Bases de C
TP 3 - Statistiques

Ce TP a pour objectif de vous faire développer une petite bibliothèque de fonctions traitant des statistiques. En pratique, nous ne génèrerons pas encore la bibliothèque à proprement parlé, mais vous devrez garder ce projet pour le semestre suivant afin d'apprendre à générer des bibliothèques automatiquement avec.

1 Bibliothèques et compilation

Afin de pouvoir compiler votre travail, vous allez devoir utiliser la bibliothèque *math* de C (souvent appelée *libmath*) pour utiliser la fonction **`sqrt(3)`**. Vous aurez besoin de faire un `include` de **`math.h`**. Vous verrez qu'à la compilation, un message d'erreur s'affichera et vous indiquera que les fonctions de la libmath ne sont pas trouvées.

Par exemple, essayez d'utiliser **`sqrt(3)`** dans un exemple simple, et vous verrez que le message d'erreur affiché ne correspond pas à un problème de syntaxe, de type, de nom de fonction, ou de variable, mais plutôt à une fonction non référencée (avec éventuellement la mention de **`collect`** et **`ld`** qui renvoient une erreur).

Cet outil **`ld`** a des fonctions essentielles lors de la création d'un programme exécutable : celui-ci rassemble le code des fonctions et leurs noms, puis, il résout les noms des fonctions pour indiquer les adresses qui leurs sont associées. Ainsi, le programme généré peut être chargé en mémoire et exécuté par un processeur.

Pour en revenir à la bibliothèque **`math`**, lorsque l'on souhaite utiliser des bibliothèques externes déjà installées sur la machine, il est nécessaire de les indiquer à **`ld`** pour que celui-ci résolve correctement tous les noms, et ajoute éventuellement des informations indiquant quelles bibliothèques doivent être chargées en mémoire lors du lancement du programme.

En effet, lors de la compilation, le **`.h`** mis en `include` sert en réalité à ajouter le prototype des fonctions utilisées, mais sans y mettre leur implémentation/leur code. Ceci permet de vérifier les types des paramètres et des valeurs de retour, pour confirmer le bon usage des fonctions.
Une fois cela réalisé, on peut compiler le code pour le transformer en langage machine, mais il manque toujours le contenu des fonctions des bibliothèques externes. C'est à ce moment-là que **`ld`** intervient et résout les fonctions encore manquantes (telles que **`sqrt`**) afin de produire le programme exécutable final.

Pour indiquer à **`ld`** d'aller chercher des bibliothèques externes, il est nécessaire d'indiquer avec un *flag* supplémentaire la bibliothèque visée. En général, le flag démarre par un **`-l`** (tiret **L** minuscule) et est suivi du nom de la bibliothèque. Par exemple, pour utiliser la bibliothèque **`xml`**, on va ajouter le flag **`-lxml`** .

Concrètement, la ligne de compilation s'allonge d'un flag. Mais, attention, *l'ordre des flags est extrêmement important* ! La ligne de compilation naïve inclut les flags des bibliothèques en tout dernier, après les `.c` .

Comme par exemple :

```
gcc -W -Wall -Werror -std=c99 -pedantic -o Prog.exe code1.c code2.c -lm
```

En réalité, vous verrez plus tard qu'il est nécessaire de séparer l'étape de *compilation* de cette dernière étape nommée *édition de liens* (ou *link edit* en anglais).

Vous venez de voir comment *utiliser* une bibliothèque externe. Dans le cadre de ce TP, vous vous limiterez à développer des fonctions dans des fichiers spécifiques, et écrire les tests dans d'autres fichiers. Vous compilerez le tout pour générer un exécutable.

