DS5

Durée : 4 heures

L'utilisation de la calculatrice **n'est pas autorisée** pour cette épreuve.

Ce sujet est composé de 5 parties indépendantes.

1 Quelques preuves de déduction

Construire, à l'aide du système de déduction naturelle, les arbres de dérivation correspondant aux séquents suivants :

1. $\neg(\varphi \vee \psi) \vdash \neg\varphi \wedge \neg\psi$
2. $\neg\varphi \vee \psi \vdash \varphi \Rightarrow \psi$
3. $\varphi \vdash \neg\neg\varphi$
4. $\exists x\varphi \vdash \neg\forall x\neg\varphi$

2 Arbres de décision (la programmation sera à effectuer en C)

On considère le jeu de données suivant :

A	B	C	V
0	0	0	0
0	1	0	1
1	0	0	1
1	1	0	0
1	1	1	0

On souhaite construire un arbre de décision pour classifier la variable V en fonction des valeurs de A, B et C.

On note V_0 (resp. V_1) les ensembles de lignes pour lesquels la valeur de V est 0 (resp 1).

On rappelle que l'entropie est définie par :

$$H(V) = -\frac{|V_0|}{|V|}\ln\left(\frac{|V_0|}{|V|}\right) - \frac{|V_1|}{|V|}\ln\left(\frac{|V_1|}{|V|}\right)$$

et que le gain de V par rapport à un critère quelconque D est défini par :

$$G(V|D) = H(V) - \left(\frac{|D_0|}{|V|}H(V|D=0) + \frac{|D_1|}{|V|}H(V|D=1)\right)$$

1. Donner les formules numériques (sans les calculer) pour $H(V), H(V|A=0), H(V|A=1)$ et $H(V|C=1)$.
2. On admettra ici que $H(V) = 0,673$ et que $G(V|A) = 0,0142$. Déterminer $G(V|B)$ et $G(V|C)$ (on admettra que $ln(2) = 0,693$ et on rappelle que pour diviser par 5 il suffit de multiplier par 2 et diviser par 10).
3. En déduire un arbre de décision qui serait obtenu en appliquant l'algorithme ID3 à ce jeu de données.

L'objectif maintenant est de mettre en oeuvre une fonction de backtracking en C qui permettra de calculer un arbre de décision de hauteur minimale qui classifie parfaitement un jeu de données.

On dit qu'un arbre de décision classifie parfaitement un jeu de données si toutes les données aboutissant dans une feuille ont la même valeur, un tel arbre n'existe pas pour tout jeu de données mais on supposera dans le cas de notre algorithme qu'il en existe bien pour le jeu de données passé en argument.

Pour faciliter le stockage des données, les attributs sont notés de 0 à $k-2$ quand il y a $k-1$ attributs et la variable que l'on cherche à classifier est nommée $k-1$. Le jeu de données est stocké dans un tableau de booléens mettant bout à bout tous les points des données. Par exemple, le jeu de données étudié ci-dessus est représenté par :

```
bool data[]={false,false,false,false,
             false,true,false,true,
             true,false,false,true,
             true,true,false,false,
             true,true,true,false};
```

Ainsi, un jeu de données à n points sera représenté par un tableau unidimensionnel de taille $n \times k$.

Au fur et à mesure de l'exécution, pendant la construction de l'arbre, nous allons considérer seulement des sous-ensembles de points de données, ou des sous ensembles de la liste des attributs. Pour ce faire, nous utiliserons deux tableaux de booléens `bool attr_actif[]` de taille k et `bool data_actif[]` de taille n.

4. Ecrire une fonction `bool non_vide(bool data[], bool data_actif[], int n, int k)` qui prend en argument un ensemble de n données ayant $k-1$ attributs, et qui renvoie `false` ssi aucun point n'est actif et `true` sinon.
5. Ecrire une fonction `bool tous_faux(bool data[], bool data_actif[], int n int k)` qui prend en argument un ensemble de n données ayant $k-1$ attributs et qui renvoie `true` ssi tous les points actifs ont pour valeur `false` pour la variable $k-1$ que l'on cherche à classifier.
6. Ecrire une fonction `bool *filtre(bool data[], bool data_actif[], int attr, bool valeur, int n, int` qui prend en argument un ensemble de n données ayant $k-1$ attributs et qui crée une copie du tableau `data_actif` en retirant (mettant à `false`) les points dont l'attribut `attr` n'a pas la valeur `valeur`.

