Sauf indication contraire, il ne vous est pas demandé de prouver vos algorithmes. Néanmoins, il sera tenu compte de la qualité des explications accompagnant ceux-ci.

Problème 1 - Partage de ressources

Un laboratoire offre un unique ordinateur de grande puissance pour effectuer les simulations numériques réclamées par les équipes de recherche. Celles-ci doivent donc partager le temps d'utilisation de la machine. Chaque équipe réserve des plages horaires de calcul. On veut donc un outil qui permette de planifier ces réservations. L'heure de départ de l'agenda de réservation est 0.

On représente une plage horaire non vide commençant à l'heure d et terminant à l'heure f et réservée par l'équipe n (d, f, n seront supposés entiers naturels et $n \geqslant 1$) par un triplet de trois entiers de type `int * int * int`, où le premier entier est d, le deuxième f et le troisième est n. On supposera toujours qu'une plage horaire `(d,f,n)` est valide : l'heure de début est strictement inférieure à l'heure de fin.

On pourra utiliser librement les fonctions suivantes de type `int * int * int -> int` :

```
let heure_debut pl =
  match pl with | (d,f,n) -> d;;
let heure_fin pl =
  match pl with | (d,f,n) -> f;;
let equipe pl =
  match pl with | (d,f,n) -> n;;
let amplitude pl =
  match pl with | (d,f,n) -> f - d;;
```

Dans toute la suite, le symbole `plage` est un synonyme de `int * int * int`.

On tient à jour une liste de plages de type `plage list`, qui contient les réservations déjà effectuées. Cette liste est valide si les plages horaires sont disjointes et rangées dans l'ordre **décroissant** du temps : la tête de la liste est la plage horaire la plus tardive.

Par exemple, l'équipe 2 a réservé la plage $15 - 19$ et l'équipe 5 la plage $20 - 31$:

la liste correspondante est `[(20,31,5); (15,19,2)]`.

Mathématiquement, une plage horaire est un intervalle $[a, b[$ (ouvert à droite). Par conséquent, les plages horaires $[a, b[$ et $[b, c[$ sont disjointes...

Question 1) Fonctions sur les plages.

a) Écrivez en code CAML une fonction `compare p q` de type `plage -> plage -> int`, qui donne -1 si la plage horaire `p` est antérieure à la plage `q` (*i.e.* `p` se termine avant que `q` ne commence), 1 si la plage `p` est postérieure à `q` et 0 dans tous les autres cas.

b) Écrivez en code CAML une fonction `disjointes p q` de type `plage -> plage -> bool`, qui indique si les plages sont disjointes.

Question 2) Validation d'une liste.

Écrivez une fonction `valide li` de type `plage list -> bool`, qui indique si la liste de plages `li` est valide (remarque : la liste vide est valide). Quelle est sa complexité en fonction de la longueur de la liste ? Prouvez sa correction par récurrence sur la longueur de la liste et sa terminaison.

Dans toute la suite du problème, il est supposé que les listes de plages horaires sont valides.

Question 3) Occupation à une certaine heure.

Écrivez une fonction `qui li h` de type `plage list -> int -> int`, qui donne le numéro de l'équipe utilisant l'ordinateur à l'heure `h` ou 0 si l'ordinateur est libre à cette heure.

Question 4) Plage horaire libre.

Écrivez une fonction `libre pl li` de type `plage -> plage list -> bool`, qui indique si la plage horaire `pl` est disponible ou pas.

Question 5) Plus longue occupation.

Écrivez une fonction occ_max li de type `plage list -> plage`, qui donne la plage la plus longue de la liste des réservations. Si la liste est vide, la fonction donnera en résultat le triplet (-1,-1,-1) ;

Question 6) Annulation.

Écrivez une fonction annuler pl li de type `plage -> plage list -> plage list`, qui construit la liste obtenue en supprimant la plage horaire pl. Si la plage horaire n'existe pas dans la liste, le résultat est la liste elle-même.

Question 7) Insertion d'une plage horaire.

Écrivez une fonction inserer pl li de type `plage -> plage list -> plage list`, qui construit, si c'est possible, la nouvelle liste après insertion de la plage horaire : cette insertion doit respecter la validité de la liste (liste ordonnée dans l'ordre décroissant). Si c'est impossible, la fonction lève une exception par failwith "message".

Question 8) Insertion d'une plage sous contrainte de durée.

L'équipe numéro n veut avoir un temps de calcul, dès que possible. Écrivez une fonction offre n d li de type `int -> int -> plage list -> plage list`, qui construit la liste obtenue après insertion d'une plage horaire d'amplitude d le plus tôt possible. Vous pouvez utiliser la fonction rev, qui calcule la liste inversée d'une liste. Sachant que cette dernière de complexité linéaire par rapport à la longueur de la liste, quelle est la complexité de votre fonction ?

Devoir non surveillé 2 - Corrigé

Problème 1

Question 1)

a) Le code ci-dessous est explicite.

```
let compare p q =
        if heure_fin p <= heure_debut q then -1
        else if heure_fin q <= heure_debut p then 1
        else 0;;
```

b) Deux plages sont disjointes si et seulement si l'une est avant ou après l'autre...

```
let disjointes p q =
        compare p q <> 0;;
```

Question 2)

```
let rec valide li =
        match li with
        | [] -> true
        | [p] -> true
        | p :: q :: su -> (compare p q = 1) && (valide (q :: su));;
```

L'algorithme termine évidemment, car la longueur de la liste décroît strictement à chaque appel récursif et que les cas de base correspondant sont présents. Les deux cas de base sont la liste vide et les listes à une seule plage horaire, qui sont évidemment valides, ce que confirme l'algorithme.

