Pour les élèves ayant la certitude de prendre l'option SI au second semestre, un sujet en Python est disponible à la fin.

I Petites questions

1. On définit le type d'arbre binaire : `type 'a tree = E | N of 'a * 'a tree * 'a tree`.
   Ecrire une fonction `prefix : 'a tree -> 'a list` permettant d'obtenir la liste du parcours préfixe d'un arbre, de préférence en complexité linéaire en le nombre de noeuds.

   Solution : Voir cours.

2. Définir la matrice suivante en C, par la méthode de votre choix :

$$\begin{pmatrix} 1 & 7 & 4 \\ 3 & 2 & 0 \end{pmatrix}$$

   Solution :

```c
int m[2][3] = {
    {1, 7, 4},
    {3, 2, 0}
};
```

II Égalité de Kraft

On considère ici des arbres binaires stricts (chaque noeud possède 0 ou 2 fils). Étant donné un arbre a et un sommet s de a, on définit $p(s, a)$ comme la profondeur de s dans a.

1. Démontrer l'égalité suivante (appelée égalité de Kraft), pour tout arbre binaire strict a :

$$\sum_{f \in \mathcal{F}(a)} 2^{-p(f,a)} = 1$$

   où $\mathcal{F}(a)$ est l'ensemble des feuilles de a.

   ▶ Si a est un arbre binaire strict, on note $P(a)$ la propriété « $\sum_{f \in \mathcal{F}(a)} 2^{-p(f,a)} = 1$ ».

   Nous allons montrer $P(a)$ par **induction structurelle**[1], c'est à dire $P(V)$ (cas de base: arbre vide) et que: $P(g) \wedge P(d) \implies \forall r,\ P(r, g, d)$ (étape d'induction, $\wedge$ étant le « et logique »).

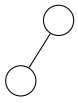
   - $P(V)$ est vrai car un arbre vide n'a qu'une feuille de profondeur 0 et $2^0 = 1$.
   - Supposons $P(g)$ et $P(d)$ vrais pour deux arbres binaires stricts g et d. Soit r une étiquette. Alors, si f est une feuille de g, $p(f, a) = p(f, g)+1$. De même, si f est une feuille de d, $p(f, a) = p(f, d)+1$. Comme de plus $\mathcal{F}(a) = \mathcal{F}(g) \cup \mathcal{F}(d)$:

$$\sum_{f \in \mathcal{F}(a)} 2^{-p(f,a)} = \sum_{f \in \mathcal{F}(g)} 2^{-p(f,a)} + \sum_{f \in \mathcal{F}(d)} 2^{-p(f,a)} = \sum_{f \in \mathcal{F}(g)} 2^{-p(f,g)-1} + \sum_{f \in \mathcal{F}(d)} 2^{-p(f,d)-1}$$

$$= \frac{1}{2}\left(\sum_{f \in \mathcal{F}(g)} 2^{-p(f,g)} + \sum_{f \in \mathcal{F}(d)} 2^{-p(f,d)}\right) \underset{P(g) \wedge P(d)}{=} \frac{1}{2}(1+1) = 1$$

   Remarque : on pouvait aussi le faire par récurrence (sur le nombre de noeuds par exemple), ce qui est très similaire.

2. Est-ce que cette égalité subsiste pour un arbre binaire non strict?

   ▶ Non, $\sum_{f \in \mathcal{F}(a)} 2^{-p(f,a)} = \frac{1}{2}$ si a est l'arbre suivant : 

[1] c'est une généralisation du principe de récurrence sur les entiers

III Trie (arbre préfixe)

Un *trie* est un arbre (non-binaire) permettant de gérer un ensemble de mots (chaînes de caractères). Ils sont utiles pour la complétion automatique (sur téléphone ou sur un environnement de développement) ou pour la correction orthographique.
A chaque arête du *trie* est associé un caractère. Pour savoir si un mot appartient au *trie*, on part de la racine et on lit les caractères du mot, un par un, en descendant suivant l'arête correspondante à chaque fois. Voici un exemple de *trie* contenant les mots cat, dog, door, deer, pan et panda : Sur chaque noeud, on met un booléen permettant de savoir si le chemin de la racine à ce noeud correspond à un mot ou pas. Dans l'exemple ci-dessus, les noeuds en noir foncé correspondent à un mot et pas les autres. Par exemple, pan est un mot du *trie* mais pas do.