On suppose que l'on dispose d'une fontion `tous_vrais` analogue à la fonction `tous_faux`.

On utilise la structure suivante pour définir un arbre de décision :

```
struct arbre { int test; struct arbre* left; struct arbre* right;};

typedef struct arbre arbre_t;
```

Le champs `test` correspond au numéro de l'attribut considéré, le champ `left` correspond au sous-arbre composé des éléments dont la valeur est négative (`false`) et le champ `right` correspond au sous-arbre composé des éléments dont la valeur est positive. Les feuilles, sont représentées en utilisant une valeur

particulière pour `test` : on utilise -1 pour une classification positive et -2 pour une classification négative. Les champs `left` et `right` ont des valeurs `NULL` pour une feuille. On vous fournit les fonctions suivantes pour créer de nouvelles feuilles :

```
arbre_t* feuille_zero(){
    arbre_t* a = malloc(sizeof(arbre_t));
    a->test=-2;
    a->left=NULL;
    a->right=NULL;
    return a;
}

arbre_t* feuille_un(){
    arbre_t* a = malloc(sizeof(arbre_t));
    a->test=-1;
    a->left=NULL;
    a->right=NULL;
    return a;
}
```

7. Ecrire une fonction `int depth(arbre_t *a)` qui calcule la profondeur d'un arbre.
8. Ecrire une fonction `void libere(arbre_t *a)` qui libère l'espace alloué pour un arbre.
9. On est maintenant en mesure d'écrire notre fonction de backtracking, voici une propostion pour laquelle vous devez compléter les cas de bases : indiquer ce qu'il faut mettre dans les lignes 5 et 8.

```
1  arbre_t* backtrack_ss(bool data[], bool data_actif[], bool attr_actif[], int n, int k){
2    if (non_vide(data,data_actif,n,k)){
3
4        if (tous_faux(data,data_actif, n,k)){
5
6        }
7        else if (tous_vrais(data,data_actif,n,k)){
8
9        }
10       else {int meilleur = MAX_INT;
11            arbre_t* left=NULL; arbre_t* right=NULL;
12            int meilleur_critere=-1;
13
14            for (int c=0;c<k-1;c++){
15                if (attr_actif[c]){
16                attr_actif[c]=false;
17
18                bool* dataplus = filtre(data,data_actif,c,true,n,k);
19                arbre_t* aplus = backtrack(data,dataplus,attr_actif,n,k);
20                free(dataplus);
21                int hplus= depth(aplus);
22
23                bool* datamoins = filtre(data,data_actif,c,false,n,k);
24                arbre_t* amoins = backtrack(data,datamoins,attr_actif,n,k);
25                free(datamoins);
26                int hmoins= depth(amoins);
27
28                if (1+max(hplus,hmoins)< meilleur){
29
30                    left=amoins;
31                    right=aplus;
32                    meilleur_critere=c;
33                }
34                else {                          }
35                attr_actif[c]=true;
36                }
37            }

38        arbre_t* a = malloc(sizeof(arbre_t));
39        a->test=meilleur_critere;
40        a->left=left;
41        a->right=right;
42        return a;
43        }
44
45    }
46
47    else {return NULL;}
48 }
```

10. Il y a des fuites mémoire dans la solution proposée, corrigez ce problème. On pourra se contenter d'indiquer les lignes où on ajoute quelque chose sans tout recopier et en s'appouyant sur la numérotation du code.

11. La solution proposée n'est pas vraiment une solution de backtracking, expliquer pourquoi? Proposer une stratégie d'élagage et écrire une solution finale dans laquelle on adopte une stratégie de backtracking.

