# Devoir maison n°1 - À rendre le 03 Novembre 2024
# Recherche des $k$ plus proches voisins et arbres $k$-dim

**Notions abordées**

- algorithme de recherche des $k$-plus proches voisins
- révisions de C : allocation dynammique de mémoire, tableaux, struct
- listes simplement chaînées en C
- sélection rapide d'un élément de rang fixé dans une liste
- construction récursive d'un arbre $k$-dimensionnel
- recherche des plus proches voisins dans un arbre $k$-dimensionnel

*La première partie de ce devoir, à programmer en C, est l'occasion de revoir la compilation séparée et l'utilisation de* Makefile. *Ni la rédaction ni l'utilisation de* Makefile *ne sont au programme, mais leur utilisation vous facilitera la gestion des multiples compilations. Vous trouverez sur cahier de prépa une archive contenant un squelette du code, c'est-à-dire un lot de fichiers à remplir et le* Makefile *associé, ainsi qu'un fichier* `README.md` *contenant des précisions utiles sur l'organisation de ces fichiers. En particulier les jeux de tests concernant tout ce qui a été codé dans la section* `i` *seront rassemblés dans un fichier nommé* `tests_section_i.c`.

## Exercice 1 : $k$ plus proches voisins en C

Dans ce DM de programmation, on s'intéresse à l'implémentation en C, de l'algorithme des $k$-plus proches voisins. La figure 1 ci-dessous ♣ représente un extrait de la base de données de `MNIST`, qui contient des images de chiffres entre 0 et 9 manuscrits associées au chiffre représenté dans l'image, autrement dit c'est un jeu de données déjà classifiées.

FIGURE 1 – Extrait de la base `MNIST`

♣. merci Wikipédia

Dans ce projet nous nous proposons de construire, en apprenant sur les données de la base `MNIST`, un classifieur capable de déterminer quel chiffre est écrit sur une image, autrement dit capable de classifier des images de chiffre manuscrit. Les images considérées seront toutes au même format utilisé dans la base `MNIST`, à savoir des images de $28\times28$ pixels en niveaux de gris codés sur un octet. Une telle image est donc encodée par un tableau d'entiers de $[\![0, 255]\!]$ de dimension $784(28 \times 28)$. Cette base de données est à télécharger sur github : `https://github.com/halimb/MNIST-txt`. Elle sera nécessaire pour réaliser la section 6.

## 1. Définitions de types pour les données

**Le type `vector`.** Afin de représenter les images comme des tableaux munis de leur taille, nous définissons un type structuré `vector_s`, ainsi qu'un type `vector` comme étant un pointeur vers une telle structure.

```
struct vector_s {
  int            taille  ;      /* taille du tableau content */
  unsigned char* content ;      /* tableau d'éléments de [[0, 255]] */
};
typedef struct vector_s * vector;
```

**Q. 1** Définir la fonction `vector create_zero_vector (int n)` retournant un vecteur de taille `n` nouvellement alloué, dont toutes les coordonnées sont initialisées à $0$.

**Q. 2** Définir la fonction `void delete_vector (vector v)` qui libère l'espace mémoire occupé par le vecteur qu'elle prend en argument.

**Q. 3** Définir une fonction d'affichage des vecteurs nommée `print_vector`. Par exemple l'affichage du vecteur vide pourra afficher `()`, celui de `{0, 1, 2}` pourra afficher `(0, 1, 2)`

**Q. 4** Définir une fonction `double distance(vector u, vector v)` permettant de calculer la distance euclidienne entre deux vecteurs de même taille. On rappelle que l'inclusion de la bibliothèque `math.h` permet l'utilisation de la fonction `sqrt` mais nécessite de passer l'option `-lm` à la commande de compilation `gcc`.

**Le type `database`.** Une donnée classifiée, est la donnée d'un couple (donnée, classe). Dans notre cas les classes sont des entiers de l'intervalle $[\![0, 9]\!]$. On définit donc un type `classified_data` permettant de représenter de tels couples. L'algorithme de classification travaillera ensuite sur un jeu de données, qui sera représenté par un tableau de données classifiées accompagné de sa taille, soit le type `database` défini ci-dessous.

```
struct classified_data_s {
  vector vector ;                   /* le vecteur */
  int    class  ;                   /* sa classe */
};
struct database_s {
  int size ;                        /* taille du jeu de données */
  struct classified_data_s* datas ; /* tableau des données classifiées */
};
typedef struct database_s * database;
```

**Q. 5** Définir la fonction `database create_empty_database (int n)` qui alloue la mémoire pour une base de données de taille `n`, initialement vide, et la retourne.

**Q. 6** Définir la fonction `void delete_database (database b)` qui libère l'espace mémoire occupé par la base de données qu'elle prend en argument.