2 Statistiques

Vous allez maintenant implémenter plusieurs fonctions utiles pour faire des statistiques, tout en utilisant des tableaux. Pour cela vous créerez d'abord une structure **my_distribution** pouvant accueillir un tableau (précisément, un pointeur vers un tableau) et un entier représentant la taille de ce tableau. Écrivez votre code dans deux fichiers distincts : **my_stats.c** et **my_stats.h**

2.1 Générateur et contenu

1) Après avoir écrit la structure, vous écrirez une fonction générant une *distribution* dont la taille sera donnée en paramètre, ainsi que la valeur maximale que le générateur peut produire.

```
struct my_distribution my_distribution_generator(int len, int max)
```

Pour réussir cela, vous devrez utiliser un générateur aléatoire. En C, il existe quelques fonctions permettant de générer des valeurs aléatoires : **random(3)** et **srandom(3)** permettent cela (ainsi que **rand(3)** et **srand(3)**, fonctions aujourd'hui déconseillées, mais qui ont le même comportement).

random(3) ne prend aucun paramètre et renvoie un nombre aléatoire qui varie à chaque appel.

srandom(3) sert à fixer une *seed* (une *graine* en anglais) qui permet d'initier la séquence de nombres aléatoires qui seront produits par **random(3)**. Ce comportement, bien qu'il puisse paraitre surprenant (générer la même séquence de nombres aléatoires à chaque lancement du programme) est en réalité extrêmement utile pour produire des tests sur des valeurs aléatoires mais dont on peut reproduire la séquence (et donc répéter un bug pour le corriger).

2) Implémentez une fonction **my_print_distribution** qui imprime la distribution donnée en paramètre.

```
int my_print_distribution(struct my_distribution *distribution)
```

Pour imprimer la distribution, vous devrez respecter ce format :

[position] : valeur

Ce qui donnerait pour la distribution $[10, 5, 42]$:

```
$ ./my_distribution
[0] : 10
[1] : 5
[2] : 42
$
```

Si la distribution est vide, vous n'afficherez rien.

2.2 Statistiques de base

Avant de commencer à coder les fonctions suivantes, vous allez d'abord séparer la déclaration de la structure dans un fichier nommé **my_distrib.h** que vous incluerez dans les suivants. Puis, vous mettrez le code du générateur et de l'afficheur dans un fichier nommé **my_distrib.c**, et enfin, vous mettrez vos tests d'exécution dans un fichier nommé **main_distrib.c**.

Lors de la compilation, vous devrez indiquer chacun de ces fichiers en tapant la ligne de compilation suivante :

gcc -W -Wall -Werror -std=c99 -pedantic -o Out my_distrib.c main_distrib.c

3) Implémentez une fonction **my_min** renvoyant la plus petite valeur de la distribution donnée en paramètre.

```
int my_min(struct my_distribution *distribution)
```

4) Implémentez une fonction **my_max** renvoyant la plus grande valeur de la distribution donnée en paramètre.

```
int my_max(struct my_distribution *distribution)
```

5) Implémentez une fonction **my_cardinality** renvoyant le cardinal de la distribution donnée en paramètre.

Pour rappel, le cardinal est le nombre de valeurs présentent dans la distribution.

```
int my_cardinality(struct my_distribution *distribution)
```

6) Implémentez une fonction **my_range** renvoyant l'étendue de la distribution donnée en paramètre.

Pour rappel, l'étendue est la différence entre la valeur la plus élevée de la distribution et la valeur la plus petite.

```
int my_range(struct my_distribution *distribution)
```

2.3 Statistiques

Les fonctions de ces exercices devront être écrites dans un fichier nommé **my_stats.c** et leurs prototypes dans un fichier nommé **my_stats.h** .