Si l'algorithme est correct pour les listes de longueur n ($n \geqslant 1$), alors soit `li` une liste de longueur $n+1$; elle est de longueur au moins 2, donc elle s'écrit `[p; q; ... ]`.

Si la liste est valide, alors sa queue l'est aussi, donc l'appel `valide (q :: su)` retourne "vrai" par hypothèse de récurrence. La liste étant valide, sa plage horaire de tête est postérieure à la plage suivante, donc `compare p q` vaut 1, donc le test `compare p q = 1` vaut "vrai". Avec le « et » logique, la fonction retourne donc "vrai" pour la liste `li`.

Si maintenant la liste n'est pas valide, alors si sa queue n'est pas valide, par hypothèse de récurrence, l'appel `valide (q :: su)` retourne "faux", donc avec le « et » logique, la fonction retourne "faux" ; et si la queue est valide, alors il ne reste comme seule possibilité que `p` et `q` ne sont pas rangées dans l'ordre décroissant du temps, donc le test `compare p q = 1` vaut "faux", avec le « et » logique, la fonction retourne donc "faux" pour la liste `li`.

Dans les deux cas, la fonction retourne "vrai" si et seulement si la liste est valide.

D'après le principe de récurrence, l'algorithme est correct pour toutes les listes.

La comparaison est de complexité $O(1)$, les tests sur les motifs aussi ainsi que le « et » logique, donc la formule de récurrence sur la complexité est $C(n) = C(n-1) + O(1)$, donc $C(n) = O(n)$.

Question 3) Cas de base : liste vide, pas de réservation donc on retourne 0. Sinon, on teste si l'heure `h` appartient à la première plage horaire de la liste : si oui, on retourne le numéro de l'équipe, sinon on fait un appel récursif pour explorer le reste de la liste et voir si une autre plage horaire contient l'heure `h`

```
let rec occupe li h =
        match li with
        | [] -> 0
        | p :: su -> if (h >= heure_debut p && h < heure_fin p) then equipe p
                     else occupe su h;;
```

Question 4) Une plage horaire est disponible si et seulement si elle est disjointe de toutes les plages horaires déjà réservées. C'est ce que teste la fonction suivante.

```
let rec libre pl li =
        match li with
        | [] -> true
        | p :: su -> (disjointes p pl) && (libre pl su);;
```

Question 5) C'est une adaptation de la fonction de recherche de maximum dans une liste (exercice classique).

```
let rec occ_max li =
        match li with
        | [] -> (-1,-1,-1)
        | p :: su -> let m = occ_max su in
                     if amplitude p > amplitude m then p else m;;
```

Question 6) Même chose avec la suppression d'un élément (ça marche car il est supposé que la liste est valide, donc ne contient pas deux fois le même élément).

```
let rec annuler pl li =
        match li with
        | [] -> []
        | p :: su -> if p = pl then su else p :: (annuler pl su);;
```

Question 7) Si la plage à insérer est postérieure à la première plage de la liste (cas 1), alors on la place en tête. Si elle est non disjointe avec la première plage (cas 2), alors on ne peut pas réserver aux heures dites, donc on retourne une erreur. Si elle est antérieure (cas 3), alors par appel récursif on tente de l'insérer dans la queue de la liste. Le cas de base est le cas de la liste vide, dans laquelle on peut toujours insérer la plage horaire.

```
let rec inserer pl li =
        match li with
        | [] -> [pl]
        | p :: su -> if compare pl p = 1 then pl :: li
                else if compare pl p = 0 then failwith "insertion␣impossible"
                else p :: (inserer pl su);;
```

Question 8) Comme la liste des plages horaires est rangée par ordre décroissant du temps, on la retourne d'abord pour avoir un ordre dans le sens croissant du temps. Offrir une plage horaire le plus tôt possible est donc déterminer le premier intervalle de longueur suffisante dans cette liste. La fonction récursive interne (avec accumulateur) exécute cette recherche : on l'appelle avec un paramètre h qui représente l'heure à laquelle on essaye de placer la plage horaire demandée. Initialement, il vaut 0. À chaque appel récursif, on examine si l'espace disponible entre l'heure h et le début de la première plage horaire est assez long pour y placer la plage demandée (test 1) : si c'est le cas, on place la plage en tête de la liste et on s'arrête, sinon (cas 2) on replace la première plage en tête et on examine la queue de la liste à partir d'une nouvelle heure h, égale à l'heure de fin de la première plage. Le cas de base est la liste vide, dans laquelle on place à tout coup la plage à l'heure h. Puis on retourne de nouveau la liste pour la remettre dans l'ordre décroissant.

La fonction interne parcourt linéairement la liste et à chaque étape, elle ne fait que quelques tests et opérations élémentaires, donc sa complexité est linéaire. Avec les deux retournements, eux-mêmes linéaires, on aboutit à une complexité linéaire.

```
let offre n d li =
        let il = rev li in
        let rec offraux h il =
                match il with
                | [] -> [(h,h+d,n)]
                | p :: su -> if (heure_debut p >= d + h) then
                                        (h,h+d,n) :: il   (* 1 *)
                             else p :: offraux (heure_fin p) su   (* 2 *)
        in
        let il2 = offraux 0 il in
        rev il2;;
```