emacs, ou depuis `https://BetterOCaml.ml/`.

On rappelle qu'on compile ce fichier avec `COMPILATEUR = ocamlc` ou `ocamlopt` avec la ligne de commande suivante, puis on exécute le binaire produit avec la deuxième ligne (sans les dollars qui représentent le prompt de la ligne de commande du terminal).

```
$ COMPILATEUR -o TP23.exe TP23.c
$ ./TP23.exe
```

Suite au cours de mardi 10/05, on s'intéresse à des problèmes d'ordonnancement.

Problème 1 : *Interval Scheduling*

Entrées : un ensemble non vide d'intervalles non vides $S = \{]a_i, b_i[, 0 \leqslant i \leqslant n1\}$. On parlera de requête numéro i pour l'intervalle $]a_i, b_i[$, et l'on notera $s(i) = a_i$ et $f(i) = b_i$ (*start* et *finish*).

Sortie : le plus grand sous-ensemble de S (au sens de la cardinalité) constitué d'intervalles deux-à-deux disjoints. On parlera de *requêtes compatibles*.

Algorithme glouton : à chaque étape, on choisit une requête compatible avec celles déjà choisies, suivant une certaine règle, et l'on élimine les requêtes devenues incompatibles.

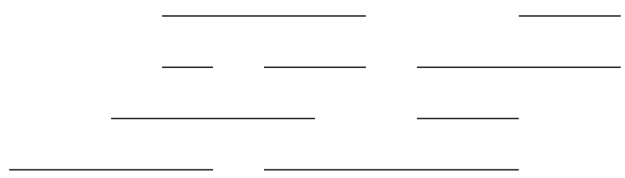

Figure 1 – Une instance du problème IntervalScheduling.

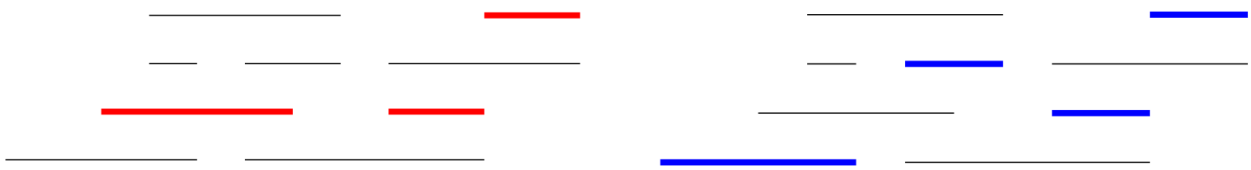

Figure 2 – Deux solutions : non optimale (en rouge) à gauche, et optimale (en bleu) à droite.

Plusieurs choix de règles sont envisageables :

— *i)* prendre la requête commençant le plus tôt ;
— *ii)* prendre la requête la plus courte (i.e., minimisant $b_i a_i$) ;
— *iii)* prendre la requête ayant le moins de conflits avec des requêtes non encore choisies ;
— *iv)* prendre la requête se terminant en premier.

Questions théoriques - *à faire maintenant, et finir chez soi si besoin*

1. Montrer que les trois premières règles ne fournissent pas (en général) une solution optimale. Autrement dit, pour chacune des trois premières règles *i)*, *ii)* et *iii)*, il faut exhiber un exemple d'instance pour lequel la solution gloutonne suivant cette règle ne donne pas une solution optimale.
2. Optimalité de l'algorithme glouton :
   On note $G = (i_1, \dots, i_m)$ les requêtes choisies par l'algorithme glouton suivant la règle *iv)*, classées dans l'ordre croissant (de leurs horaires de fin). On considère un autre ensemble $H = (j_1, \dots, j_p)$ de requêtes compatibles, également classées, avec $p \geqslant m$.
   a. Montrer par récurrence (finie) sur $1 \leqslant r \leqslant p$ la propriété P_r : « $f(i_r) \leqslant f(j_r)$ » ($f(i)$ est l'heure de *fin* de la requête i).
   b. Conclure.