7) Implémentez une fonction **my_mean** calculant la moyenne de la distribution donnée en paramètre.

```
double my_mean(struct my_distribution *distribution)
```

8) Implémentez une fonction **my_variance** calculcant la variance de la distribution donnée en paramètre.

```
double my_variance(struct my_distribution *distribution)
```

La variance permet de mesurer la dispersion des valeurs de la distribution par rapport à la moyenne. L'interprétation est simple : une variance élevée indique que les nombres de la distribution sont éloignés de la moyenne (1 et 99 par rapport à 50), une variance faible indique que les nombres de la distribution sont proches de la moyenne (42 et 58 par rapport à 50).

$$\text{variance} = \sigma^2 = Var(L) = \frac{1}{|L|}\sum(L_i - \bar{L})^2$$

Ce calcul peut paraître rebutant, mais il est très simple lorsque l'on regarde de plus près chaque partie de l'expression :

- $(L_i - \bar{L})$ correspond à la différence entre chaque élément de la distribution et la moyenne de la distribution.
- $(L_i - \bar{L})^2$ correspond au carré de la différence entre chaque élément de la distribution et la moyenne de la distribution.
- $\sum(L_i - \bar{L})^2$ correspond à la somme des carrés précédemment décrits. Il s'agit donc de faire une boucle effectuant les traitements précédemment décrits, pour ajouter le résultat à une variable initialisée à 0 juste avant la boucle.
- $\frac{1}{|L|}\sum(L_i - \bar{L})^2$ correspond à la division par le nombre d'éléments de la distribution du total de la somme précédente.

En développant le calcul, on peut arriver à une formule beaucoup plus adaptée au développement, surtout avec des distributions extrêmement grandes (c'est-à-dire des distributions sur lesquelles il coûterait très/trop cher de passer plusieurs fois pour calculer tout d'abord la *moyenne*, et seulement ensuite la différence de chaque élément avec la moyenne), ou lorsque les valeurs arrivent au fur et à mesure sans savoir précisément quand la distribution s'arrêtera (et que l'on souhaite un aperçu des statistiques en temps réel) :

$$\text{variance} = \sigma^2 = Var(L) = \frac{1}{|L|}\sum(L_i - \bar{L})^2 = \left(\frac{1}{|L|}\sum L_i^2\right) - \bar{L}^2$$

Analysons cette deuxième formule pour comprendre son intérêt en tant que développeur :

- L_i^2 correspond à la mise au carré de chaque élément de la distribution.
- $\sum L_i^2$ correspond à la somme de tous les éléments de la distribution mis au carré individuellement, c'est-à-dire d'itérer sur chaque élément, en faire son propre carré, puis d'ajouter ce résultat à une variable mise à 0 juste avant la boucle.
- $\frac{1}{|L|} \sum L_i^2$ correspond à la division de la somme précédente par le nombre d'éléments de la distribution.
- $(\frac{1}{|L|} \sum L_i^2) - \bar{L}^2$ correspond à la différence entre la division précédente, et la moyenne de la distribution mise au carré.

Cette deuxième formule est en réalité bien plus efficace dans un contexte séquentiel (c'est-à-dire lorsque chaque opération est effectuée l'une après l'autre), car nous pouvons faire une somme de tous les éléments qui arrivent et une somme de chacun de leur carré, et seulement lorsque tous les nombres ont été analysés, nous pouvons en déduire la moyenne et la soustraire. Chaque élément n'aura été lu qu'une seule et unique fois lorsqu'il a été généré, et nous avons fait évolué deux variables distinctes à côté : une somme simple (pour produire la moyenne), et une somme des carrés.

9) Implémentez une fonction **my_standard_deviation** calculant l'écart type de la distribution donnée en paramètre.

```
double my_standard_deviation(struct my_distribution *distribution)
```

L'écart type permet de mesurer la dispersion d'une distribution. Plus la mesure est élevée, plus la distribution contient des valeurs éloignées (3 et 90), et plus la mesure est faible, plus la distribution contient des valeurs proches (42 et 46). On peut comparer les écarts types de distributions issues de mêmes choses (par exemple, les écarts types de plusieurs classes composées d'étudiants).

$$\text{écart type} = \sigma = \sqrt{Var(L)}$$

10) Implémentez une fonction **my_relative_standard_deviation** calculant le coefficient de variation (aussi appelé écart type relatif) de la distribution donnée en paramètre.

```
double my_relative_standard_deviation(struct my_distribution *
    distribution)
```

L'écart type relatif permet de mesurer la dispersion d'une distribution autour de la moyenne en générant un pourcentage indépendant des objets étudiés par la distribution. Plus le pourcentage est faible, plus la distribution est concentrée autour de sa moyenne, plus le pourcentage est élevé, plus la distribution est étalée loin de sa moyenne. Cependant, l'écart type relatif ne fonctionne pas lorsque la moyenne est proche de 0 : celui-ci va tendre vers l'infini (donc dépasser les 100%) et sera très sensible aux variations.

