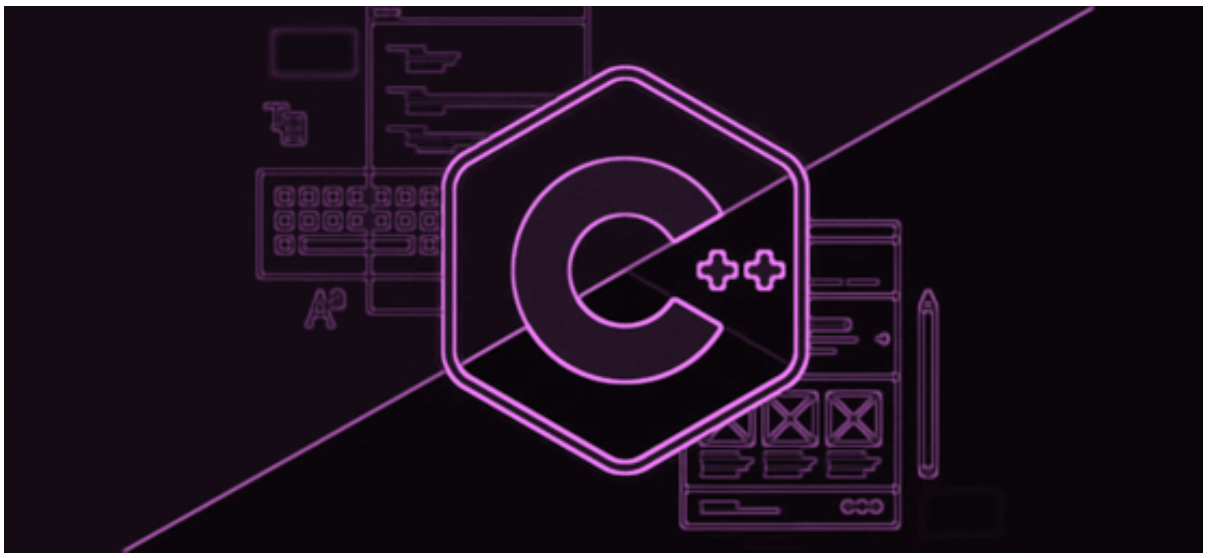
POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR

Embedded C++

Lab 1
Tool Chain and STL

25 Février 2022



Lab Objectives:

- Install a compilation tool chain on your laptop.
- Compile and test a small program.
- Gain competencies on the C++ Standard Template Library (STL)

Auteurs : GHOBRIAL Sara, JONAS Audrey, MOFID Océane

Exercice 1 : STL string and vector<>

Consigne: Ecrire un programme simple qui affiche un histogramme des données lues à partir d'un fichier texte. Nous devons:

- Ne prendre en compte que les valeurs comprises entre 0 et 7999.99 dans le fichier.
- Créer des buckets de largeur 100 sachant qu'une valeur v appartient au bucket b si $100b \leq v < 100b + 100$.
- Utiliser des * pour représenter l'histogramme sur le terminal. Pour le bucket le plus grand, le nombre d'étoiles est de 60, pour les autres il doit être proportionnel.

/!\ Pas de code de bas niveau, chercher dans la librairie STL

Voici les différentes étapes de notre programme:

- ➔ Récupération du nom de fichier qui est passé en argument
- ➔ Création des variables :
  - 2 vecteurs : un double ***buf*** qui contiendra toutes nos valeurs inférieures à 7999.99 lues à partir du fichier et un int ***all_sizes*** qui contiendra toutes les tailles des sous vecteurs créés par la suite.
  - un entier ***maxi*** qui contiendra la taille du plus grand sous vecteur.
  - un string ***line*** qui nous permettra la lecture du fichier.
  - un auto ***mean*** qui contiendra la moyenne de toutes les valeurs récupérées.

```
33  int main(int argc, char *argv[]) {
34
35    string file_name{argv[1]};
36    std::ifstream fin(file_name, std::ios::in);
37
38    vector<double> buf;
39    vector<int> all_sizes; //vecteur contenant les size des subvecteurs
40    int maxi = 0;
41    string line;
42    auto mean = 0.0;
```