Implémentation en OCaml

3. Proposez un type enregistrement `requete` en OCaml, qui permette de représenter une requête i, qui encapsule les trois données suivantes : son heure de début $s(i) = a_i$ et son heure de fin $f(i) = b_i$, deux `float`, et son numéro i (`int`).
4. Écrire une fonction `disjoints : requete -> requete -> bool` qui vérifie si deux requêtes r_i et r_j sont deux intervalles disjoints.

On se contentera de représenter S une instance du problème d'IntervalScheduling et G une solution (partielle ou optimale) par une liste de `requete`. On pourrait utiliser un ensemble via le module `Set` ou une table de hachage via le module `Hashtbl`, mais cela compliquerait le code.

5. Écrire une fonction `compatibles : (requete list) -> bool` qui vérifie que toutes les requêtes de l'ensemble de requêtes fourni sont bien deux-à-deux compatibles. *Astuce* : on pourra utiliser deux appels à `List.for_all : ('a -> bool) -> 'a list -> bool` imbriqués, en utilisant que deux requêtes (pas forcément différentes) `ri` et `rj` sont compatibles *ssi* elles sont égales ou bien disjointes.
6. Sachant que l'on dispose de la fonction `List.sort`, de signature `('a -> 'a -> int) -> 'a list -> 'a list`, expliquer comment s'en servir pour trier une `requete list` selon différent critère (ceux des règles *i)* à *iv)*).
   — Pour le rôle du premier argument, de signature `'a -> 'a -> int`, la documentation explique ceci : *The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller.*
   — Pour plus de détails, allez lire la documentation `https://ocaml.org/api/List.html#VALsort`.
   — Si vous pensez qu'une des règles ne peut pas être implémentée directement en utilisant cette fonction de tri `List.sort`, expliquez pourquoi.
7. Pour chacun des quatre règles *i)* à *iv)*, si c'est possible, implémenter une fonctions `compare_i` (ou `compare_ii`, `compare_iii`, `compare_iv`), de signature `requete -> requete -> int`.
8. Utiliser les pour écrire quatre fonctions `tri_i`, `tri_ii`, `tri_iii` et `tri_iv` (si c'est possible), utilisant ces fonctions de comparaison de requêtes et `List.sort`.

9. Écrire une fonction `filtre_incompatibles : requete list -> requete -> requete list` qui prend un ensemble E de requêtes, une requête r, et renvoie un ensemble E' sous-ensemble de E, constitué des requêtes différentes et disjointes de r.
   - Autrement dit, cette fonction élimine les requêtes devenues incompatibles, après avoir choisi r dans l'ensemble E.
   - Il faudra faire attention à préserver l'ordre de la liste initiale, car on la triera qu'une seule fois dans l'algorithme glouton.
   - Si vous utilisez une référence sur une liste, initialement vide, et que vous ajoutez chaque requête r_j compatible avec r_i en tête de la référence (avec un appel à `List.iter (fun rj -> ...) liste_r`), à la fin vous aurez besoin d'utiliser `List.rev` pour "renverser" la liste. Essayez la sur un exemple, ou allez lire sa documentation en ligne, si vous ne vous rappelez pas de son fonctionnement.
   - Vous pouvez aussi utiliser `List.filter : ('a -> bool) -> 'a list -> 'a list`, qui préserve l'ordre.
10. Implémenter l'algorithme glouton, pour chacune des règles *i)* à *iv)* avec un appel initial à la fonction `tri_X` (`X=i`, `X=ii`, `X=iii` ou `X=iv`), si possible (vous devriez pouvoir le faire pour trois des quatre règles). Ces fonctions seront appelées `intervalscheduling_glouton_X`.

— On pourra écrire une fonction récursive `aux_glouton : requete list -> requete list`, qui suppose la liste de requêtes donnée en argument `liste_r` déjà triée selon le critère à utiliser par l'algorithme glouton.
   - Si `liste_r` a zéro ou un seul argument, on la renvoie telle quelle.
   - Sinon, `liste_r = ri :: queue`, et on sélectionne `ri` son premier argument, et on appelle récursivement `aux_glouton` sur `filtre_incompatibles queue ri`.
11. Sans chercher à être spécialement très efficace, si on suppose que le tri `List.sort` s'effectue en temps $\mathcal{O}(n\log(n))$ (ce qui est bien le cas), quelle sera la complexité temporelle pire cas d'une de ces fonctions `tri_X`, exprimée en fonction de n le nombre de requêtes ?