**Q. 7** Définir une fonction nommée `print_data_base` d'affichage des jeux de données. Par exemple l'affichage du jeu de données associant la classe $1$ au vecteur `{0, 1, 2}` et la classe $0$ au vecteur `{2, 5, 4}` pourra produire l'affichage ci-contre.

```
    {
      (0, 1, 2)
↪   ~> 1
      (2, 5, 4)
↪   ~> 0
    }
```

**Détecter les fuites mémoires en C avec Valgrind**

Valgrind est un petit utilitaire qu'on peut lancer depuis un terminal pour analyser un exécutable C, notamment pour analyser quel usage de la mémoire fait le programme, et détecter d'éventuelles fuites mémoire. Après avoir compilé le programme avec gcc avec l'option -g, on peut lancer une exécution en demandant à Valgrind d'analyser ce qui se passe.

```
gcc -g mon_programme.c -o mon_programme
valgrind -v --leak-check=summary ./mon_programme
```

Cette opération produit beaucoup d'affichages[a], mais la partie `HEAP SUMMARY` est particulièrement intéressante puisqu'elle résume l'utilisation du tas. Ainsi, sur un programme sans fuite mémoire on obtient par exemple :

```
==8112== HEAP SUMMARY:
==8112==     in use at exit: 0 bytes in 0 blocks
==8112==   total heap usage: 11 allocs, 11 frees, 1,164 bytes allocated
==8112==
==8112== All heap blocks were freed -- no leaks are possible
```

tandis que sur un programme où des zones mémoires n'ont pas été libérées on obtient par exemple :

```
==8175== HEAP SUMMARY:
==8175==     in use at exit: 140 bytes in 10 blocks
==8175==   total heap usage: 11 allocs, 1 frees, 1,164 bytes allocated
==8175==
==8175== Searching for pointers to 10 not-freed blocks
==8175== Checked 74,744 bytes
==8175==
==8175== LEAK SUMMARY:
==8175==    definitely lost: 32 bytes in 2 blocks
==8175==    indirectly lost: 108 bytes in 8 blocks
==8175==      possibly lost: 0 bytes in 0 blocks
==8175==    still reachable: 0 bytes in 0 blocks
==8175==         suppressed: 0 bytes in 0 blocks
```

**NB** : Pour l'installation, vous pouvez vous référez à la page suivante. `https://doc.ubuntu-fr.org/valgrind#installation`.

*a*. il est sûrement possible de réduire ces affichages avec des options bien choisies

## 2. Les $k$ plus proches voisins par listes

**Attention :** Cette section est mutuellement exclusive avec la section suivante et propose une implémentation de l'algorithme de recherche des plus proches voisins par listes chaînées plutôt que par tableaux. La présente section est à préférer à la suivante, mais la fonction d'insertion dans les listes est non triviale et demande une bonne compréhension des mécanismes de pointeurs en C (au programme de MP2I/MPI). ***Vous devez vous essayer à cette partie.*** *Si vous passez cette partie avec succès, vous pouvez ignorer la section 3 et passer directement à la section 4.*

**Description de l'algorithme.** Pour finalement obtenir la liste des $k$ plus proches voisins du vecteur `input`, on parcourt la liste des vecteurs de la base de données en maintenant au fur et à mesure une liste des meilleurs candidats, c'est-à-dire des vecteurs les plus proches de `input` déjà rencontrés. Plutôt que de stocker directement les vecteurs eux-mêmes (qui sont de taille 784 dans l'application proposée ici), on stocke leur indice dans la base de données. De plus, afin de pouvoir efficacement comparer un nouveau vecteur aux candidats sélectionnés auparavant, on munit chaque vecteur candidat de sa distance à `input`, et on range ces candidats par distance à `input` décroissante♣. De plus, on limite la capacité de cette liste à $k$ candidats, car on n'a pas besoin de plus dans le résultat♡.

**Exemple.** Pour la base de données $\left[\begin{pmatrix}3\\0\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}2\\4\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}0\\2\end{pmatrix}\right]$ et `input` $= \begin{pmatrix}1\\1\end{pmatrix}$, la recherche des $k=3$ plus proches voisins de `input` en suivant l'algorithme proposé ci-avant est décrite par tableau suivant, où les listes et structures sont représentées **à la OCaml** (*i.e.* `i` indique le champs `indice`, `d` indique le champs `distance`).

| j | r | candidats | commentaires |
|---|---|---|---|
| NA | 0 | `[]` | Avant la boucle |
| 1 | 1 | `[{i : 0, d : 2.23}]` | Insertion de $\begin{pmatrix}3\\0\end{pmatrix}$ |
| 2 | 2 | `[{i : 0, d : 2.23}; {i : 1, d : 2.00}]` | Insertion de $\begin{pmatrix}3\\1\end{pmatrix}$ |
| 3 | 3 | `[{i : 2, d : 3.16}; {i : 0, d : 2.23}; {i : 1, d : 2.00}]` | Insertion de $\begin{pmatrix}2\\4\end{pmatrix}$ |
| 4 | 3 | `[{i : 0, d : 2.23}; {i : 1, d : 2.00}; {i : 3, d : 1.00}]` | Insertion de $\begin{pmatrix}0\\1\end{pmatrix}$ |
| 5 | 3 | `[{i : 1, d : 2.00}; {i : 4, d : 1.41}; {i : 3, d : 1.00}]` | Insertion de $\begin{pmatrix}0\\1\end{pmatrix}$ |

La liste `candidats` finalement obtenue donne bien les $k$ plus proches vecteurs de `input` par leur indices dans la base de données.

♣. Ranger les candidats par distances décroissantes permet de rejeter rapidement les mauvais candidats. En effet si un nouveau candidat est plus éloigné de `input` que tous ceux déjà dans la liste, on s'en rend compte en $\mathcal{O}(1)$, dès qu'on compare sa distance à `input` à celle de l'élément en tête de liste.

♡. En fait lors des $k$ premières étapes on met dans la liste chaque candidat envisagé, et après on garde toujours exactement $k$ éléments dans la liste, les $k$ meilleurs de ceux déjà envisagés.