On parcourt notre fichier ligne par ligne et on vient convertir cette ligne en double, si le nombre est inférieur à 7999.99 on le conserve dans le vecteur ***buf*** et on calcule ensuite la moyenne.

```
44    while (std::getline(fin, line)) {
45      auto d = std::stod(line);
46      //si dans l'intervalle [0;7999.9] on ajoute au buffer et on calcule la moyenne
47      if(d < 7999.99){
48        buf.push_back(d);
49        //moyenne
50        mean = (buf.size() == 1) ? d : mean + (d - mean) / buf.size();
51      }
52
53    }
```

➔ Calcul de la médiane

```
55    //médiane
56    //on trie le vector par ordre croissant
57    std::sort(buf.begin(), buf.end());
58    auto mid = buf.size() / 2;
59    int t = buf.size();
60    double median = (buf.size() % 2) ? buf[mid] :
61                                       (buf[mid - 1] + buf[mid]) / 2;
```

Comme nos valeurs sont comprises entre 0 et 7999.99, seuls 80 intervalles b de [100*b] à [100*b + 100] peuvent être créés. C'est pourquoi, après avoir créé un sous vecteur, nous venons parcourir 2 boucles for imbriquées :

- une sur 80 pour créer tous les buckets.
- une sur la taille de notre vecteur ***buff***.

Cela nous permet de parcourir toutes les valeurs de notre ***buf*** et de placer celles qui appartiennent à l'intervalle actuel dans le sous-vecteur ***subvector*** correspondant**.** Une fois le sous vecteur plein, nous récupérons sa taille, la comparons au maximum enregistré (afin de modifier le max éventuellement) et nous la stockons dans notre vecteur de int ***all_sizes***. Nous effaçons le sous vecteur et nous réitérons pour l'intervalle suivant.

```
63    //construction des brackets
64    vector<double> subvector;
65    for(int i = 0; i<80; ++i){
66      for(int m = 0; m<t; ++m){
67        //on parcourt tout le buf et
68        //on met dans un sous vector si la valeur est dans l'intervalle
69        if(buf[m]<100*i+100 && buf[m]>100*i){
70          subvector.push_back(buf[m]);
71        }
72      }
73      //on récupère la taille du subvecteur
74      int taille = subvector.size();
75      //on compare la size de la bracket actuelle au max précédent
76      maxi = std::max(maxi,taille);
77      all_sizes.push_back(taille); //on stocke la size du subvecteur
78      //on efface le subvecteur
79      subvector.clear();
80    }
```

➔ Nous affichons les données du fichier : le nombre d'éléments, la moyenne et la médiane.

➔ Nous procédons à l'affichage de l'histogramme, pour cela nous parcourons les intervalles puis :

- nous calculons le nombre d'étoiles à afficher pour chaque intervalle en réalisant un simple produit en croix en fixant que l'intervalle le plus grand aura 60 étoiles.
- nous affichons la borne inférieure suivie du nombre d'éléments dans l'intervalle; les nombres sont espacés avec std::setfill(' ') et std::setw afin que toutes les données suivantes soient alignées.
- nous affichons ensuite la ligne d'étoiles avec std::string() et après un retour à la ligne nous réitérons pour l'intervalle suivant.

```
81    //infos
82    std::cout << "number of elements = " << buf.size()
83              << ", median = " << median
84              << ", mean = " << mean << std::endl;
85
86    //affichage
87    //affiche le bracket inf puis le nbr d'éléments de la bracket puis les étoiles
88    for(int i = 0; i<80 ; ++i){
89      //les setw nous permettent d'aligner les éléments
90      int nbr = int((all_sizes[i]*60)/maxi);
91        std::cout << std::setfill(' ') << std::setw(6) << 100*i <<
92        std::setw(6) << all_sizes[i] << ' ' <<std::string(nbr, '*') << std::endl ;
93    }
94
95  }
```

