Important. Une grande importance sera apportée à la clarté, la lisibilité des raisonnements et des programmes.

Remarque. On pourra toujours librement utiliser une fonction demandée à une question précédente, même si cette question n'a pas été traitée.

1 Programmation en OCaml

Important. Dans cette partie, on rédigera les programmes dans le langage OCaml.

Exercice 1 (Typage) : Déterminer le type des fonctions suivantes :

1.
```
let fonction1 x y =
  if y mod 2 = 0 then x.(y) else 0.
;;
```

2.
```
let rec fonction2 x y = match x,y with
  | [],_ -> 0
  | _,0 -> 0
  | a::q,_ -> a + fonction2 q (y-1)
;;
```

3.
```
let fonction3 x y z = (x y) (y z) ;;
```

Exercice 2 (Problème de l'arrêt) : Si `f` est une fonction, et `x` un argument de `f`, on dit que `f` termine sur `x` si l'appel `f x` renvoie un résultat au bout d'un certain temps, ou est interrompu à la suite d'une erreur, par exemple une division par zéro ou un problème de type.

1. Déterminer les entiers pour lesquels la fonction suivante termine, et démontrer la terminaison de cette fonction sur cet ensemble d'entiers.

```
let rec f1 n = match n with
  | 0 -> 0
  | _ -> n + f1 (n-2)
;;
```

2. Même question pour la fonction :

```
let f2 n =
  let i = ref n in
  while !i != 10 do
    incr i
  done
;;
```

3. Écrire une fonction `infini` qui ne termine sur aucun entier.
4. L'objectif de cette question est d'examiner la possibilité de décider automatiquement (via un algorithme) si une fonction `f` termine sur une entrée `x`, en analysant le code de `f`.

   On suppose l'existence d'une fonction curryfiée `arret` prenant 2 arguments :

   - une chaîne de caractère `code_f` correspondant au code d'une fonction `f` ;
   - un objet de type quelconque `x` ;

   et qui renvoie `true` si `f` termine sur `x`, et `false` sinon.

   (a) Donner le type de la fonction `arret`.

(b) Écrire une fonction `bizarre` prenant en argument une chaîne de caractère `code_f` correspondant au code d'une fonction `f`, ainsi qu'un objet de type quelconque `x`, telle que `bizarre code_f x` termine si et seulement si `f` **ne** termine **pas** sur `x`.
Le résultat renvoyé par `bizarre` lorsqu'elle termine n'a aucune importance.

(c) Écrire une fonction `paradoxe` prenant en argument une chaîne de caractère `code_f` correspondant au code d'une fonction `f`, telle que `paradoxe code_f` termine si et seulement si `f` **ne** termine **pas** sur `code_f`.

(d) En utilisant `code_paradoxe`, le code de la fonction `paradoxe`, exhiber une contradiction et conclure.

Exercice 3 (Analyse d'algorithme) : On considère la fonction suivante :

```
let f n =
  let s = ref 1 in
  let c = ref 0 in
  while !s <= n do
    incr c ;
    s := !s + 2 * !c + 1
  done ;
  !c
;;
```

1. Démontrer la terminaison de `f`.
2. Pour $n \in \mathbb{N}$, conjecturer une relation entre `f n` et `n`. En déduire une spécification de `f`.
3. Démontrer la correction de `f` relativement à cette spécification.
4. Déterminer la complexité de `f`.

Exercice 4 (Valeur majoritaire) :

1. Écrire une fonction `valeur_majoritaire` prenant en argument un tableau `t` et renvoyant un élément de `t` dont le nombre d'occurrences est maximal.
   Par exemple :
   `valeur_majoritaire [|1;3;2;1;2;1|]` doit renvoyer 1.
   `valeur_majoritaire [|1;3;2;1;2|]` peut renvoyer 1 ou 2.
2. Démontrer la terminaison de cette fonction.
3. Déterminer la complexité de cette fonction.
4. Dans le cas d'une fonction itérative, donner pour chaque boucle utilisée un invariant de boucle permettant de déduire la correction de la fonction.
   Dans le cas d'une fonction récursive, donner la propriété à démontrer par récurrence.
   On ne demande pas le détail des démonstrations elles-mêmes.
5. On suppose qu'on dispose d'une fonction `trier` qui prend en argument un tableau `t` et renvoie un tableau **trié** contenant les mêmes éléments, et ayant une complexité dans le pire cas en $O(n \log n)$ (où n est la taille de `t`).
   En utilisant cette fonction, améliorer la complexité du calcul de la valeur majoritaire.