**Le type `candidats`.** Afin de représenter la structure de liste `candidats` en C, on se munit des types suivants, implémentant des listes simplement chaînées dont les cellules contiennent des couples (indice, distance).

```
typedef struct s_cellule* candidats;
struct s_cellule {
  int     indice ;        /* indice du point dans le jeu de données */
  double  distance ;      /* distance au point de recherche         */
  candidats next;
};
```

On rappelle que dans cette implémentation, une liste est un pointeur vers sa première cellule.

**Q. 8** Définir la fonction `candidats create_list(int ind, double dist)` qui alloue une cellule pour représenter le candidat d'indice `ind` et de distance `dist` et qui retourne la liste réduite à ce candidat.

**Q. 9** Définir la fonction `void delete_list (candidats lc)` qui libère l'espace mémoire occupé par toutes les cellules de la liste `lc`.

**Q. 10** Définir une fonction `print_list` qui affiche les candidats d'une liste de type `candidats`.

**L'insertion dans une liste triée.** On va maintenant définir une fonction permettant d'effectuer l'insertion dans une liste de candidats triés par distances décroissantes♣. Il est possible que l'insertion ait lieu en tête, ce qui change l'adresse de la première cellule, c'est-à-dire la valeur de la liste. Ainsi la liste dans laquelle insérer doit être passée par référence et non par valeur à la fonction d'insertion. Autrement dit, on doit passer à la fonction, non pas la liste (qui est un pointeur vers la première cellule), mais un pointeur vers la liste (qui est un pointeur vers la case qui enregistre l'adresse de la première cellule), et afin de pouvoir, si besoin, modifier la valeur de la liste afin qu'elle pointe vers la nouvelle première cellule♡.

**Q. 11** Définir la fonction `int insertion_list(candidats* pl, int r, int k, database db, int i, vector input)` modifiant la liste triée pointée par `pl`, qui contient initialement `r` candidats, pour y faire apparaître le candidat `db->datas[i]` si sa distance à `input` le justifie♠. On veillera à préserver le caractère trié de `*pl`, à conserver au plus $k$ cellules dans cette liste candidats, et à libérer les éventuelles zones mémoires devenues inutiles. Cette fonction devra renvoyer le nombre de candidats dans `*pl` à l'issue de l'insertion.

**Le calcul des plus proches voisins.** On peut maintenant implémenter l'algorithme de calcul des plus proches voisins en utilisant une liste pour stocker les (au plus) $k$ meilleurs candidats.

**Q. 12** Définir une fonction `candidats pproche(database data, int k, vector input)` prenant en arguments un jeu de données `data`, un entier `k` tel que `db->size` $\geqslant$ `k` et un vecteur `input`, et retournant le tableau des indices dans `db->datas` des `k` plus proches vecteurs de `input` (parmi les vecteurs dans `data`).

## 3. Les $k$ plus proches voisins par tableaux

**Attention :** Cette partie est mutuellement exclusive de la précédente, relire le chapeau de la section 2 avant de se lancer dans cette section.

♣. On dira seulement liste triée dans la suite.

♡. La suppression dans une liste chaînée demande le même passage par référence, pour gérer le cas où c'est la première cellule qui est supprimée.

♠. c'est-à-dire si cette distance est plus petite que celle realisée par l'un des candidats retenus dans `*pl`, ou si `*pl` contient strictement moins de $k$ candidats, auquel cas on y ajoute le candidat considéré peu importe sa distance à `input`

**Description de l'algorithme.** Afin de calculer les $k$ plus proches voisins d'un point `input` dans un jeu de données `db`, on propose l'algorithme suivant.

- On maintient un entier `j`, un entier `r`, un tableau `indice_pproche` et un tableau `distance_pproche` (tous deux de taille $k$) de sorte que :
  - pour tout indice $i \in [\![0, \texttt{r}[\![$, `indice_pproche[`$i$`]` est l'indice du $i+1$-ème plus proche vecteur de `input` parmi les `j` premières cases de `db->datas` ;
  - `distance_pproche[`$i$`]` est alors la distance entre `input` et `data[indice_pproche[`$i$`]]` ;
  - `distance_pproche`$[\![0, \texttt{r}[\![$ est trié ;
  - `r` $= \min(\texttt{j}, \texttt{k})$ indique le nombre de cases remplies ♣ des tableaux `indice_pproche` et `distance_pproche`.
- On fait parcourir à `j` les valeurs de $0$ à `db->size -1` et on insère, si c'est pertinent, `j` dans le tableau `indice_pproche`, en maintenant les 4 invariants ci-dessus.

**Exemple.** Pour la base de données $\left[\begin{pmatrix}3\\0\end{pmatrix}, \begin{pmatrix}3\\1\end{pmatrix}, \begin{pmatrix}2\\4\end{pmatrix}, \begin{pmatrix}0\\1\end{pmatrix}, \begin{pmatrix}0\\2\end{pmatrix}\right]$ et `input` $= \begin{pmatrix}1\\1\end{pmatrix}$, la recherche des $k = 3$ plus proches voisins du point `input` par l'algorithme proposé ci-avant est décrite, étape par étape, dans le tableau ci-dessous.

| j | r | indice_pproche | distance_pproche | commentaires |
|---|---|---|---|---|
| NA | 0 | {?, ?, ?} | { ? , ? , ? } | Avant la boucle |
| 1 | 1 | {0, ?, ?} | {2.23, ? , ? } | Insertion triée de $\begin{pmatrix}3\\0\end{pmatrix}$ |
| 2 | 2 | {1, 0, ?} | {2.00, 2.23, ?} | Insertion triée de $\begin{pmatrix}3\\1\end{pmatrix}$ |
| 3 | 3 | {1, 0, 2} | {2.00, 2.23, 3.16} | Insertion triée de $\begin{pmatrix}2\\4\end{pmatrix}$ |
| 4 | 3 | {3, 1, 0} | {1.00, 2.00, 2.23} | Insertion triée de $\begin{pmatrix}0\\1\end{pmatrix}$ |
| 5 | 3 | {3, 4, 1} | {1.00, 1.41, 2.00} | Insertion triée de $\begin{pmatrix}0\\1\end{pmatrix}$ |

Le tableau `indice_pproche` donne alors bien la liste des $k$ plus proches vecteurs de `input` par leur indice.

