

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 :**

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

## 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  $a$ .

**Solution :** 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**, 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  $\mathcal{F}(a) = \mathcal{F}(g) \cup \mathcal{F}(d)$ :

$$\begin{aligned} \sum_{f \in \mathcal{F}(a)} 2^{-p(f, a)} &= \sum_{f \in \mathcal{F}(g)} 2^{-p(f, a)} + \sum_{f \in \mathcal{F}(d)} 2^{-p(f, a)} = \sum_{f \in \mathcal{F}(g)} 2^{-p(f, g)-1} + \sum_{f \in \mathcal{F}(d)} 2^{-p(f, d)-1} \\ &= \frac{1}{2} \left( \sum_{f \in \mathcal{F}(g)} 2^{-p(f, g)} + \sum_{f \in \mathcal{F}(d)} 2^{-p(f, d)} \right) \stackrel{P(g) \wedge P(d)}{=} \frac{1}{2} (1 + 1) = 1 \end{aligned}$$

**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 :



### 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 :

---

```
1 typedef struct trie {
2     bool word; // le noeud en cours correspond-il à un mot?
3     trie* childs[26]; // tableau des fils
4 } 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 :**

---

```
1 int c_to_i(char c) {
2     return c - 'a';
3 }
```

---

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

**Solution :**

---

```
1 trie* new_node() {
2     trie* t = (trie*)malloc(sizeof(trie));
3     t->word = false;
4     for(int i = 0; i < 26; i++)
5         t->childs[i] = NULL;
6     return t;
7 }
```

---

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 :

---

```
1 bool has(trie* t, char* s, int i) {
2     if(i == strlen(s))
3         return t->word;
4     int k = c_to_i(s[i]);
5     if(t->childs[k] == NULL)
6         return false;
7     return has(t->childs[k], s, i+1);
8 }
```

---

On peut aussi utiliser une boucle :

---

```

1  bool has(trie* t, char* s) {
2      for(int i = 0; s[i] != '\0'; i++) {
3          k = c_to_i(s[i]);
4          if(!t->childs[k])
5              return false;
6          t = t->childs[k];
7      }
8      return t->word;
9  }

```

---

4. Ecrire une fonction `void add(trie* t, char* s)` ajoutant le mot `s` dans la 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 :

---

```

1  void add(trie* t, char* s, int i) {
2      if(i == strlen(s)) {
3          t->word = true;
4          return t;
5      }
6      int k = c_to_i(s[i]);
7      if(t->childs[k] == NULL)
8          t->childs[k] = new_node();
9      return add(t->childs[k], s, i + 1);
10 }

```

---

## IV Arbre d'intervalles

Dans cette partie, on représentera un intervalle (au sens mathématique)  $I = [a, b]$  par un couple  $(a, b)$  en OCaml, défini 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  $J_0(1)I_1 \leq J_1$  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

$$I \cap J \neq \emptyset \iff I_0 \leq J_1 \text{ et } J_0 \leq I_1$$

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

**Solution :**

---

```

1  let intersect i j =
2      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 toute 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.

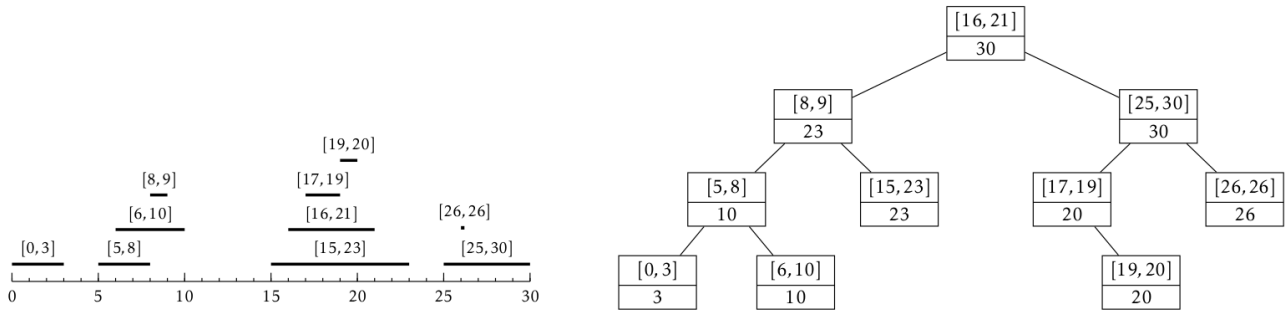


FIGURE 1 – Un ensemble de 10 intervalles, et l'arbre d'intervalles qui les représente.

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 :**

---

```

1  let maxi = function
2      | E -> min_int (* max d'un ens. vide = -infini *)
3      | 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 :**

```

1 let rec tree_intersect j t = match t with
2   | E -> failwith "pas d'intervalle"
3   | N(r, m, g, d) -> if intersect i j then i
4       else if fst j > maxi g then tree_intersect j d
5       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 :**

```
1  let make j g d =
2      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.