En appliquant notre programme au fichier data/data_100000.txt nous obtenons le résultat attendu :

```
oceanemofid@DESKTOP-T2T25BS:/mnt/d/Cours/ELEC4/C++/Embarqué/TPs/TP1$ ./histogram data/data_100000.txt
number of elements = 99011, median = 1702.73, mean = 2048.26
     0     0
   100     0
   200   334 ****
   300   857 **********
   400  1706 ********************
   500  2392 ******************************
   600  3130 ***************************************
   700  3684 **********************************************
   800  3947 *************************************************
   900  4270 *****************************************************
  1000  4466 ********************************************************
  1100  4432 *******************************************************
  1200  4327 ******************************************************
  1300  4175 ****************************************************
  1400  4031 **************************************************
  1500  3897 ************************************************
  1600  3738 **********************************************
  1700  3388 ******************************************
  1800  3334 *****************************************
  1900  3140 ***************************************
  2000  2856 ***********************************
  2100  2656 *********************************
  2200  2471 *******************************
  2300  2340 *****************************
  2400  2155 ***************************
  2500  1957 ************************
  2600  1894 ***********************
  2700  1692 *********************
  2800  1537 *******************
  2900  1493 ******************
  3000  1371 *****************
  3100  1298 ****************
  3200  1155 **************
  3300  1143 **************
  3400   999 ************
  3500   902 ***********
  3600   882 ***********
  3700   813 **********
  3800   748 **********
  3900   669 ********
  4000   647 ********
  4100   561 *******
  4200   539 *******
  4300   513 ******
  4400   470 ******
  4500   431 *****
  4600   403 *****
  4700   366 ****
  4800   350 ****
  4900   342 ****
  5000   290 ***
  5100   280 ***
  5200   279 ***
  5300   239 ***
  5400   250 ***
  5500   198 **
  5600   207 **
  5700   188 **
  5800   165 **
  5900   174 **
  6000   170 **
  6100   131 *
  6200   131 *
  6300   130 *
  6400   109 *
  6500   136 *
  6600    99 *
  6700   115 *
  6800    97 *
  6900    84 *
  7000    74
  7100    69
  7200    82 *
  7300    67
  7400    63
  7500    68
  7600    50
  7700    48
  7800    58
  7900    54
```

Exercice 2 : STL Trees and Hashes

One way

Consigne: Ecrire un programme map1.cpp qui lit un fichier data/full_XX.txt constitué de 2 colonnes: un identifiant unique puis un nombre et qui propose une entrée à l'utilisateur.

- L'utilisateur entre un identifiant correct (présent dans le fichier), le programme retourne le nombre correspondant.
- L'utilisateur n'entre pas un identifiant correct, le programme affiche un message d'erreur puis propose de nouveau une saisie d'identifiant.
- L'utilisateur entre le mot 'END', le programme s'arrête.

/!\ Pas plus de 50 lignes de code

Voici les différentes étapes de notre programme:

- ➔ Récupération du nom de fichier qui est passé en argument.
- ➔ Création des variables :
  - ◆ 2 strings : ***line*** qui nous permettra la lecture du fichier, ***qin*** qui contiendra l'identifiant saisi par l'utilisateur.
  - ◆ une std::unordered_map ***identifierKey*** qui contiendra des paires de string,double qui nous permettra de stocker identifiants et les valeurs associées du fichier texte.