On définit une fonction qui réalise une potentielle insertion d'un nouveau candidat, avant d'implémenter l'algorithme de calcul des plus proches voisins qui utilise cette fonction d'insertion.

**Q. 13** Définir la fonction `int insertion_triee(int* indice_pproche, double* distance_pproche, int r, int k, database data, int j, vector input)` modifiant, si la distance de `data[i]` à `input` le justifie, les tableaux `indice_pproche` et `distance_pproche` pour faire éventuellement apparaître l'indice `j` dans `indice_pproche` et la distance associée dans `distance_pproche`, tout en préservant la correspondance entre les valeurs de ces tableaux, et en maintenant `distance_pproche` trié. Cette fonction devra renvoyer la nouvelle valeur de `r`.

♣. comprendre remplies par des valeurs significatives

**Q. 14** Définir une fonction `int* pproche(database db, int k, vector input)` prenant en arguments un jeu de données `db`, un entier `k` tel que `db->size` $\geqslant$ `k` et un vecteur `input`, et retournant le tableau des indices dans `db->datas` des `k` plus proches vecteurs de `input` (parmi ceux de `db->datas`).

## 4. Classification

Dans le cadre de l'application de l'algorithme des $k$ plus proches voisins à la base de données `MNIST`, l'ensemble des classes est $[\![0, 9]\!]$. Dans le cadre de la classification des points du plan en quadrants (qui fait l'objet de la section 5) l'ensemble des classes est $[\![0, 3]\!]$. Pour l'implémentation on définit donc une constante globale `int nb_max_class`, à initialiser à 10 ou 4 selon l'application, signifiant que l'ensemble des classes à considérer est $[\![0, \texttt{nb\_max\_class} - 1]\!]$.

**Q. 15** Définir une fonction `int classe_majoritaire(database db, candidats lc, int r)`♣ prenant en arguments une liste de candidats `lc` de taille `r` et retournant la classe la plus présente parmi les `r` données classifiées dans `db` repérées par leur indice dans `lc`.

**Q. 16** Implémenter une fonction `int classify(database data, int k, vector input)` prenant en arguments un jeu de données `data`, la taille du jeu `data_size`, un entier `k` tel que `db->size` $\geqslant$ `k` et un vecteur `input`, et retournant la classe la plus présente parmi les `k` plus proches voisins de `input`.

## 5. Création d'un jeu de données "jouet"

Dans cette partie on se propose de vérifier sur un jeu de données simple la correction des fonctions implémentées ci-avant. Pour ce faire on s'intéresse aux vecteurs du carré $[\![0, 255]\!] \times [\![0, 255]\!]$ classifiés selon le quadrant auquel ils appartiennent.

- $[\![0, 127]\!] \times [\![0, 127]\!]$ : classe $0$
- $[\![128, 255]\!] \times [\![0, 127]\!]$ : classe $1$
- $[\![0, 127]\!] \times [\![128, 255]\!]$ : classe $2$
- $[\![128, 255]\!] \times [\![128, 255]\!]$ : classe $3$

**Q. 17** Définir une fonction `int quadrant(vector v)` prenant en argument un vecteur de $[\![0, 255]\!]^2$ et retournant sa classification en quadrant.

**Q. 18** Définir une fonction `database fabrique_jeu_donnees(int db_size)` prenant en argument un entier `db_size` et fabriquant un jeu de `db_size` données. Pour ce faire on peuplera le jeu de données de vecteurs choisis au hasard uniformément dans $[\![0, 255]\!]^2$ accompagnés de la classe indiquée par `quadrant`.

**Q. 19** Vérifier que l'implémentation de la fonction `classify` est correcte. On pourra par exemple vérifier que la classification avec $k = 1$ des points se trouvant dans le jeu de données retourne bien leur quadrant. On pourra vérifier que l'on obtient un taux de classification correcte raisonnable (de l'ordre de $95\%$ et non $25\%$).

## 6. Tests et matrice de confusion

Si ce n'est déjà fait, télécharger la base de données `MNIST` sur github : `https://github.com/halimb/MNIST-txt`. Décompresser l'archive téléchargée de manière à avoir, à côté de votre exécutable, un

♣. **Attention :** Si vous avez traité la Section 3, votre fonction `classe_majoritaire` ne prend pas en deuxième argument une liste de candidats mais le tableau des indices des points les plus proches.

dossier `MNIST-txt` contenant les fichiers `MNIST_train.txt` et `MNIST_test.txt`. Le jeu de données `MNIST_train.txt` contient $60000$ associations image / classe, le jeu de données `MNIST_test.txt` en contient $10000$. Nous n'aurons pas besoin des $60000$ données d'entraînement, ainsi nous utiliserons une partie de ces données pour entraîner notre classifieur et une partie nous servira pour le jeu de tests. Le jeu de données `MNIST_test.txt` pourra donc être mis de côté car il ne sera pas utilisé dans la suite. Dans le fichier `lecture_mnist.c` accompagnant ce sujet sur cahier de prépa, on vous fournit une fonction qui permet de lire cette base de données. Plus précisément la fonction `void mnist_input(int train_size, data* dtrain, int test_size, data* dtest)` prend en arguments :

- un entier `train_size` indiquant la taille souhaitée du jeu de données d'apprentissage ;
- un pointeur `dtrain` vers un jeu de données, la case pointée est écrite, mais non lue ;
- un entier `test_size` indiquant la taille souhaitée du jeu de données de test ;
- un pointeur `dtest` vers un jeu de données, la case pointée est écrite, mais non lue.