3 Caractérisation des programmes bien typés

Le langage OCAML utilise un système d'inférence pour vérifier que les expressions qu'on lui donne sont bien typées. Nous allons, dans cette partie, manipuler ce système relatif à un petit fragment du langage OCAML.

Nous allons établir une relation entre les expressions bien typées (e) et les types (τ) que nous noterons $\vdash e : \tau$. Par exemple, on aura $\vdash 12 : int$ ou encore $\vdash true : bool$. Ce type de relation sera appelé un jugement.

1. Donner le jugement que l'on obtiendrait avec l'expression `List.map (fun x->x-1)`.

   La présentation donnée ici est incomplète car si on considère une fonction `fun x -> e` de type `'a -> 'b` alors le type de l'expression `e` sera bien `'b` à condition que l'on ait bien pris l'argument `x` de type `'a`. Ainsi, on aura besoin d'avoir un contexte Γ qui contiendra les types des variables qui sont susceptibles d'être présentes dans une des expressions.

   Nous écrirons finalement $\Gamma \vdash e : \tau$. Par exemple :

   $$\{x : \text{int}, l : \text{int list}, f : \text{int} \to \text{int}\} \vdash ((f\ x) :: l) : \text{int list}$$

   Le système d'inférence utilise alors les axiomes suivants :

   (a) Pour toutes les constantes du langage on dispose d'un axiome le reliant à son type. Par exemple, $\Gamma \vdash 42 : int$.

   (b) Pour toute variable x contenue dans Γ, si son type associé par Γ est τ alors $\Gamma \vdash x : \tau$. Par exemple : $\{x : \text{int}, l : \text{int list}\} \vdash x : \text{int}$

   (c)
   $$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (\text{fun } x \to e) : \sigma \to \tau}.$$

   (d)
   $$\frac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f\ e : \tau}.$$

   Pour rendre vos preuves lisibles, vous indiquerez à côté de chaque étape de dérivation, la règle (a,b,c ou d) utilisée.

2. On considère l'environnement $\Gamma = \{f : \alpha \to (\beta \to \gamma), g : \beta \to \alpha\}$ où α, β et γ sont des types arbitraires. Montrer à l'aide du système décrit ci-dessus que : $\Gamma, x : \beta \vdash g\ x : \alpha$.

3. En déduire, dans le même environnement que : $\Gamma \vdash (\text{fun } x \to f\ (g\ x)\ x) : \beta \to \gamma$.

4. Considérons maintenant l'expression $e = (\text{fun } h \to h\ 1\ 2)(\text{fun } x \to 3)$. Nous allons raisonner par l'absurde pour montrer que cette expression n'est pas typable. Supposons donc que l'on a réussi à dériver $\vdash e : \alpha$. On pourra utiliser le fait que le type `int` ne peut pas être décomposé comme un type de la forme $\alpha \to \beta$.

   (a) Quelle est la seule règle qui a pu être utilisée en dernier pour obtenir ce jugement ? On remarque alors qu'il est nécessaire de fournir des dérivations pour les jugements : $\vdash \text{fun } x \to 3 : \beta$ et $\vdash \text{fun } h \to h\ 1\ 2 : \beta \to \alpha$ pour un certain type β.

   (b) Considérons d'abord le jugement $\vdash (\text{fun } x \to 3) : \beta$. De quelle forme doit alors être le type β?

   (c) En procédant par conditions nécessaires sur le schéma de dérivation qui permettrait d'obtenir $\vdash (\text{fun } h \to h\ 1\ 2) : \beta \to \alpha$, montrer que nécessairement, $\beta = int \to (int \to \alpha)$.

   (d) Conclure.

   Etendons notre système pour autoriser la manipulation des couples à l'aide de la règle, nommée (couple), suivante :

$$\frac{\Gamma \vdash e_1 : \tau_1; \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}.$$

   Considérons les fonctions associées aux manipulations de couples `fst : 'a* 'b -> 'b` et `snd : 'a* 'b->'b`

5. Rappeler ce que font ces deux fonctions. Parmi les expressions Ocaml suivantes, lesquelles sont correctement typées ? Expliquer. Quand l'expression est syntaxiquement correcte, donner son type et sa valeur.