Choix du container: Nous avons choisi, parmi tous les containers, d'utiliser une unordered_map. Nous avons fait ce choix car les unordered_map sont des conteneurs associatifs qui permettent de stocker des éléments de la forme : clé -> valeur. Cette description correspond tout à fait aux données lues dans le fichier puisque chaque identifiant (clé) est lié à un nombre (valeur). De plus pour les unordered_map, une valeur ne peut avoir qu'un identifiant unique, ce qui correspond à notre cas. D'autre part, la complexité de ce type de container est intéressant (cf. réponse à la question 4 - Complexité d'une query).

```
34      string file_name{argv[1]};
35      std::ifstream fin(file_name, std::ios::in);
36      string line;
37      string qin;
38      std::unordered_map<std::string, double> identifierKey;
```

- ➔ On parcourt d'abord notre fichier ligne par ligne afin de récupérer les valeurs des différents champs. Nous avons décidé d'utiliser un istringstream ***stream*** car à l'aide de l'opérateur >> nous pouvons récupérer les différents champs de chaque ligne ***line*** du fichier d'entrée.
- ➔ On stocke l'identifiant lu dans une variable string **s** et la valeur associée dans la variable double **f.**
- ➔ On ajoute dans l'unordered_map ***identifierKey*** l'élément lu et sa clé d'identification avec la fonction insert()

➔ On recommence pour la ligne suivante jusqu'à la fin du fichier.

```
40      while (std::getline(fin, line)) {
41          std::istringstream stream(line);
42          string s;
43          double f;
44          stream >> s >> f ;
45          identifierKey.insert({s, f});
46      }
```

➔ On crée ensuite une boucle infinie à l'aide d'un for afin de gérer l'interface avec l'utilisateur.

- Tant que l'utilisateur ne tape pas 'END', on affiche le prompt "query" et on récupère l'identifiant saisi.
- Si l'utilisateur a saisi "END", on ne vérifie pas s'il s'agit d'un identifiant correct, on affiche directement "Bye" et on quitte la boucle.
- Sinon, afin de vérifier si l'identifiant saisi est correct, on utilise la fonction find() de unordered_map. Cette dernière renvoie un itérateur sur l'élément qui correspond à la clé passée en paramètre, si aucun élément ne correspond, elle renvoie unordered_map::end. A l'aide d'un 'if'/'else' on vérifie donc que l'identifiant est correct:
  - si oui on affiche l'identifiant puis la valeur associée
  - si non on affiche "this ID does not exist"

```
48      for(;;) { // boucle infinie
49          std::cout << "query > ";
50          std::cin >> qin;
51
52          auto it = identifierKey.find(qin);
53          if(qin == "END"){
54                  break;
55              }
56          else{
57              if(it != identifierKey.end()) {
58                  std::cout << "value[" << qin << "] = "
59                        << it->second << std::endl;
60              }
61              else {
62                  std::cout << "This ID does not exist" << std::endl;
63              }
64          }
65      }
66      std::cout << "Bye..." << std::endl;
67  }
```

En appliquant notre programme au fichier data/full_10.txt nous obtenons le résultat attendu :

```
oceanemofid@DESKTOP-T2T25BS:/mnt/d/Cours/ELEC4/C++/Embarqué/TPs/TP1$ ./map1 data/full_10.txt
query > 8789d2ce5b3b
value[8789d2ce5b3b] = 2638.9
query > 8a5b74971f70
value[8a5b74971f70] = 1990.52
query > 8a5b74972f70
This ID does not exist
query > END
Bye...
```

Complexité en notation big O d'une query :

Au départ, nous avions utilisé un container de type std::map. Ce type de container correspond à un arbre red-black qui, pour une requête, a une complexité de **O(log(n))**. Alors que les containers de type std::unordered_map ont une complexité, pour une requête, de **O(1)**, c'est-à-dire une complexité constante.