Après appel à cette fonction, le pointeur `dtrain` pointe vers un jeu de données de taille `train_size`, le pointeur `dtest` pointe vers un jeu de données de taille `test_size` (pourvu que `train_size`+ `test_size` n'excède pas $60000$).

**Q. 20** Afficher la matrice de confusion obtenue pour $1000$ tests effectués par un classifieur de paramètre $k = 3$ et entraîné sur un jeu de $10000$ données. À titre d'exemple ci-dessous la matrice obtenue dans le corrigé.

| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 96 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 0 | 0 |
| **1** | 0 | 111 | 1 | 0 | 0 | 0 | 0 | 0 | 3 | 0 |
| **2** | 0 | 1 | 94 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 97 | 0 | 2 | 0 | 0 | 2 | 1 |
| **4** | 0 | 0 | 0 | 0 | 101 | 0 | 0 | 0 | 1 | 3 |
| **5** | 0 | 0 | 0 | 1 | 0 | 92 | 1 | 0 | 0 | 0 |
| **6** | 1 | 0 | 0 | 0 | 1 | 0 | 100 | 0 | 0 | 0 |
| **7** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 96 | 1 | 0 |
| **8** | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 78 | 0 |
| **9** | 0 | 0 | 0 | 1 | 4 | 0 | 0 | 0 | 1 | 100 |

La première partie de ce devoir a montré comment trouver les $k$ plus proches voisins d'une donnée à partir d'une base de données classifiées non structurée. En effet la base de données était un simple tableau dans lequel les données étaient placées de manière arbitraire. De ce fait la complexité de la classification était au moins linéaire par rapport à la taille de la base de données, puisqu'il fallait parcourir toutes les données pour trouver les $k$ plus proches du point à classifier. Afin d'améliorer cette complexité on peut réorganiser les données classifiées dans une meilleure structure de données, ce qui permettra, lorsqu'on cherchera les plus proches voisins, de ne considérer qu'une partie des points de la base. Cette amélioration est l'objet de la seconde partie de ce devoir, à implémenter entièrement en OCaml.

# Exercice 2 : Arbres $k$-dimensionnels en OCaml

Si les données étaient de dimension 1, il suffirait de trier les données de la base, puis de chercher le point à classifier dans le tableau trié par dichotomie. Bien sûr le point à classifier ne serait pas nécessairement présent dans la base, mais la dichotomie permettrait de trouver la zone du tableau où se trouvent les points les plus proches du point à classifier.
Pour des données multidimensionnelles, il n'y a pas d'ordre total pertinent qui permette de ranger les données dans une structure linéaire. En effet, si $E$ est un produit cartésien d'ensembles totalement ordonnés, on peut munir $E$ de l'ordre produit, mais celui-ci n'est pas total, ou munir $E$ de l'ordre lexicographique, mais celui-ci favoriserait de manière injustifiée♣ la première composante.

Ainsi pour des données de dimension $k$ on va opter pour une structure arborescente, appelée **arbre $k$-dimensionnel**. **Attention :** le paramètre $k$ ne désigne plus le nombre de voisins recherchés, mais bien la dimension des données. On suppose dans la suite que les vecteurs considérés sont des vecteurs de $\mathbb{R}^k$. Ils seront représentés en OCaml par des objets de type `float array` de taille $k$.

La recherche de la position d'un nouveau point dans cette structure sera l'analogue de la dichotomie : à chaque étape on fait une comparaison entre une valeur de la structure et une valeur du nouveau point, et à l'issue de cette comparaison on sait de quel côté poursuivre la recherche : le sous-arbre gauche ou le sous-arbre droit (contre le sous-tableau gauche ou le sous-tableau droit dans la dichotomie habituelle). Plus précisément, à chaque branchement une seule composante du nouveau point est utilisée dans la comparaison (sinon on retombe sur le problème de l'ordre sur des données multidimensionnelles...).
Afin de ne privilégier aucune composante, la composante considérée change à chaque branchement. Plus précisément à la profondeur $p$, le branchement s'effectue sur la composante $i = p \mod k$. De plus, afin d'avoir la meilleur complexité possible, on souhaite former un arbre équilibré, et donc brancher sur une valeur médiane parmi les valeurs sur la $i$-ème composante des points considérés.
Dans ce sujet, on commence par réaliser le calcul de médiane pour une composante fixée, puis on s'intéresse à la création de l'arbre $k$-dimensionnel représentant un jeu de données, et enfin à la recherche, dans cet arbre, des plus proches voisins d'une nouvelle donnée.