   (a) `fst (3,4)`

   (b) `fst 3`

   (c) `snd (true, false)`

   (d) `snd (true, [])`

   (e) `fst (fst ([1;2],[3;4]))`

6. Les fonctions `fst` et `snd` sont dites polymorphes. Rappeler le sens de ce terme et donner un (ou plusieurs) exemple(s) concret(s) permettant de mettre en évidence cette propriété.

   Ainsi, à partir de maintenant on pourra utiliser dans notre système d'inférence ces deux fonctions comme des constantes dont les types respectifs sont les suivants :

$$fst : \forall\alpha\forall\beta, (\alpha * \beta) \to \alpha \text{ et } snd : \forall\alpha\forall\beta, (\alpha * \beta) \to \beta$$

   Nous allons donc ajouter une règle d'inférence, nommée (poly), permettant de manipuler des types contenant un quantificateur universel :

$$\frac{\Gamma \vdash e : \forall\alpha\tau}{\Gamma \vdash e : \tau[\alpha \to \sigma]}.$$

   où l'on peut choisir le type souhaité pour valeur de σ et où $\tau[\alpha \to \sigma]$ désigne la formule τ où on substitue σ à chaque occurence de α.

7. Ecrire une preuve de $\vdash (\text{ fst } (42, false)) : int$.

4 Séquents valides en calcul des prédicats

Toutes les fonctions de ce problème seront implémentées en utilisant le langage Ocaml.

Dans cet exercice, on se place dans le contexte du calcul des prédicats. La première partie de l'énoncé introduit une façon de manipuler des formules du premier ordre en Ocaml en toute généralité. La seconde étudie la validité de séquents dans un langage du premier ordre fixé.

Partie 1 *Formules bien formées*

Pour manipuler des formules sur un langage du premier ordre, on introduit les types polymorphes suivants pour représenter les symboles de fonction, les symboles de relation et les termes. On introduit également un type spécifique pour les connecteurs binaires afin d'alléger l'écriture des fonctions à venir.

```
type 'f fonction = 'f*int
type 'r relation = 'r*int
type ('f,'v) terme = V of 'v | F of 'f fonction * ('f,'v) terme list
type op_binaire = Et | Ou | Implique | Equivalent
```

Les fonctions sont définies comme étant des couples : le premier élément est leur symbole, de type 'f, et le second est leur arité. La chose est similaire pour les relations. Par exemple,

```
let ex = F(('f',1),[V(4)])
```

est de type (char,int) terme et représente le terme $f(4)$. Autrement dit, les symboles de variables sont de type int et les symboles de fonction sont de type char. Attention, dans l'expression ci-dessus, 4 est bien un symbole dénotant une variable et pas une constante.

1. Déterminer le type du terme t défini ci-dessous et indiquer quel terme il représente :

   ```
   let t = F(("plus",2), [V('x'); F(("cos",1),[V('y')])])
   ```

Le type définissant les termes ne peut pas vérifier que les arités des fonctions utilisées sont bien respectées, il est donc possible d'introduire un terme mal formé. Pour éviter cela :

2. Ecrire une fonction récursive terme_bien_forme de signature ('a,'b) terme -> bool renvoyant true si le terme en entrée respecte les arités des symboles de fonctions et false sinon.

On introduit à présent un type polymorphe pour manipuler des formules du premier ordre dont les variables sont de type 'v, les symboles de fonction de type 'f et ceux de relation de type 'r :

```
type ('f,'r,'v) formule =
  R of 'r relation * ('f,'v) terme list
|Forall of 'v * ('f,'r,'v) formule
|Exists of 'v * ('f,'r,'v) formule
|Non of ('f,'r,'v) formule
|Op_bin of op_binaire * ('f,'r,'v) formule * ('f,'r,'v) formule
```

3. Déclarer une variable f de type (string, string, char) formule représentant la formule

$$\forall x\, x + \cos(y) = x + \cos(y)$$

4. Comme pour les termes, le type utilisé pour les formules n'interdit pas d'écrire une formule ne respectant pas les arités de relations. Ecrire une fonction formule_bien_formee de signature ('a,'b,'c)

formule -> bool indiquant si la formule en entrée est bien formée vis-à-vis de **toutes** les arités.