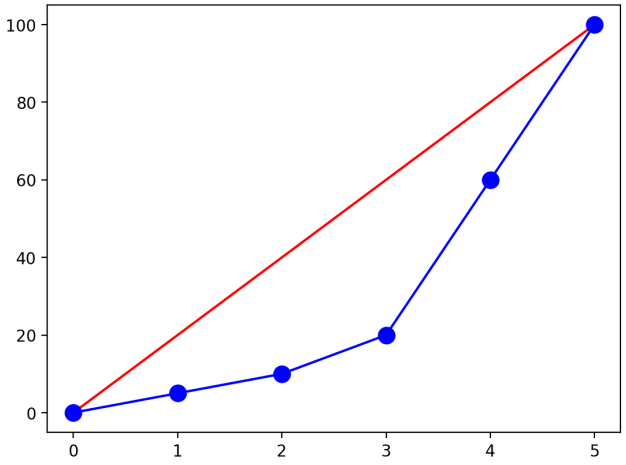
$$\text{coefficient de variation} = c_v = \frac{\sigma}{\mu} = \frac{\sqrt{Var(L)}}{\bar{L}}$$

11) Implémentez une fonction **my_gini_coefficient** calculant le coefficient de Gini de la distribution donnée en paramètre.

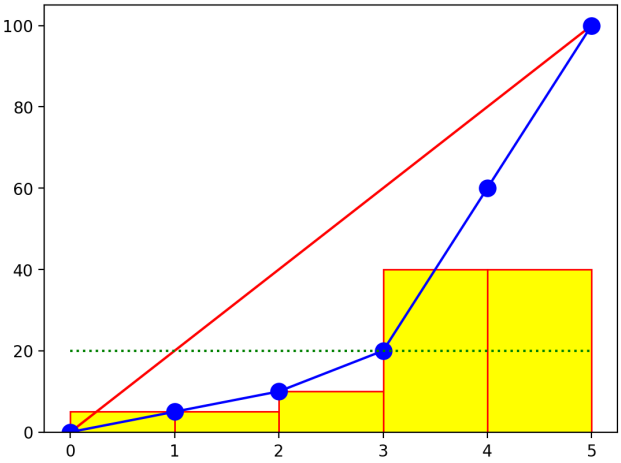
```
double my_gini_coefficient(struct my_distribution *distribution)
```

Un autre indice intéressant issu particulièrement du monde économique et social se nomme le *coefficient de Gini*. Ce coefficient a été constitué pour mesurer les inégalités de revenus au sein d'une population, mais il permet plus généralement de mesurer l'inégalité de répartition d'une variable (donc son interprétation s'approche des mesures de dispersion). Le coefficient de Gini calcule une valeur entre 0 (égalité parfaite entre toutes les valeurs de la distribution) et 1 (inégalité totale/une seule valeur est positive, et toutes les autres sont nulles).
Le calcul théorique de ce coefficient peut paraitre très rebutant à cause des termes employés, et pourtant, il est très facile à comprendre. De plus, le calcul pratique dans le cas discret (pour rappel : non continu/on peut dénombrer les valeurs de la distribution), n'est pas complexe.

Le coefficient de Gini est défini comme le double de l'aire entre la courbe de Lorenz d'une distribution parfaitement égalitaire, et la courbe de Lorenz de la distribution étudiée.
Qu'est-ce qu'une courbe de Lorenz ? Tout simplement la courbe formée par la somme des éléments triés par ordre croissant d'une distribution. Plus visuellement, le tracé bleu correspond à la courbe de Lorenz de la distribution 5 10 40 5 40, et le tracé rouge correspond à la courbe de Lorenz d'une distribution parfaitement égalitaire.