## 1. Sélection rapide

On s'intéresse ici à un problème qui généralise le calcul d'une médiane selon une composante donnée, celui du calcul d'un élément de rang $r$ pour un pré-ordre♡ total quelconque. En effet, pour $i \in [\![1, k]\!]$, la relation $\leqslant_i$ définie sur $\mathbb{R}^k$ par $\forall (x, y) \in \mathbb{R}^k \times \mathbb{R}^k, x \leqslant y \Leftrightarrow x_i \leqslant y_i$ est un pré-ordre total, et en fixant $r$ égal à la moitié du cardinal de l'ensemble considéré, on retrouve bien le problème de médiane initial.

**Q. 1** De manière assez usuelle, on représente un pré-ordre $\preccurlyeq$ sur un ensemble $X$ par une fonction de comparaison, c'est-à-dire une fonction qui prend en entrée deux éléments de $X$ et qui teste si le premier est inférieur au second pour $\preccurlyeq$. Coder une fonction `f_cmp` qui prend en entrée une composante $i \in [\![0, k-1]\!]$ et qui renvoie la fonction de comparaison représentant $\leqslant_{i+1}$. *On prendra garde au décalage entre l'indice des composantes d'un vecteur en maths et les positions de ces composantes dans le tableau représentant ce vecteur.*

♣. le plus proche voisin de $(2, 10)$ parmi $(2, 4), (2, 8), (2, 12), (2, 20), (2, 50), (3, 1), (3, 2), (3, 4), (3, 7), (3, 9), (3, 10)$ est $(3, 10)$, mais il est loin de la position où on insérerait $(2, 10)$ dans cette liste triée pour l'ordre lexicographique.

♡. un pré-ordre sur $X$ est une relation binaire $\preccurlyeq$ réflexive et transitive, on dit qu'il est total ssi $\forall (x, y) \in X^2$, $x \preccurlyeq y$ ou $y \preccurlyeq x$

**Problème de la sélection.** Étant donné un ensemble $X$ muni d'un pré-ordre total $\preccurlyeq$, le problème de sélection se formalise comme suit.

| | | |
|---|---|---|
| SÉLEC | **Entrée :** | $n \in \mathbb{N}^*, r \in [\![0, n[\![$ et $T$ un tableau indicé par $[\![0, n[\![$ à valeurs dans $X$ |
| | **Sortie :** | $v \in X$ tel qu'il existe un tri des valeurs de $T$ dans lequel $v$ est de rang $r$<br>*i.e.* $\exists \sigma \in \mathfrak{S}_n$ tq $(T.(i))_{i \in [\![0,n[\![}$ est croissante et $T.(\sigma(r)) = v$ |

Un algorithme naïf pour résoudre ce problème serait de trier en place le tableau, puis de retourner la valeur se trouvant dans la case d'indice $r$ à l'issue du tri. Cet algorithme en $O(n\log(n))$ (si on utilise le tri fusion par exemple), fait en réalité beaucoup plus que calculer un élément de rang $r$ : il calcule en fait un élément de rang $r'$ pour tout $r' \in [\![0, n[\![$. On propose ici un algorithme récursif, en pratique plus efficace, basé sur la même idée que le tri rapide.

Afin de pouvoir appliquer cet algorithme récursif sur des sous-tableaux sans avoir à les recopier, les algorithmes 1 et 2 ci-dessous prennent en paramètre non seulement un tableau $T$, mais aussi les bornes $d$ et $f$ de l'intervalle d'indices à considérer, ainsi ils n'agissent que sur le sous-tableau $T[\![d, f]\!]$.

**Algorithme 1 :** Partitionner$(T, d, f, p)$

**Entrée :** Un tableau $T$ indicé par $[\![0, n]\!]$, trois entiers $d$, $f$ et $p$ tels que $0 \leqslant d \leqslant p \leqslant f < n$
**Sortie :** Un indice $q \in [\![d, f]\!]$ tel que $T[q]$ vaut finalement la valeur qu'avait $T[p]$ avant l'appel
**Effet :** Le sous-tableau $T[\![d, f]\!]$ est permuté de sorte que $\begin{cases} \forall k \in [\![d, q[\![,\ T[k] \leqslant T[q] \\ \forall k \in [\![q, f]\!],\ T[k] > T[q] \end{cases}$

```
1  pivot ← T[p] ;
2  échanger T[p] et T[d] ;                          (*  on place le pivot au début *)
3  a ← d + 1 ;                                      (*invariant : ∀k ∈ [[d+1, a[[, T[k] ⩽ pivot*)
4  b ← f ;                                          (*invariant : ∀k ∈ ]b, f]], T[k] > pivot*)
5  tant que a ⩽ b faire
6      si T[a] ⩽ pivot alors
7          a ← a + 1
8      sinon
9          échanger T[a] et T[b] ;
10         b ← b − 1 ;
11 échanger T[d] et T[a − 1] ;
12 retourner a − 1
```

**Algorithme 2 :** Sélection$(T, d, f, r)$

**Entrée :** Un tableau $T$, trois entiers $d$ (début), $f$ (fin), $r \in [\![d, f]\!]$ (rang de sélection)
**Sortie :** Une valeur de rang $r$ dans $T$

```
1  si d < f alors
2      p ← U([[d, f]])              // choix d'un pivot uniformément dans [[d, f]] ;
3      q ← Partitionner(T, d, f, p) ;
4      si r = q alors
5          retourner T[r] ;
6      sinon si r < q alors
7          retourner Sélection(T, d, q − 1, r) ;
8      sinon
9          retourner Sélection(T, q + 1, f, r) ;
```

**Q. 2** Coder une fonction `partition` qui prend en argument un tableau `t`, deux indices `d` et `f` valides dans `t` et tels que `d`⩽`f`, un indice de pivot `p` compris (au sens large) entre `d` et `f` et une fonction

de comparaison `cmp`, et qui réalise l'opération de partition du tableau autour de la valeur de pivot `t.(p)`. Cette fonction devra renvoyer l'indice $q$ où est finalement positionnée la valeur pivot.