On utilise le type suivant :

```
typedef struct trie {
    bool word; // le noeud en cours correspont-il à un mot?
    trie* childs[26]; // tableau des fils
} trie;
```

Si `t` est un `trie*`, `t->childs` est un tableau tel que `t->childs[0]` est le fils de `t` associé à la lettre `'a'`, `t->childs[1]` est le fils de `t` associé à la lettre `'b'`...
Par exemple, si `t` est le *trie* sur le dessin ci-dessus, `t->childs[0]` est `NULL` (pas d'arête sortante de la racine avec la lettre `'a'`), `t->childs[2]` correspond au sous-arbre le plus à gauche, `t->childs[3]` au sous-arbre du milieu...
On supposera utiliser uniquement des chaînes de caractères avec des caractères entre `'a'` et `'z'`.

1. Ecrire une fonction `int c_to_i(char c)` convertissant un caractère `c` (entre `'a'` et `'z'`) en un entier. Par exemple, `c_to_i('a')` doit renvoyer `0`, `c_to_i('b')` doit renvoyer `1`...

   Solution :

   ```
   int c_to_i(char c) {
       return c - 'a';
   }
   ```

2. Ecrire une fonction `trie* new_node()` créant un nouveau `trie*`, avec un attribut `word` égal à `false` et `childs` rempli de `NULL`.

   Solution :

   ```
   trie* new_node() {
       trie* t = (trie*)malloc(sizeof(trie));
       t->word = false;
       for(int i = 0; i < 26; i++)
           t->childs[i] = NULL;
       return t;
   }
   ```

3. Ecrire une fonction `bool has(trie* t, char* s)` déterminant si `s` appartient à `t`. On pourra soit parcourir l'arbre récursivement en utilisant un indice pour savoir quel caractère de `s` est en train d'être lu, soit utiliser une boucle `while`.

   Solution : On peut rajouter un indice qui va changer lors des appels récursifs :

   ```
   bool has(trie* t, char* s, int i) {
       if(i == strlen(s))
           return t->word;
       int k = c_to_i(s[i]);
       if(t->childs[k] == NULL)
           return false;
       return has(t->childs[k], s, i+1);
   }
   ```

   On peut aussi utiliser une boucle :

```
bool has(trie* t, char* s) {
    for(int i = 0; s[i] != '\0'; i++) {
        k = c_to_i(s[i]);
        if(!t->childs[k])
            return false;
        t = t->childs[k];
    }
    return t->word;
}
```

4. Ecrire une fonction `void add(trie* t, char* s)` ajoutant le mot `s` dans le *trie* `t`. Il faut donc parcourir les caractères de `s` en ajoutant les noeuds correspondants dans `t`, si besoin. On supposera que le `t` donné en argument n'est pas vide.

Solution : Version récursive avec un argument supplémentaire :

```
void add(trie* t, char* s, int i) {
    if(i == strlen(s)) {
        t->word = true;
        return t;
    }
    int k = c_to_i(s[i]);
    if(t->childs[k] == NULL)
        t->childs[k] = new_node();
    return add(t->childs[k], s, i + 1);
}
```

IV Arbre d'intervalles

Dans cette partie, on représentera un intervalle (au sens mathématique) $I = [a, b]$ par un couple (a, b) en OCaml, définit par le type suivant : `type interval == int*int`
Cette définition de type permet simplement d'utiliser `interval` au lieu de `int*int`, ce qui est un peu plus clair.
On note de plus $I_0 = a$ la borne inférieure de l'intervalle, et $I_1 = b$ la borne supérieure.

1. Soient I et J deux intervalles. Donner une condition nécessaire et suffisante sur I_0, I_1, J_0, J_1 pour que les intervalles I et J s'intersectent (c'est-à-dire : $I \cap J \neq \emptyset$). Prouver cette condition.