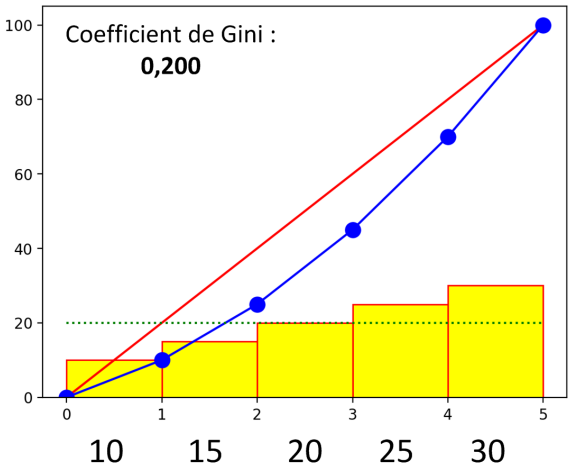
Encore plus visuellement, une fois la distribution ordonnée, on obtient donc 5 5 10 40 40. L'histogramme suivant montre visuellement chaque valeur ajoutée entre les points de la courbe bleue. Donc, en effectuant la somme des valeurs au fur et à mesure, chaque point de la courbe bleue correspond à 5 10 20 60 100.

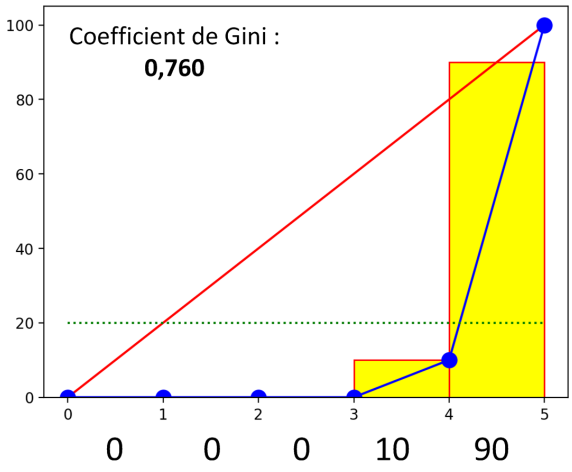


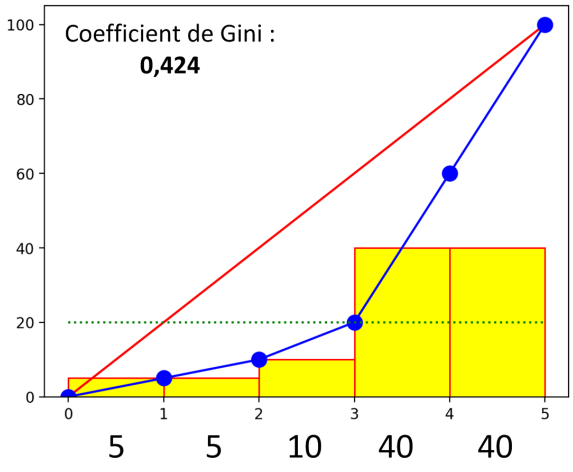
Pour rappel, le coefficient de Gini correspond au double de l'aire entre la courbe bleue et la courbe rouge. Pour bien apprécier visuellement l'écart de cette aire, prenons maintenant deux autres distributions, et comparons l'ensemble des cas :

- 0 0 0 10 90 (une distribution très inégale) : Coeff Gini $= 0,760$
- 5 5 10 40 40 (une distribution plutôt inégale) : Coeff Gini $= 0,424$
- 10 15 20 25 30 (une distribution peu inégale) : Coeff Gini $= 0,200$

Pour bien visualiser ces cas, la moyenne des distributions (20) a été tracée en pointillés verts.