**Q. 3** Coder `selection_rapide` qui implémente l'algorithme récursif présenté plus haut.

En général, même si l'ordre considéré est fixé, il n'y a pas unicité de l'élément de rang $r$ dans un tableau de taille $n$. Cependant, dans la mesure où les différentes valeurs possibles sont a priori équivalentes pour la suite de l'algorithme, on parlera dans la suite de **la** médiane pour désigner l'élément identifié comme tel lors d'un appel à la fonction `selection_rapide` avec le paramètre $r = \lfloor n/2 \rfloor$.

## 2. Calcul de l'arbre $k$-dimensionnel d'un jeu de données

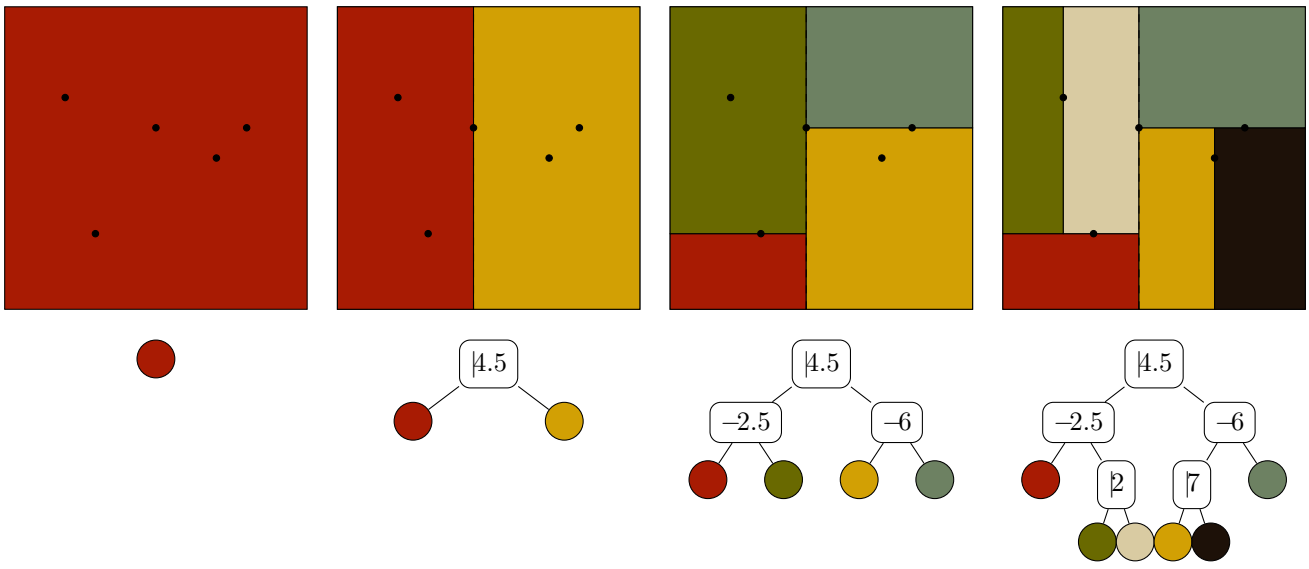
**Définition de l'arbre.** On considère un ensemble $\mathcal{V}$ de vecteurs de $\mathbb{R}^k$. Pour un vecteur $v \in \mathcal{V}$ et une composante $i \in [\![1, k]\!]$, il est possible de partitionner les vecteurs de $\mathcal{V} \setminus \{v\}$ selon leur valeur sur la $i$-ème composante, on définit ainsi les deux ensembles suivants.

$$\mathcal{V}_i^{<v} = \left\{x \in \mathcal{V} \setminus \{v\} \mid x_i < v_i\right\} \text{ et } \mathcal{V}_i^{\geqslant v} = \left\{x \in \mathcal{V} \setminus \{v\} \mid x_i \geqslant v_i\right\}$$

Pour une composante $i \in [\![1, k]\!]$, on peut alors construire récursivement l'arbre $k$-dimensionnel associé à $\mathcal{V}$ commençant à brancher sur la composante $i$ de la manière suivante.

- Si $\mathcal{V} = \emptyset$, c'est l'arbre vide.
- Sinon, on calcule $v \in \mathcal{V}$, la médiane pour la composante $i$, on calcule récursivement $G$ (resp. $D$) l'arbre $k$-dimensionnel pour $\mathcal{V}_v^{<i}$ (resp. $\mathcal{V}_v^{\geqslant i}$) commençant par brancher sur la composante $i+1 \mod k$ et on renvoie alors l'arbre dont la racine est étiquetée par $(v, i)$, le fils gauche est $G$ et le fils droit est $D$.