```

1  let rec add i = function
2    | E -> N(i, snd i, E, E)
3    | N(r, m, g, d) -> let m = max m (snd i) in
4        if fst i <= fst r then N(r, m, add i g, d)
5        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 :**

```

1 let rec del_max = function
2   | E -> failwith "vide"
3   | N(r, m, g, E) -> r, g
4   | N(r, m, g, d) -> let maxd, d' = del_max d in
5                       maxd, make r g d';;
6
7 let rec del i t = match t with
8   | E -> failwith "l'intervalle n'existe pas"
9   | N(r, m, g, d) -> if i = r then let maxg, g' = del_max g in
10                          make maxg g' d
11                       else if fst i < fst r then make r (del i g) d
12                       else make r g (del i d)

```

**Exercice 1**

1

8
5
2
4

2.1

```
def hauteur_pile(P):  
    Q = creer_pile_vide()  
    n = 0  
    while not est_vide(P):  
        n = n + 1  
        x = depiler(P)  
        empiler(Q,x)  
    while not est_vide(Q):  
        x = depiler(Q)  
        empiler(P,x)  
    return n
```

2.2

```
def max_pile(P,i):  
    # si la pile comporte moins de i élément ou que i=0 on renvoie 0  
    if i > hauteur_pile(P) or i==0:  
        return 0  
    maxi = depiler(P)  
    Q = creer_pile_vide()  
    empiler(Q,maxi)  
    j = 1  
    indice = 1  
    while j < i:  
        j = j + 1  
        x = depiler(P)  
        if x > maxi:  
            maxi = x  
            indice = j  
        empiler(Q,x)  
    while not est_vide(Q):  
        empiler(P, depiler(Q))  
    return indice
```

3

```
def retourner(P,j):
    Q1 = creer_pile_vide()
    Q2 = creer_pile_vide()
    i = 0
    while not est_vide(P) and i < j:
        i = i + 1
        x = depiler(P)
        empiler(Q1, x)
    while not est_vide(Q1):
        x = depiler(Q1)
        empiler(Q2, x)
    while not est_vide(Q2):
        x = depiler(Q2)
        empiler(P, x)
```

4

```
def tri_crepes(P):
    N = hauteur_pile(P)
    i = N
    while i > 1:
        j = max_pile(P,i)
        retourner(P,j)
        retourner(P,i)
        i = i - 1
```

## **Exercice 2**

1.1

Le chemin comprend 2 déplacements vers le bas

1.2

Sachant que les déplacements en diagonale ne sont pas autorisés, il faudra obligatoirement se déplacer 3 fois vers la droite (parcours 4 cases) et 2 fois vers le bas (parcours 2 cases supplémentaires) quel que soit l'ordre de ces déplacements. On aura donc bien un chemin de longueur égale à 6 quel que soit le chemin emprunté.

2

avec le parcours 0,0 -> 1,0 -> 2,0 -> 2,1 -> 2,2 -> 2,3  
on obtient la somme  $4 + 2 + 3 + 1 + 5 + 1 = 16$  qui est la somme maximale.

### 3.1

4	5	6	<b>9</b>
6	<b>6</b>	8	10
9	10	<b>15</b>	16

### 3.2

La somme obtenue à la colonne  $j$  est égale à la somme obtenue à la colonne  $j-1$  (à gauche de  $j$ ) plus la valeur de la case  $0, j$  (puisque l'on peut uniquement aller à droite)

$$d'o\grave{u} : T'[0][j] = T[0][j] + T'[0][j-1]$$

### 4

Quand on se trouve à la case  $(i, j)$ , on vient soit de la case  $(i-1, j)$  (case située au-dessus de  $(i, j)$ ), soit de la case  $(i, j-1)$  (case située à gauche de  $(i, j)$ ). Donc on doit ajouter à la valeur de la case  $T[i][j]$  soit la somme obtenue à la case  $(i-1, j)$ , soit la somme obtenue à la case  $(i, j-1)$  (on prendra la somme maximum).

$$d'o\grave{u} : T'[i][j] = T[i][j] + \max(T'[i-1][j], T'[i][j-1])$$

### 5.1

Le cas de base est le cas où  $i = 0$  et  $j = 0$ , on renvoie alors la valeur  $T[0][0]$

### 5.2

```
def somme_max(T,i,j):
    if i==0 and j==0:
        return T[0][0]
    else :
        if i==0:
            return T[0][j]+somme_max(T,0,j-1)
        elif j==0:
            return T[i][0]+somme_max(T,i-1,0)
        else:
            return T[i][j]+max(somme_max(T,i-1,j), somme_max(T,i,j-1))
```

### 5.3

Pour résoudre le problème initial, on doit effectuer l'appel suivant : `somme_max(T, 2, 3)`