Solution :

- Supposons $I \cap J \neq \emptyset$ et soit $x \in I \cap J$. Comme $x \in I$, $I_0 \leq x \leq I_1$. Comme $x \in J$, $J_0 \leq x \leq J_1$. On en déduit $I_0 \leq x \leq J_1$ et $J_0 \leq x \leq I_1$. Donc :
$$I_0 \leq J_1 \quad (1)$$
$$J_0 \leq I_1 \quad (2)$$

- Montrons que cette condition nécessaire est aussi suffisante. Soit donc deux intervalles I et J tels que $I_0 \leq J_1$ et $J_0 \leq I_1$.
Distinguons deux cas possibles : $I_1 \leq J_1$ ou $I_1 > J_1$.
Si $I_1 \leq J_1$ alors $J_0 \overset{(1)}{\leq} I_1 \leq J_1$ donc $I_1 \in J$ et $I \cap J \neq \emptyset$.
Si $I_1 > J_1$ on trouve de façon symétrique que $J_1 \in I$ donc $I \cap J \neq \emptyset$.

On a donc bien démontré que
$$\boxed{I \cap J \neq \emptyset \iff I_0 \leq J_1 et J_0 \leq I_1}$$

2. En déduire une fonction `intersect : interval -> interval -> bool` déterminant si deux intervalles s'intersectent.

Solution :

```
let intersect i j =
    fst i <= snd j && fst j <= snd i;;
```

On définit une relation d'ordre $\leq$ sur des intervalles I et J par :

$$I \leq J \iff I_0 \leq J_0$$

Un **arbre d'intervalles** est un arbre binaire de recherche (pour $\leq$) dont chaque noeud possède une étiquette qui est un intervalle. Ainsi, si le noeud d'étiquette I a pour sous-arbre gauche g et sous-arbre droit d :

- Pour toute étiquette J dans g, $J_0 \leq I_0$
- Pour tout étiquette J dans d, $J_0 \geq I_0$

De plus, chaque noeud v contient une information supplémentaire qu'on note $m(v)$: la borne maximale des intervalles dans le sous-arbre enraciné en v.
L'intérêt des arbres d'intervalles est qu'ils permettent de rechercher rapidement un intervalle (parmi un ensemble d'intervalles) intersectant un point ou un intervalle donné. Ceci est utile en informatique graphique.

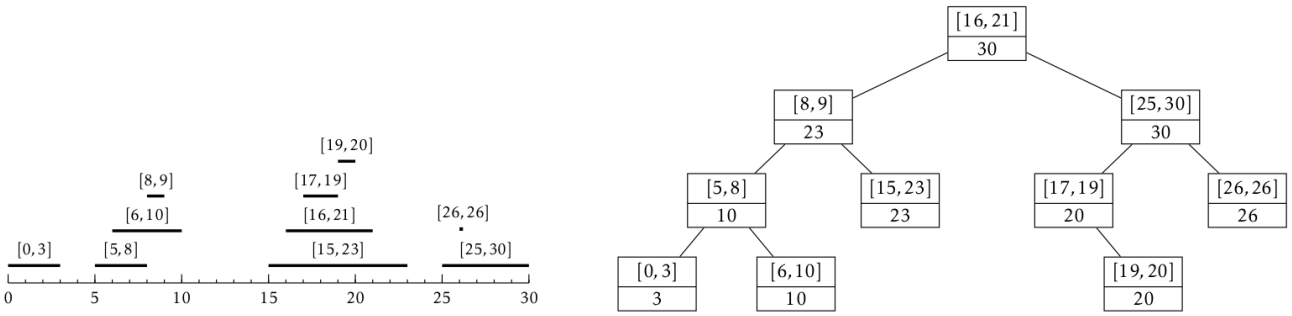


FIGURE 1 – Un ensemble de 10 intervalles, et l'arbre d'intervalles qui les représente.

Dans l'exemple ci-dessus, un ensemble d'intervalles est représenté à gauche et un arbre d'intervalles correspondant à droite. La racine r de l'arbre a pour étiquette $I = [16, 21]$ et $m(r) = 30$ nous indique que la plus grande borne parmi les intervalles dans l'arbre est 30 (correspondant à $[25, 30]$).
On utilisera le type suivant en OCaml pour représenter un arbre d'intervalles :

```
type interval_tree = E | N of interval * int * interval_tree * interval_tree
```

Ainsi, `N(i, m, g, d)` est un arbre dont la racine est l'intervalle `i`, le sous-arbre gauche est g, le sous-arbre droit est d et que la plus grande borne dans un noeud de l'arbre est `m`.