Comme vous pouvez le constater, le cas peu inégal montre une courbe bleue proche de la courbe rouge, et inversement, le cas très inégal montre une courbe bleue très éloignée de la courbe rouge.

Pour rappel, la moyenne n'est pas un indicateur de dispersion, mais bien un indicateur de tendance centrale ou de position. Les nombreux indicateurs que vous venez d'implémenter vous permettront donc de mieux interpréter les données dont vous disposerez dans votre carrière, et éventuellement de voir des phénomènes jusque-là peu visibles.

Reprenons l'implémentation du coefficient de Gini : vous venez de comprendre comment celui-ci est théoriquement construit et interprété. Mais comment l'implémenter facilement ?
Deux formules nous intéressent particulièrement, la première est l'équation de Kendal et Stuart. Dans celle-ci, n correspond au nombre d'éléments dans la distribution, et $L_1, L_2, ..., L_n$ correspondent à chaque élément de la distribution.

$$G = \frac{(1/2n^2) \sum_{i=1}^{n} \sum_{j=1}^{n} |L_i - L_j|}{(1/n) \sum_{i=1}^{n} L_i}$$

En observant cette formule, vous devriez remarquer que la double somme teste littéralement tous les éléments de la distribution deux à deux, ce qui est particulièrement long. Et qu'il est par la suite nécessaire de diviser le tout par la somme des éléments.

Si vous implémentez chacune des parties indépendamment, le temps de calcul sera particulièrement long. Et si vous implémenter toute la formule dans une fonction, vous aurez beaucoup de variables à conserver, et le code risque d'être difficile à comprendre.

Heureusement, vous devriez remarquer que le diviseur est constitué du nombre d'éléments de la distribution, et de leur somme... ce qui ressemble à la moyenne. Et effectivement, Mookherjee et Shorrocks ont simplifié cette équation en introduisant la moyenne (μ) :

$$G = \frac{1}{2\mu n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} |L_i - L_j|$$

Bien que cette formule semble encore complexe, décomposons-là :

- $L_i - L_j$ correspond à la différence entre chaque élément de la distribution.
- $|L_i - L_j|$ correspond à la valeur absolue de la différence entre chaque élément de la distribution.
- $\sum_{i=1}^{n} \sum_{j=1}^{n} |L_i - L_j|$ correspond à la somme des différences absolues des éléments de la distribution.
  Il s'agit donc de faire une double boucle effectuant la différence entre chaque élément de la distribution, pour ajouter le résultat à une variable initialisée à 0 juste avant la boucle. La double boucle implique simplement d'itérer comme sur un tableau à deux dimensions ($|L_1 - L_1| + |L_1 - L_2| + ... + |L_2 - L_1| + |L_2 - L_2| + ... + |L_n - L_n|$).
- $2\mu n^2$ correspond simplement à 2 multiplié par la moyenne de la distribution, multiplié par la quantité d'éléments (le cardinal) au carré.

2.4 Statistiques sur distributions triées

Les fonctions de ces exercices devront être écrites dans un fichier nommé **my_stats_ordered.c** et leurs prototypes dans un fichier nommé **my_stats_ordered.h** .

12) Implémentez une fonction **my_sort** générant une version triée de la distribution donnée en paramètre. La nouvelle distribution *doit* être distincte de celle donnée en paramètre : il faut pouvoir libérer indépendamment l'une sans affecter l'autre et vice versa.

```
struct my_distribution *my_sort(struct my_distribution *distribution)
```

13) Implémentez une procédure **my_inplace_sort** effectuant un tri en place de la distribution donnée en paramètre. Cette procédure déplace donc les valeurs dans la structure sans allouer de nouvelle structure intermédiaire.

Pour rappel, n'hésitez pas à consulter les algorithmes vus dans les précédents cours (tri à bulles, tri par fusion, tri par insertion).

```
void my_inplace_sort(struct my_distribution *distribution)
```

14) Implémentez une fonction **my_median** calculant la médiane de la distribution donnée en paramètre.