Red–black tree

Type	Tree
Invented	1972
Invented by	Rudolf Bayer

Complexities in big O notation

Space complexity	
Space	$O(n)$

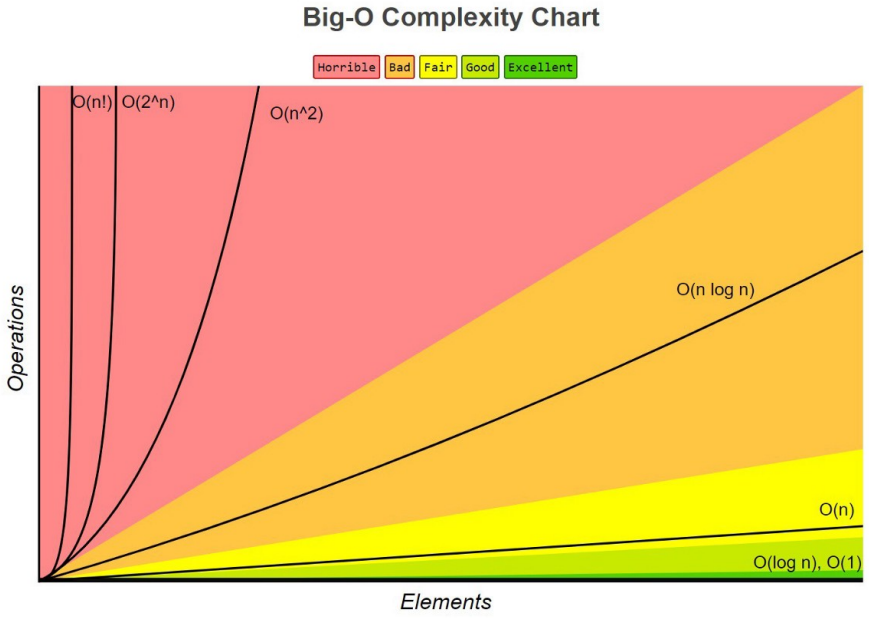
Time complexity		
Function	Amortized	Worst Case
Search	$O(\log n)$[1]	$O(\log n)$[1]
Insert	$O(1)$[2]	$O(\log n)$[1]
Delete	$O(1)$[2]	$O(\log n)$[1]

En effet, d'après la documentation, unordered_map a en moyenne une complexité constante O(1) et dans le pire des cas une complexité linéaire O(n). Cela dépend de la fonction de hachage utilisée par le container. Or une fonction de hachage spécialisée existe déjà pour les clés de type string.

Comme la requête se fait en une seule ligne de code à l'aide de la fonction std::unordered_map::find, la complexité d'une requête est égale à la complexité de la fonction find, soit O(1).

Complexity

Average case: constant.
Worst case: linear in container size.



Hash table

Type	Unordered associative array
Invented	1953

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(*n*)[1]	O(*n*)
Search	O(1)	O(*n*)
Insert	O(1)	O(*n*)
Delete	O(1)	O(*n*)

Exercice 3 : STL Trees and Hashes

Two ways

Consigne: Améliorer le code précédent afin que lorsque l'utilisateur entre un nombre v, le programme renvoie tous les identifiants dont le nombre associé soit égal $v \pm 1\%$.

/!\ - Pas plus de 15 lignes de plus
- Plusieurs identifiants peuvent avoir la même valeur
- Itérer sur des vecteurs ne donnent pas tous les pts

Voici les différentes étapes de notre programme: Au programme précédent nous ajoutons :

➔ Une multimap nommée ***keyByValueMap*** qui contient des paires de type (double, string), et dont la clé est la valeur (et non plus l'identifiant). Ce conteneur permettra de retourner tous les identifiants associées à une valeur dans l'intervalle recherché.

```
34      string file_name{argv[1]};
35      std::ifstream fin(file_name, std::ios::in);
36      string line;
37      string qin;
38      std::unordered_map<std::string, double> identifierKey;
39      std::multimap<double, std::string> keyByValueMap;
40
41      while (std::getline(fin, line)) {
42          std::istringstream stream(line);
43          string s;
44          double f;
45          stream >> s >> f ;
46          identifierKey.insert({s, f});
47          keyByValueMap.insert({f, s});
48      }
```

Choix du container : Comme dans une std::map, les valeurs dans une std::multimap sont ordonnées et cela diminue le temps de réponse d'une requête, car les valeurs retournées se succèdent. De plus, contrairement à une map, plusieurs éléments du conteneur peuvent avoir des clés équivalentes.