Dans la suite, les formules manipulées seront systématiquement syntaxiquement correctes et il ne sera pas nécessaire de vérifier que c'est le cas.

5. Ecrire une fonction apparait de signature 'a -> ('b,'a) terme -> bool telle que apparait x t indique si la variable x est présente dans le terme t.

6. Ecrire une fonction est_libre de signature 'a -> ('b,'c',a) formule -> bool telle que est_libre x f renvoie true si et seulement si une des occurrences de x est libre dans f.

7. En déduire une fonction est_close de signature ('a,'b,'c) formule -> 'c list -> bool prenant en entrée une formule f et une liste lv de variables qu'on supposera être la liste des variables intervenant dans f et indiquant si la formule f est close. Justifier brièvement sa correction.

Partie 2 *Sémantique et déduction sur un langage du premier ordre*

Dans cette partie on considère le langage du premier ordre L dont la signature est la suivante :

$$\mathcal{F} = \emptyset \text{ et } \mathcal{R} = \{=: 2, R : 2\}$$

Sur ce langage, on se donne les trois formules closes suivantes :

- $F_1 = \forall x\, R(x, x)$
- $F_2 = \forall x \forall y\, (R(x, y) \wedge R(y, x) \Rightarrow x = y)$
- $F_3 = \forall x \forall y \forall z\, (R(x, y) \wedge R(y, z) \Rightarrow R(x, z))$

On note $F = F_1 \wedge F_2 \wedge F_3$.

8. La structure $\mathcal{M}$ dont le domaine est $\mathbb{R}$ et dans laquelle la relation R est interprétée par l'inégalité stricte $<$ (la relation $=$ étant toujours interprétée comme étant l'égalité) est-elle un modèle pour F ? Si oui, justifier, si non, exhiber un modèle pour F.

9. Indiquer pour chacun des séquents $\Gamma \vdash G$ sur L suivants s'il est valide ou non. Si $\Gamma \vdash G$ est valide, en donner une preuve syntaxique en déduction naturelle. Si $\Gamma \vdash G$ n'est pas valide, donner une preuve sémantique de ce fait en exhibant une structure qui contredit $\Gamma \models G$.

   a) $\vdash \neg F_1 \wedge \neg F_2 \Rightarrow \neg(F_1 \vee F_2)$. Dire sans justifier si l'implication réciproque est vraie.

   b) $F \vdash \forall x \forall y\, (R(x, y) \vee R(y, x))$.

   c) $F \vdash \forall x \exists y\, R(x, y)$.

   d) $F \vdash \forall x \exists y\, (R(x, y) \wedge \neg(x = y))$.

   e) $F \vdash \exists x \forall y\, R(y, x)$

5 Clusturing en dimension 1

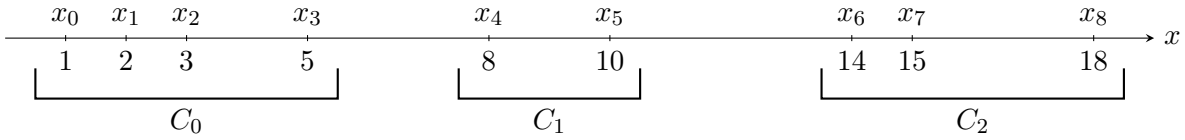
On souhaite mettre en place du soutien différencié au travail dans une classe de lycée. Pour ce faire, on souhaite séparer les élèves en plusieurs groupes de niveaux homogènes pour leur proposer des exercices adaptés à leur niveau. On dispose pour cela des notes des élèves et on veut les regrouper selon la similitude de leurs notes.