Exercice 5 (Recherche dichotomique) : Dans cet exercice, on s'intéresse à la recherche par dichotomie d'un élément dans un tableau trié, implémentée par la fonction suivante :

```
let dicho t x =
  let n = Array.length t in
  let g = ref 0 in
  let d = ref (n-1) in
  while !d - !g > 0 do
    let m = (!g + !d)/2 in
    if t.(m) >= x then d := m
    else g := m
  done ;
  t.(!g) = x
;;
```

1. Déterminer le type de la fonction `dicho`.
2. Pour quelle raison cette fonction ne commence-t-elle pas par appliquer la fonction `trier` de l'exercice précédent sur le tableau argument, ce qui permettrait de ne pas forcément supposer que le tableau soit initialement trié ?
3. Justifier que la proposition suivante est préservée à chaque passage dans la boucle while :

$$P : x \text{ apparait dans } \texttt{t} \text{ si, et seulement si, il apparait entre les indices } g \text{ et } d \text{ inclus.}$$

4. La fonction `dicho` ne termine pas. L'illustrer sur un exemple.
5. Corriger la fonction pour qu'elle termine sur toute entrée tout en restant correcte.
6. Démontrer la terminaison et la correction de la fonction corrigée.
7. (a) Écrire une fonction `tricho`, ayant la même spécification que `dicho`, mais découpant l'espace de recherche en 3 plutôt qu'en 2 à chaque étape.
   Cette fonction devra être récursive (ou utiliser une fonction auxiliaire récursive).
   (b) Déterminer la complexité de `tricho`.

2 Programmation en C

Important. Dans cette partie, on rédigera les programmes dans le langage C. On suppose que les librairies standards `stdbool.h`, `stdio.h` et `stdlib.h` sont déjà importées et vous n'avez pas besoin de le préciser.

Exercice 6 (Multi-ensembles) : Un multi-ensemble est un objet mathématique représentant une collection non ordonnée d'éléments. À la différence d'un ensemble, un multi-ensemble peut contenir plusieurs occurrences d'un même élément. Un multi-ensemble M d'éléments d'un ensemble S peut être décrit au moyen d'une fonction de S dans $\mathbb{N}$ indiquant pour chaque élément, le nombre de fois qu'il apparaît dans l'ensemble M. On typographie (invention de l'enseignant) un multi-ensemble de la manière suivante $X = \{\{\pi, \sqrt{2}, \pi, \sqrt{2}, e, \sqrt{2}, e, 1.5\}\}$. Puisque les éléments ne sont pas ordonnés, on remarque que l'on a aussi $X = \{\{\pi, \pi, \sqrt{2}, \sqrt{2}, \sqrt{2}, e, e, 1.5\}\}$.

1. Donner la représentation de X sous forme d'une fonction de $\mathbb{R}$ dans $\mathbb{N}$.

On appelle cardinal d'un multi-ensemble son nombre d'éléments. Si x apparaît n fois dans le multi-ensemble il compte pour n dans le cardinal du multi-ensemble.

2. Donner le cardinal de X.

Dans toute la suite on s'intéresse à des multi-ensembles de cardinal fini. On se propose de construire une représentation en C des multi-ensembles permettant de faire les opérations ensemblistes usuelles d'ajout d'un élément et d'union.

Un multi-ensemble sera représenté par un tableau C alloué dynamiquement. Plutôt que d'avoir plusieurs occurrences d'un même élément dans le tableau, nous allons préférer stocker dans une case du tableau, non seulement l'élément mais aussi son nombre d'occurrences dans le multi-ensemble. Par exemple l'ensemble X sera représenté par un tableau contenant : π et 2 dans une case, $\sqrt{2}$ et 3 dans une seconde, e et 2 dans une troisième et finalement 1.5 et 1 dans une quatrième.