3. Écrire une fonction `maxi : 'a interval_tree -> int` qui renvoie $m(t)$ pour un arbre d'intervalles t. Il suffit donc de renvoyer le « `m` » de la racine de t.

Solution :

```
let maxi = function
    | E -> min_int (* max d'un ens. vide = -infini *)
    | N(_, m, _, _) -> m
```

Dans les deux questions suivantes, on considère un intervalle J et un arbre d'intervalle t de racine I et de sous-arbres g et d. Supposons que J intersecte un intervalle $K \neq I$ dans t et que g est non vide.

4. (Facile) Montrer que si $J_0 > m(g)$ alors J intersecte un intervalle dans d.

Solution : Si $J_0 > m(g)$ alors K ne peut pas être dans g. Comme de plus $K \neq I$, K est forcément dans d.

5. (Moins facile) Montrer que si $J_0 \leq m(g)$ alors J intersecte un intervalle dans g.

Solution : Comme $K \neq I$, K appartient soit à g, soit à d. Si $K \in g$, il n'y a rien à démontrer (J intersecte bien un intervalle de g). Supposons donc que $K \in d$.

Soit L un intervalle de g dont la borne supérieure est $m(g)$. Notons aussi $J' = J \cap K$ et montrons que $J' \subseteq L$, ce qui montrera bien que J intersecte un intervalle de g. En effet :

- $J'_1 \leq J_1 \leq m(g) = L_1$ par hypothèse
- $J'_0 \geq K_0 \geq I_0 \geq L_0$ car $K \in d$ et $L \in g$.

On a donc bien $J' = [J'_0, J'_1] \subseteq L$ donc L intersecte J' et donc, aussi, J.

6. En déduire une fonction `tree_intersect : interval_tree -> interval -> intersect` telle que, si `t` est un arbre d'intervalle et `j` un intervalle, `tree_intersect t j` renvoie un intervalle de `t` intersectant `j`. Si un tel intervalle `j` n'existe pas, on pourra déclencher une exception (`failwith...`).
Cette fonction doit être en complexité linéaire en la hauteur de l'arbre.

Solution :

```
let rec tree_intersect j t = match t with
    | E -> failwith "pas d'intervalle"
    | N(r, m, g, d) -> if intersect i j then i
       else if fst j > maxi g then tree_intersect j d
       else tree_intersect j g
```

7. Écrire une fonction `make` telle que, si `j` est un intervalle, et `g`, `d` deux arbres d'intervalles, `make j g d` renvoie un arbre d'intervalles dont la racine est `j` et dont les sous-arbres gauche et droit sont `g` et `d`.

Solution :

```
let make j g d =
    N(j, max (snd j) (max (maxi g) (maxi d)), g, d)
```

8. Écrire une fonction `add : interval -> interval_tree -> interval_tree` ajoutant un noeud dans un arbre d'intervalles, tout en conservant la propriété d'être un arbre d'intervalles.

Solution : C'est en fait très similaire à la fonction `add` du cours, avec comme seules différences qu'il y a une information supplémentaire sur chaque noeud et que les étiquettes sont des couples.

```
let rec add i = function
    | E -> N(i, snd i, E, E)
    | N(r, m, g, d) -> let m = max m (snd i) in
       if fst i <= fst r then N(r, m, add i g, d)
       else N(r, m, g, add i d)
```

9. Écrire une fonction `del : interval -> interval_tree -> interval_tree` supprimant un noeud dans un arbre d'intervalles, tout en conservant la propriété d'être un arbre d'intervalles. On pourra utiliser la même méthode que dans le cours, sur les arbres binaires de recherche.

Solution :

```
let rec del_max = function
    | E -> failwith "vide"
    | N(r, m, g, E) -> r, g
    | N(r, m, g, d) -> let maxd, d' = del_max d in
                       maxd, make r g d';;

let rec del i t = match t with
    | E -> failwith "l'intervalle n'existe pas"
    | N(r, m, g, d) -> if i = r then let maxg, g' = del_max g in
       make maxg g' d
       else if fst i < fst r then make r (del i g) d
       else make r g (del i d)
```