La figure ci-dessous illustre le processus de construction d'un arbre $k$-dimensionnel pour un jeu de données constitué de points de $[0, 10]\times[0, 10] \subseteq \mathbb{R}^2$ (donc ici $k = 2$). Les points du jeu de données sont représentés par ● ci-dessous. De plus, comme partitionner les points selon leur $i$-ème composante avec $i = 0$, c'est-à-dire selon leur abscisse, revient à regarder où ils sont placés selon une droite verticale, on note | plutôt que $i = 0$ dans les noeuds de l'arbre. De même, – dans un noeud de l'arbre indique une séparation selon une droite horizontale, c'est-à-dire selon la composante $i = 1$, des points



Les arbres $k$-dimensionnels sont donc des arbres binaires potentiellement vides dont les nœuds sont étiquetés par un couple constitué d'une donnée de la base, soit un vecteur de $\mathbb{R}^k$, et d'une composante, soit un entier de $[\![1, k]\!]$. On définit donc les types suivants pour les représenter en OCaml.

```
type vector = float array

type kd_tree =
  | Vide
  | Node  of int * vector * kd_tree * kd_tree
```

De plus, puisqu'on n'utilise pas les classes associées aux données dans cette partie, un jeu de données sera simplement représenté par un tableau de vecteurs.

**Q. 4** Définir une fonction récursive `cree_arbre_kd` qui prend en argument un jeu de données, la dimension de ces données (appelée $k$ jusqu'ici), et qui calcule, par l'algorithme récursif présenté plus haut, l'arbre $k$-dimensionnel associé à ce jeu de données (qui commence par brancher sur la composante $0$).

Dans le fichier compagnon `kd_tree.stu.ml` vous trouverez une fonction permettant la génération d'un jeu de données de test pour votre programme. Vous y trouverez aussi une fonction `draw_kd_tree : kd_tree -> unit` permettant un affichage graphique d'un `kd_tree` et un exemple `main_exemple` de point d'entrée initialisant les fonctions nécessaires à un affichage graphique. En fin de sujet se trouve une encadré présentant succintement comment utiliser le module `Graphics` en OCaml.

## 3. Calcul des plus proches voisins à partir d'un arbre $k$-dimensionnel

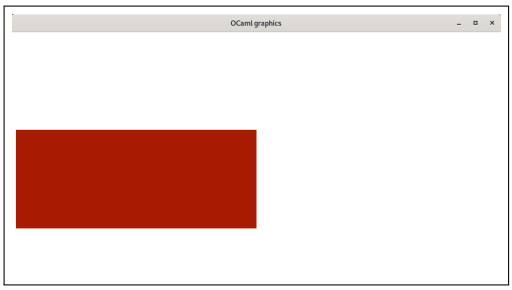
**Q. 5** **Question difficile**
Définir une fonction récursive `pp_voisins` qui prend en argument un entier $n$, un arbre $k$-dimensionnel représentant un jeu de données, et un vecteur $v$ et calculant les $n$ points les plus proches de $v$ dans le jeu de données représenté par l'arbre $k$-dimensionnel.

### Utilisation du module Graphics de OCaml

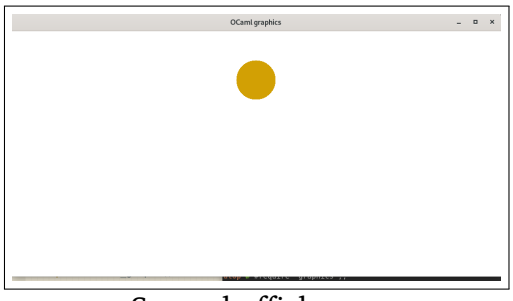
Le module Graphics de OCaml permet de rapidement produire un affichage dans une fenêtre graphique. La fonctionnalité de base du module Graphics est de fournir un **cannevas** sur lequel on peut dessiner. Le présent descriptif n'a pas vocation à remplacer la documentation : https://ocaml.github.io/graphics/graphics/Graphics/index.html, on fournit ici seulement les quelques instructions qui permettent de démarrer, par l'exemple.

```
let () =
  (* Ouvre une fenêtre de 1000*500 pixels (attention espace néc.)*)
  Graphics.open_graph " 1000x500" ;
  (* Change la couleur courante                                  *)
  Graphics.set_color (Graphics.rgb 168 27 3) ;
  (* Dessine un rectangle                                        *)
  Graphics.fill_rect 10 100 490 200 ;
  (* Attend qu'une touche du clavier soit appuyée                *)
  let _ = Graphics.wait_next_event [Key_pressed] in
  (* On supprime le contenu de la fenêtre graphique              *)
  Graphics.clear_graph ();
  (* Change la couleur courante                                  *)
  Graphics.set_color (Graphics.rgb 210 160 4) ;
  (* Dessine un rectangle                                        *)
  Graphics.fill_circle 500 400 40;
  (* Attend qu'une touche du clavier soit appuyée                *)
  let _ = Graphics.wait_next_event [Key_pressed] in
  (* On ferme la fenêtre graphique                               *)
  Graphics.close_graph ()
```

À l'exécution de ce programme, une fenêtre graphique avec un rectangle rouge s'ouvre. Après appui sur une lettre du clavier, l'image est effacée et s'affiche alors un disque jaune. Après un nouvel appui sur une lettre, la fenêtre est fermée (avec utop éviter d'utiliser la croix pour fermer). Si on commente la .11, la 2ème image présentera à la fois le rectangle et le disque.



Premier affichage



Second affichage

**Attention :** Pour utiliser Graphics avec utop on évaluera `#use "topfind";;` puis `#require "graphics";;` afin de charger le module, et ce avant de charger le fichier qui y fait appel. Ce chargement est à renouveler à chaque fois (comme les `#use`).

Pour compiler un fichier code.ml utilisant Graphics en un exécutable code, taper :
`ocamlfind ocamlopt -o code -linkpkg -package graphics code.ml`