D'abord, on identifie une valeur double par la présence d'un signe "+". Comme l'élément saisi est stocké dans une chaîne de caractères ***qin***, on le convertit en double. Cette conversion se fait à l'aide de la fonction de la STL ***std::stod()***, qui peut émettre des exceptions, d'où l'utilisation d'un bloc try/catch. Dans le cas où une exception est détectée, un message est affiché sur la sortie d'erreur standard.

```cpp
56          if (qin[0] == '+') { // reading a double from qin string
57              try {
58                  double din = std::stod(qin);
59                  double dmin = .99 * din;
60                  double dmax = 1.01 * din;
61                  bool found = false;
62
63                  for(auto it = keyByValueMap.lower_bound(dmin); it != keyByValueMap.end(); ++it) {
64                      if(it->first > dmax)
65                          break;
66
67                      std::cout << "value[" << it->second << "] = " << it->first << std::endl;
68                      found = true;
69                  }
70
71                  if(!found)
72                      std::cout<< " No identifier found for specified value: " << qin << std::endl;
73
74              } catch(std::invalid_argument&) {
75                  std::cerr << qin << " is not a valid key value" << std::endl;
76              }
```

On parcourt ensuite la multimap en partant du premier élément dont la clé est supérieure ou égale à v - 1% en utilisant lower_bound, et on sort de la boucle si la clé de l'élément courant est supérieure à v + 1%.
Si la valeur saisi n'existe pas dans le fichier, on affiche "No identifier found for specified value".

En appliquant notre programme au fichier data/full_1000.txt nous obtenons le résultat attendu:

```
oceanemofid@DESKTOP-T2T25BS:/mnt/d/Cours/ELEC4/C++/Embarqué/TPs/TP1$ ./map2 data/full_1000.txt
query > 44e2d4b8d7aa
value[44e2d4b8d7aa] = 1358.56
query > +5000
value[375df8b1ac86] = 5022.42
query > +614
value[f6a5f1e9f733] = 612.69
value[7860f4b10a57] = 615.25
value[1201267a89a7] = 615.25
query > END
Bye...
```

Gestion des exceptions et cas ou la valeur saisi n'existe pas dans le fichier:

```
oceanemofid@DESKTOP-T2T25BS:/mnt/d/Cours/ELEC4/C++/Embarqué/TPs/TP1$ ./map2 data/full_1000.txt
query > +0
No identifier found for specified value: +0
query > +34567890
No identifier found for specified value: +34567890
query > dfgyhujiop
This ID does not exist
query > +dfghjklm
+dfghjklm is not a valid key value
query > END
Bye...
```

Complexité en notation big O d'une query :

Pour ***map2.cpp***, nous avons 2 cas de figure : la recherche d'une ***key*** et la recherche d'une ***value***. Pour la recherche d'une ***key***, c'est identique à la complexité de std::unordered_map::find, aussi utilisé dans ***map1.cpp***, soit O(1).