Formellement, on dispose d'un ensemble E de N réels $x_0 \leq x_1 \leq \ldots \leq x_{N-1}$. On cherche à déterminer une partition de $[\![0, N-1]\!]$ en $K \leq N$ sous-ensembles $\mathcal{P} = \{C_0, C_1, \ldots, C_{K-1}\}$ non vides tels que le score

$$S(\mathcal{P}) = \sum_{i=0}^{K-1} \sum_{j \in C_i} (x_j - \mu_i)^2$$

soit minimal, où $\mu_i = \dfrac{1}{|C_i|} \sum\limits_{j \in C_i} x_j$ est la moyenne des éléments correspondant à la classe C_i. Autrement dit, on veut minimiser la somme des carrés des écarts de chaque élément à la moyenne de sa classe.

Par exemple, pour $E = \{1, 2, 3, 5, 8, 10, 14, 15, 18\}$, une solution optimale pour $K = 3$ est donnée par la partition suivante :



Si $\{C_0, C_1, \ldots, C_{K-1}\}$ est une partition solution du problème, on remarque qu'on peut supposer sans perte de généralité que les classes sont rangées par ordre croissant, c'est-à-dire que pour $i < i'$ et $j \in C_i$, $j' \in C_{i'}$, alors $x_j \leq x_{j'}$. On supposera cette hypothèse vérifiée pour l'ensemble des partitions du problème.

5.1 Préliminaires

1. Comment trouver une solution au problème lorsque $K = N$? Lorsque $K = N - 1$? Justifier.
2. Appliquer l'algorithme des K-moyennes sur l'ensemble E de l'exemple précédent. On prendra $K = 3$, et on initialisera les barycentres à $b_0 = 1$, $b_1 = 8$ et $b_2 = 10$. On donnera les détails des étapes de calculs jusqu'à convergence de l'algorithme.

   On accepte (et on souhaite) une représentation graphique pour la description des étapes de calcul.

   On cherche à écrire une fonction `void tri(double* tab, int n)` qui trie un tableau `tab` de taille `n` en C. Pour ce faire, on propose les fonctions suivantes :

```
void fusion(double* tab1, int n1, double* tab2, int n2, double* tab){
    int i1 = 0;
    int i2 = 0;
    for (int i=0; i<n1 + n2; i++){
        if (tab1[i1] <= tab2[i2]){
            tab[i] = tab1[i1];
            i1++;
        } else {
            tab[i] = tab2[i2];
        }
    }
}

void tri(double* tab, int n){
    if (n < 1) return;
    int n1 = n / 2;
    int n2 = n - n / 2;
    double* tab1 = malloc(n1 * sizeof(double));
    double* tab2 = malloc(n2 * sizeof(double));
    for (int i=0; i<n; i++){
        if (i < n1) tab1[i] = tab[i];
        else tab2[i - n1] = tab[i];
    }
    tri(tab1, n1);
    tri(tab2, n2);
    fusion(tab1, n1, tab2, n2, tab);
}
```

3. Le code précédent comporte plusieurs (moins de 5) erreurs. Préciser les lignes où apparaissent ces erreurs et un moyen de les corriger. On demande de **NE PAS** recopier tout le code.

Pour la suite de cette partie, on supposera que le tableau `E` représentant un ensemble E sera toujours trié.

Pour $E = \{x_0, \dots, x_{N-1}\}$ et $0 \leq i < j \leq N$, on note $S_{\text{emc}}(i, j)$ (pour « somme des écarts à la moyenne au carré ») la valeur

$$S_{\text{emc}}(i, j) = \sum_{k=i}^{j-1} (x_k - \mu)^2$$

où μ est la moyenne des éléments de $\{x_i, x_{i+1}, \dots, x_{j-1}\}$.

4. Écrire une fonction `double moyenne(double* E, int i, int j)` qui prend en argument un tableau E de N valeurs $\{x_0, \dots, x_{N-1}\}$ triées et deux indices $0 \leq i < j \leq N$ et renvoie la moyenne des éléments de $\{x_i, x_{i+1}, \dots, x_{j-1}\}$.