### **Exercice 3**

1

taille = 9 ; hauteur = 4

2.1

G : 1010

2.2

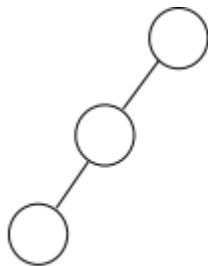
noeud l

2.3

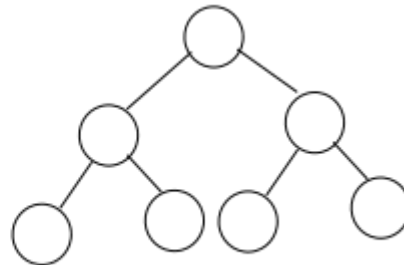
À chaque "étage", on augmente le nombre de bits de 1 : si  $h = 1$ , nombre de bit = 1 ; si  $h = 2$ , nombre de bits = 2... pour une hauteur  $h$  le nombre de bits est de  $h$ .

2.4

Prenons un exemple avec  $h = 3$  : nous avons 2 cas extrêmes : un arbre filiforme ou un arbre complet. Toutes les autres possibilités sont des cas intermédiaires.



arbre filiforme



arbre complet

Dans le cas de l'arbre filiforme nous avons, pour  $h = 3$ ,  $n = 3$  ( $n$  : taille). Si on généralise pour un arbre de hauteur  $h$ , nous avons  $n = h$

Dans le cas d'un arbre complet, pour  $h = 3$  nous avons  $n = 7$ , donc  $n = 2^3 - 1 = 7$ . Si on généralise pour un arbre de hauteur  $h$ , nous avons  $n = 2^h - 1$

Sachant qu'un arbre quelconque est un intermédiaire entre l'arbre filiforme et l'arbre complet, nous pouvons donc dire que :

$$h \leq n \leq 2^h - 1$$

3.1

[15, 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O']

3.2

$\frac{i}{2}$  si  $i$  est pair et  $\frac{i-1}{2}$  si  $i$  est impair

4

```
def recherche(arbre, element):
    n = arbre[0]
    i = 1
    while i <= n:
        if arbre[i] == element:
            return True
        elif element > arbre[i]:
            i = 2*i+1
        else :
            i = 2*i
    return False
```

#### **Exercice 4**

ATTENTION : Il y a une erreur dans l'énoncé : num\_eleve ne peut pas être en entier car il est de la forme 133310FE (ou alors, c'est un entier en base 16 ;-))

1.1

num\_eleve va jouer le rôle de clé primaire. La clé primaire permet d'identifier de manière unique un t-uplet de la relation seconde.

1.2

```
INSERT INTO seconde
(num_eleve, langue1, langue2, option, classe)
VALUES
('133310FE', 'anglais', 'espagnol', '0', '2A')
```

on a choisi de mettre 0 pour l'option quand l'élève n'a pas d'option

1.3

```
UPDATE seconde
SET langue1 = 'allemand'
WHERE num_eleve = '156929JJ'
```

2.1

Cette requête donne tous les num\_eleve contenus dans la table seconde

2.2

Le résultat est 30 car il y a 30 entrées

## 2.3

```
SELECT COUNT(num_eleve)
FROM   seconde
WHERE  langue1 = 'allemand' OR langue2 = 'allemand'
```

## 3.1

La clé étrangère permet de créer un lien entre la table *eleve* et la table *seconde*. Ce lien est unique pour chaque entrée. Pour préserver l'intégrité de la base de données : pour que toutes les valeurs de la clé étrangère correspondent bien à des valeurs présentes dans la clé primaire.

## 3.2

```
SELECT nom, prenom, datenaissance
FROM   eleve
INNER JOIN seconde ON eleve.num_eleve = seconde.num_eleve
WHERE  classe = '2A'
```

## 4

coordonnees
num_eleve (clé primaire, clé étrangère des tables <i>seconde</i> et <i>eleve</i> )
adresse
code_postal
ville
adresse_email

## **Exercice 5**

1.1

A -> C -> F -> G

1.2

Destination	Routeur suivant	Distance
A	F	3
B	E	3
C	F	2
D	E	2
E	E	1
F	F	1

2

Destination	Routeur suivant	Distance
B	B	1
D	D	1
E	D	2
F	D	4
G	D	3

3.1

A -> B : 10 Gb/s soit le coût =  $\frac{10^8}{10 \cdot 10^9} = 0,01$

3.2

$$\frac{10^8}{d} = 5 \Rightarrow d = \frac{10^8}{5} = 2 \cdot 10^7 \text{ b/s} = 20 \text{ Mb/s}$$

4

On part de A (possible d'aller en B, en D ou en C), on trouve le débit le plus important au niveau de la liaison A->D. Une fois en D, on va vers E et une fois en E, on rejoint directement G (car le coût du chemin E -> C -> F -> G serait plus grand).

d'où le chemin : A -> D -> E -> G avec un coût =  $\frac{10^8}{10 \cdot 10^9} + \frac{10^8}{100 \cdot 10^9} + \frac{10^8}{100 \cdot 10^6} = 1,02$