Complexity

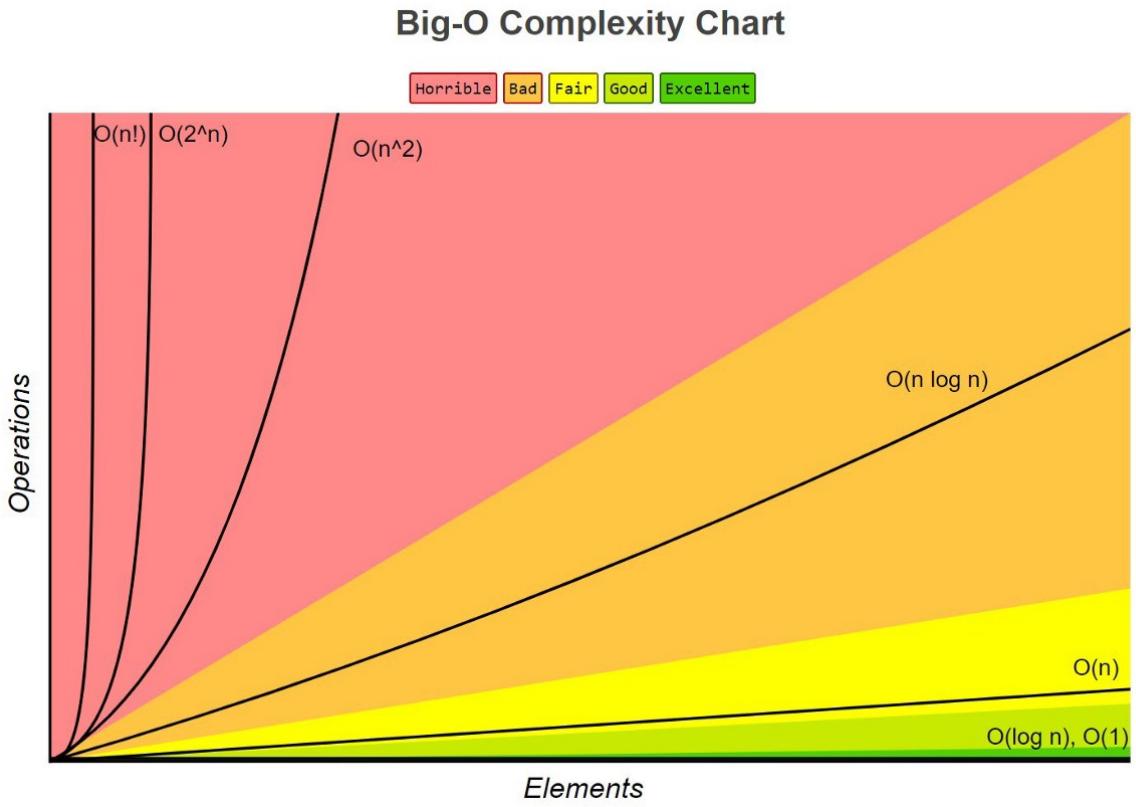
Average case: constant.
Worst case: linear in container size.

Pour la recherche d'une ***value***, la complexité est de O(n) + O(log(n)). O(n) correspond à la recherche linéaire de la boucle for, et O(log(n)) correspond à la complexité de la fonction std::multimap::lower_bound. Cette complexité peut être simplifiée à **O(n)**, car O(log(n)) est négligeable devant O(n) pour des valeurs de n très grandes.

Complexity

Logarithmic in size.

D'autre part, nous avons préféré utiliser une multimap et non une unordered_multimap (qui a une meilleure complexité) car à la différence de cette dernière, une multimap ordonne ses éléments en fonction des clés ce qui diminue le temps de la requête car les on peut sortir de la boucle dès qu'on sort de l'intervalle recherchée.



Extra credit : pseudo number generators

Consigne: Ecrire un programme simple qui crée un fichier texte tel que l'histogramme de ce dernier soit équivalent à l'histogramme obtenu avec le fichier data/data_100000.txt.