Tests et exemples On pourra commencer par faire un dessin bien propre au brouillon, illustrant les différents intervalles choisis pour les requêtes utilisées pour les exemples. On peut évidemment se limiter à des valeurs `start` et `finish` à valeurs entières, en les écrivant `2.0` par exemple.

12. Définir au moins une dizaine d'exemples de requêtes, certaines incompatibles avec d'autres, numéroté `r1` à `r10` (au moins).
13. Tester les fonctions précédentes.
14. Tester les fonctions `intervalscheduling_glouton_X`, et si possible vérifier que les règles *i)* et *ii)* ne donnent pas toujours une solution optimale.

Problème 2 : *Interval Partitioning*

Entrées : un ensemble non vide d'intervalles non vides $S = \{[a_i, b_i[, 0 \leqslant i \leqslant n1\}$. On parlera de requête numéro i pour l'intervalle $]a_i, b_i[$, et l'on notera encore $s(i) = a_i$ et $f(i) = b_i$ (*start* et *finish*).

Sortie : une partition de S en k groupes de requêtes, chacun des groupes devant être compatibles (cf. problème 1), et k devant être minimal.

Remarque : Essentiellement, la question posée est : « étant données ces n conférences ayant chacune un horaire de début et de fin fixés, de combien de salles a-t-on besoin au minimum ? ».

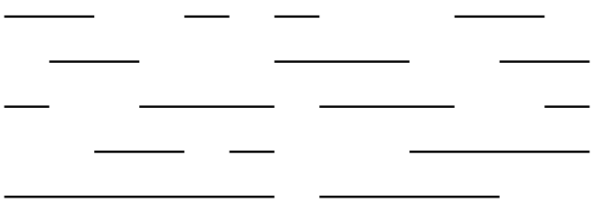

Figure 3 – Une instance du problème IntervalPartitioning.

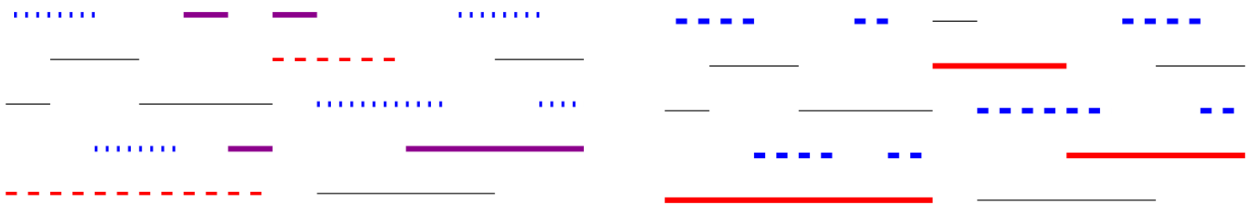

Figure 4 – Solutions : non-optimale (à gauche) avec $k = 4$, et optimale (à droite) avec $k = 3$.

On considère l'algorithme (glouton) suivant :

- Trier les n requêtes, dans un certain ordre (précisé ultérieurement) $[r_1, \ldots, r_n]$.
- Programmer la requête r_1 sur la ressource 1.
- $d = 1$ (d contiendra toujours le nombre de ressources allouées jusqu'à présent).
- Pour i de 2 à n :
  - Chercher le plus petit $k \leqslant d$ tel que l'on puisse programmer la requête r_i sur la ressource k.
  - *Si* l'en existe un : programmer la requête i sur la ressource k.
  - *Sinon* : programmer r_i sur $d + 1$, et $d = d + 1$.

15. Montrer que cet algorithme renvoie une solution compatible.
16. Exhiber une instance et un ordre de traitement des requêtes pour lesquels la solution renvoyée n'est pas optimale.
17. On suppose désormais que le tri des requêtes se fait par ordre de début croissant. Montrer que la solution renvoyée est optimale.
18. Implémenter cet algorithme glouton en OCaml et le tester sur des exemples.