Pour rappel, si la distribution a un nombre *impair* de valeurs, alors on renvoie la valeur centrale, inversement, si la distribution a un nombre *pair* de valeurs, alors on renvoie la moyenne des deux valeurs centrales.

```
double my_median(struct my_distribution *distribution)
```

15) Implémentez plusieurs fonctions permettant de calculer l'écart interquartile de la distribution donnée en paramètre.

Vous devrez implémenter les fonctions suivantes :

— **my_quartile_zero** donnant l'index du quartile 0, **my_quartile_one** donnant l'index du quartile 1, ..., **my_quartile_four** donnant l'index du quartile 4

— **my_get_quartile** donnant l'index du quartile demandé (utilisez un *switch ... case* pour appeler la bonne fonction). Si le quartile donné en paramètre n'existe pas, renvoyez -1

— **my_inter_quartile_range** donnant l'écart interquartile

Parmi les mesures statistiques utiles se trouve l'*écart interquartile* basé sur la différence entre des *quartiles*. L'idée des quartiles est de diviser la distribution en 4 parties, chaque quartile étant un séparateur, et d'observer l'écart entre certains séparateurs pour mesurer la dispersion des valeurs.

— le quartile 0 (Q_0) est la valeur la plus petite de la distribution

— le 1^{er} quartile (Q_1) sépare les 25% premières valeurs de la distribution des autres

— le $2^{ème}$ quartile (Q_2) sépare les 50% premières valeurs de la distribution des autres

— le $3^{ème}$ quartile (Q_3) sépare les 75% premières valeurs de la distribution des autres

— le quartile 4 (Q_4) est la valeur la plus élevée de la distribution

Plus visuellement, la distribution triée est séparée ainsi :

$$Q_0 \text{ ... (partie 1) ... } Q_1 \text{ ... (partie 2) ... } Q_2 \text{ ... (partie 3) ... } Q_3 \text{ ... (partie 4) ... } Q_4$$

Dans le cas dit *discret* (c'est-à-dire non continu : on peut dénombrer les valeurs que l'on étudie), il suffit de calculer le *rang* des valeurs, c'est-à-dire leur position dans la distribution triée par ordre croissant. Un quartile dans le cas discret est donc simplement une valeur à une position précise dans la distribution triée.

Pour une distribution contenant n valeurs, on calcule les rangs ainsi :

- le quartile 0 (Q_0) est donc au rang 1
- le 1^{er} quartile (Q_1) est au rang $(n+3)/4$
- le $2^{ème}$ quartile (Q_2) est au rang $(n+1)/2$ (il s'agit de la médiane)
- le $3^{ème}$ quartile (Q_3) est au rang $(3n+1)/4$
- le quartile 4 (Q_4) est donc au rang n

Testons sur la distribution : 1 2 3 4 5 6 7 8 9

- $Q_0 = 1$
- $Q_1 = 3$
- $Q_2 = 5$
- $Q_3 = 7$
- $Q_4 = 9$

La première partie contient donc les valeurs de 1 2 3, la deuxième partie les valeurs de 3 4 5, la troisième partie les valeurs 5 6 7, et la quatrième partie les valeurs 7 8 9.

Si on ne tombe pas sur un rang précis, il est normalement nécessaire de calculer un poids entre la valeur au dessus et la valeur au dessous. Mais pour simplifier notre cas, utilisons le rang inférieur (avec **`floor(3)`**).

```
int my_quartile_zero(struct my_distribution *distribution)
int my_quartile_one(struct my_distribution *distribution)
int my_quartile_two(struct my_distribution *distribution)
int my_quartile_three(struct my_distribution *distribution)
int my_quartile_four(struct my_distribution *distribution)

int my_get_quartile(struct my_distribution *distribution, int quartile)
```

Revenons à la métrique qui nous intéressait : l'écart interquartile. Celui-ci se calcule simplement ainsi :

$$\text{écart interquartile} = EI = Q_3 - Q_1$$

```
double my_inter_quartile_range(struct my_distribution *distribution)
```