3. Définir un type structuré `atome` contenant deux champs : un flottant `cle` contenant l'élément représenté et un entier `nb_occur` représentant le nombre d'occurrences de l'élément dans le multi-ensemble.

Afin de pouvoir manipuler plus facilement le tableau d'`atome` nous sauvegardons sa taille dans une structure.

4. Définir un type structuré `multiset` contenant deux champs : un tableau d'`atome`s `contenu` et un entier `nb_cle` indiquant la taille du tableau `contenu`.
   Remarque : le tableau `contenu` devra toujours être *complètement* rempli.
5. En supposant que π a été approximé par le flottant `3.14`, e par le flottant `2.72` et $\sqrt{2}$ par le flottant `1.41`, donner une représentation schématique de la mémoire représentant le multi-ensemble X.

6. Écrire une fonction `multiset empty();` ne prenant aucun argument et retournant le multi-ensemble vide.

Dans la suite on assure que si un atome apparait dans le tableau `contenu` d'un multi-ensemble alors son nombre d'occurences est supérieur strictement à 0.

7. Écrire une fonction `int cardinal(multiset m);` retournant le cardinal du multi-ensemble qui lui est passé en argument.
8. Écrire une fonction `int mem(float cle, multiset m);` permettant de tester si la clé `cle` est dans le multi-ensemble `m`. Si la clé apparaît en effet dans le tableau `contenu` de `m`, on retournera l'indice dans le tableau de l'atome de clé `cle`. Sinon on retournera `-1`.
9. Écrire une fonction `void ajout_non_present(float cle, int nb_occur, multiset* pm);` permettant l'ajout de `nb_occur` éléments `cle` dans le multi-ensemble pointé par `pm`. Comme l'indique le nom de la fonction, on supposera que `cle` n'est pas présent dans le multi-ensemble. On remarquera à la signature de la fonction que l'on ne retourne pas un nouveau multi-ensemble mais que celui passé en argument doit être modifié par votre fonction.
10. En déduire une fonction `void ajout(float cle, int nb_occur, multiset* pm);` permettant l'ajout de `nb_occur` éléments `cle` dans le multi-ensemble pointé par `pm`.

On définit mathématiquement l'union de deux multi-ensembles X_1 et X_2 représenté en termes de fonctions $S \to \mathbb{N}$ comme étant le multi-ensemble :

$$\begin{cases} S \to \mathbb{N} \\ x \mapsto X_1(x) + X_2(x) \end{cases}$$

L'union des multi-ensembles $\{\{\pi, \pi\}\}$ et $\{\{\pi, \sqrt{2}\}\}$ est donc le multi-ensemble $\{\{\pi, \pi, \pi, \sqrt{2}\}\}$.

11. En déduire une fonction `multiset union1(multiset m1, multiset m2);` calculant l'union des deux multi-ensembles `m1` et `m2` passés en arguments à la fonction.
12. Donner la complexité asymptotique (en nombre d'itérations), au moyen d'un Θ, de votre fonction `union1` en terme de $n = n_1 + n_2$ où n_1 est la taille du tableau `contenu` de `m1` et n_2 est la taille du tableau `contenu` de `m2`

La complexité obtenue à la question précédente étant très mauvaise, on se propose de l'améliorer en contraignant notre structure. Nous supposons dorénavant que le tableau `contenu` d'un multi-ensemble est toujours *trié* suivant le champs `cle` de ses atomes.

13. Donner une représentation schématique de la mémoire, satisfaisant une telle contrainte, représentant le multi-ensemble X.
14. Définir une fonction `multiset union2(multiset m1, multiset m2);` calculant l'union de deux multi-ensembles.
    On pourra s'aider d'une fonction auxiliaire `int cardinal_definition(multiset m1, multiset m2);` prenant en argument deux multi-ensembles `m1` et `m2` et calculant la taille du tableau `contenu` qui doit être alloué pour représenter l'union des deux multi-ensembles.
15. Donner la complexité asymptotique (en nombre d'itérations), au moyen d'un Θ, de votre fonction `union2` en terme de $n = n_1 + n_2$ où n_1 est la taille du tableau `contenu` de `m1` et n_2 est la taille du tableau `contenu` de `m2`.