5. Écrire une fonction `double somme_emc(double* E, int i, int j)` qui prend en argument une liste E de N valeurs triées et deux indices $0 \leq i < j \leq N$ et calcule et renvoie $S_{\text{emc}}(i, j)$.

On représente une partition $\mathcal{P} = \{C_0, C_1, \ldots, C_{K-1}\}$ de $[\![0, N-1]\!]$ par un tableau `P` de taille K telle que `P[i]` est le plus petit élément de C_i. Avec l'hypothèse faite précédemment sur les C_i, le tableau `P` doit nécessairement être trié. On remarque, de plus, que `P[0]` vaut toujours 0. Par exemple, la partition $\{\{0,1,2,3\},\{4,5\},\{6,7,8\}\}$ donnée dans l'exemple de la figure 1 est représentée par `{0, 4, 6}`.

6. Dans cette question, on suppose $N = 9$. Quelle est le tableau `P` associé à la partition $\mathcal{P} = \{\{0,1,2\},\{3,4\},\{5,6,7\},\{8\}\}$? Quelle est la partition associée au tableau `{0, 1, 4, 5}` ?

7. Écrire une fonction `double score(double* E, int N, int* P, int K)` qui prend en argument un ensemble `E` de taille N et une partition `P` de $[\![0, N-1]\!]$ de taille K et renvoie le score $S(\mathcal{P})$ tel qu'il a été défini précédemment.

8. En déduire une fonction `int* clustering3(double* E, int N)` qui prend en argument un ensemble `E` de taille $N \geq 3$ et renvoie une partition `P` de $[\![0, N-1]\!]$ de taille $K = 3$ de score minimal.

9. Quelle est la complexité temporelle de la fonction précédente ? Justifier.

5.2 Clustering hiérarchique ascendant

On cherche dans cette sous-partie à calculer une solution pas nécessairement optimale par une approche de clustering hiérarchique ascendant (CHA).

10. Écrire une fonction `int classes_plus_proches(double* E, int N, int* P, int K)` qui prend en argument un ensemble `E` de taille N et une partition `P` de $[\![0, N-1]\!]$ de taille $K > 1$ et renvoie un indice i_{opt} tel que $C_{i_{\text{opt}}}$ et $C_{i_{\text{opt}}+1}$ sont les classes les plus proches, c'est-à-dire celles qui ont leurs moyennes les plus proches. En cas d'égalité, on choisira l'indice i_{opt} minimal.

11. Écrire une fonction `int* fusion_classes(int* P, int K, int iopt)` qui prend en argument une partition `P` de $[\![0, N-1]\!]$ de taille K et un indice $0 \leq i_{\text{opt}} < K-1$ et renvoie une partition de $[\![0, N-1]\!]$ de taille $K-1$ où les classes d'indices i_{opt} et $i_{\text{opt}}+1$ ont été fusionnées.

12. En déduire une fonction `int* CHA(double* E, int N, int K)` qui calcule et renvoie une partition de taille K d'un ensemble E selon l'algorithme de clustering hiérarchique ascendant.

13. Déterminer la complexité temporelle de la fonction `CHA` en fonction de N et de K.

5.3 Solution optimale en programmation dynamique

Pour $n \in [\![0, N]\!]$ et $k \in [\![1, K]\!]$, on note $D(n,k)$ le score minimal possible d'une partition de $\{x_0, \ldots, x_{n-1}\}$ en k classes non vides.

14. Que vaut $D(n,k)$ lorsque $k = 1$?

15. Montrer que pour $n > 0$ et $k > 1$, $D(n,k) = \min_{i=k-1}^{n-1} (D(i, k-1) + S_{\text{emc}}(i,n))$.

16. En déduire une fonction `double clustering_dynamique(double* E, int N, int K)` qui calcule le score minimal possible d'une partition de E en K classes non vides.

17. Déterminer la complexité temporelle de la fonction précédente.

18. Expliquer en français comment modifier la fonction précédente pour qu'elle renvoie une partition optimale plutôt que le score minimal. On ne demande pas de coder cette solution.

⋆ ⋆ ⋆