Voici les différentes étapes de notre programme:

- Création des variables :
  - Une constante ***double nrolls*** afin de représenter le nombre de double à générer dans le fichier texte. Comme la consigne nous demande de reproduire le fichier data/data_100000.txt, cette variable est initialisée à 100 000, elle contiendra donc 100 000 doubles.
  - Un ***std::ofstream outfile*** nous permettant de diriger nos inputs vers le fichier associé soit ***test.txt***

```
27      const double nrolls=100000;  // nombre de réels à générer
28      std::ofstream outfile ("test.txt");
```

  - Un ***std::default_random_engine generator***, soit un pseudo random generator, qui nous permet de produire des nombres aléatoires
  - Un ***std::normal_distribution<double> distribution*** qui nous permet de choisir comment les nombres aléatoires seront générés. Ici nous avons choisi une distribution normale (soit de Gauss) et les paramètres du constructeur correspondent à la moyenne et la variance de la distribution de Gauss que nous souhaitons générer.

```
30      std::default_random_engine generator; //générateur
31      std::normal_distribution<double> distribution(1715.81,500.0); //moyenne et variance
```

Maintenant que nos variables ont été déclarées, nous allons procéder à l'écriture de notre fichier. Pour cela, on réalise une boucle for car nous connaissons le nombre d'itérations soit 100 000. Dans cette boucle, nous créons une variable double number qui va contenir le nombre aléatoire généré par la distribution paramétrée plus haut. Concernant ces paramètres, nous avons choisi de prendre la médiane du fichier data/data_100000.txt ce qui nous permettra d'avoir le pic d'effectif maximal au même endroit que pour data/data_100000.txt. Pour la variance, nous avons testé plusieurs valeurs afin d'avoir à peu près la même étendue.
Nous redirigeons ensuite le double vers le fichier en utilisant les chevrons, car le fichier est un ofstream.

```
33      for (int i=0; i<nrolls; ++i) {
34          double number = distribution(generator); //on génère un nombre selon la distribution spécifiée
35          outfile << number << std::endl; //on l'écrit dans le fichier
36      }
```

Après avoir écrit les 100 000 doubles dans le fichier, nous fermons le fichier.

→
```
39      outfile.close();
```

Après avoir compilé le programme, nous exécutons et le fichier test.txt apparaît dans notre arborescence, et à son ouverture on observe bien les 100 000 doubles créés.

test.txt

TP1 > test.txt

99989	1902.13
99990	2310.59
99991	613.922
99992	905.924
99993	1864.56
99994	1477.18
99995	1696.25
99996	1458.91
99997	1783.08
99998	1649.83
99999	1683.5
100000	766.896

Afin de confirmer que notre fichier respecte le cahier des charges, nous exécutons le programme histogram avec ce fichier et on observe que l'histogramme est très similaire à celui présent du fichier data/data_100000.txt

```
oceanemofid@DESKTOP-T2T25BS:/mnt/d/Cours/ELEC4/C++/Embarqué/TPs/TP1$ g++ -std=c++17 -Wall bonus.cpp  -o bonus
oceanemofid@DESKTOP-T2T25BS:/mnt/d/Cours/ELEC4/C++/Embarqué/TPs/TP1$ ./bonus
oceanemofid@DESKTOP-T2T25BS:/mnt/d/Cours/ELEC4/C++/Embarqué/TPs/TP1$ ./histogram test.txt
number of elements = 100000, median = 1715.93, mean = 1713.57
    0    38
  100    63
  200   106
  300   210 *
  400   367 **
  500   530 ***
  600   862 ******
  700  1175 ********
  800  1806 *************
  900  2480 ******************
 1000  3384 ************************
 1100  4157 *****************************
 1200  5130 ************************************
 1300  6095 *******************************************
 1400  6842 ************************************************
 1500  7613 *****************************************************
 1600  7815 ******************************************************
 1700  8005 *******************************************************
 1800  7844 ******************************************************
 1900  7095 *************************************************
 2000  6367 ********************************************
 2100  5483 **************************************
 2200  4448 *******************************
 2300  3607 *************************
 2400  2741 ******************
 2500  1964 *************
 2600  1363 *********
 2700   897 ******
 2800   601 ****
 2900   337 **
 3000   244 *
 3100   143 *
 3200    77
 3300    39
 3400    17
 3500     2
 3600     2
 3700     6
 3800     0
 3900     0
 4000     0
 4100     0
 4200     0
 4300     0
 4400     0
```