Assignment 1

(1) Number of integers in a vector

(a) (Week 1) Write a program that generates random M integers and puts them in an `vector`. Then it generates another N random integers and counts how many of them are in the array using an iterator.

a)

The code for a) can be found in 1.a.cpp.

The program generates M random integers between (0-100) and stores them i a vector<int>. It then generates N new random integers and, using an iterator, checks whether each appears in the vector. Each probe contributes at most one to the count (due to break after first match in line 42). To nsure different results each time, the program seeds the random number generator with `srand(time(nullptr));` , which extracts a seed based on the current local time.

Edge cases:

- If M=0 the vector is empty
- if N=0 no checks are made.

(b) (Week 2) Estimate the worst-case complexity of your program and justify with a line-by-line explanation of the number of steps taken in the program.

b)

We interpret the exercise such that only N matters asymptotically. Since both M and N can grow arbitrarily large, we treat them as being of the same order (M ≈ N).

Line-by-line analysis can be viewed in 1.b.cpp.

Total worst-case time:

$$T(N) = O(N) + O(N) + O(N^2) = O(N^2)$$

Interpretation:

- Best case: probes match early → O(N)
- Worst case: every probe scans the full vector → $O(N^2)$
- Space complexity: O(N) for storing the vector.

Thus, the program runs in quadratic time $\Theta(N^2)$ in the worst case.

(2) (Old exam question, week 2) What is the time complexity (Big-O) of `myMethod`? Argue for your answer:

```
int myMethod(int N)
    {
        int x = 0;
        int y = 0;
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                for (int k = 0; k < N * sqrt(N); k++)
                {
                    x++;
                }
            }
        }
        for (int i = 0; i < N * N; i++)
        {
            y++;
        }
        return x + y;
    }
```

We interpreted this task by firstly writing the function 2.cpp and then adding comments for every time complexity stating its value in Big-O notation. Then in the end we added a total sum of all the time complexities and a conclusion on the complexity of the overall function which would be the highest single value of Big-O. The time complexity of the function myMethod is listed as a comment in the bottom of program 2.cpp

(3) (Week 2) For each of the following four program fragments:
(a) Give an analysis of the running time in Big-Oh notation.
(b) Implement the code inside a C function, and give the running time for several values of N.
(c) Compare your analysis with the actual running times.
(d) Does compiling/running programs with the `-O` optimization flag change anything in your analysis?

```
(1) sum = 0;
    for (i = 0; i < n; ++i) {
        ++sum;
    }
```

```
(2) sum = 0;
    for (i = 0; i < n; ++i) {
      for (j = 0; j < n; ++j) {
        ++sum;
      }
    }
```

```
(3) sum = 0;
    for (i = 0; i < n; ++i) {
      for (j = 0; j < n * n; ++j) {
        ++sum;
      }
    }
```

```
(4) sum = 0;
    for (i = 0; i < n; ++i) {
      for (j = 0; j < i; ++j) {
        ++sum;
      }
    }
```

a)

Here's (a): Big-Oh running-time analysis for each fragment.

1.

```
for (i = 0; i < n; ++i) ++sum;
```

Executes the body n times → $\Theta(n)$ ($O(n)$).

2.

```
for (i = 0; i < n; ++i)
  for (j = 0; j < n; ++j)
        ++sum;
```

Inner loop runs n times for each of n outer iterations → $n \cdot n = n^2$ → $\Theta(n^2)$ ($O(n^2)$).

3.

```
for (i = 0; i < n; ++i)
  for (j = 0; j < n*n; ++j)
        ++sum;
```

Inner loop has n^2 iterations, repeated n times → $n \cdot n^2 = n^3$ → $\Theta(n^3)$ ($O(n^3)$).

4.

```
for (i = 0; i < n; ++i)
  for (j = 0; j < i; ++j)
        ++sum;
```

Total iterations = $\sum_{i=0}^{n-1} i = n(n-1)/2$ → Θ(n²) (O(n²)).

b), c) and d)

The four different C (not C++) implementations can be found in 3b.1,2,3,4.c

We implemented each program fragment as a standalone C function, following the structure shown in the example code. To ensure the compiler does not optimize away the increments, sum was declared as volatile. We measured execution times using clock() from <time.h>, with the duration computed as

$$\text{duration} = \frac{\text{end} - \text{start}}{\text{CLOCKS\_PER\_SEC}}$$

For each fragment, we tested different values of N and recorded the total time (see tables).

The measured results align with the theoretical analysis from part (a):

- Fragment (1): Linear growth (O(N)). Doubling N roughly doubles the running time.
- Fragment (2): Quadratic growth (O(N²)). Doubling N multiplies the running time by about 4.
- Fragment (3): Cubic growth (O(N³)). Doubling N increases the running time by about 8.
- Fragment (4): Quadratic growth (O(N²)), since the inner loop performs $\sum_{i=0}^{n-1} i = n(n-1)/2$ increments.

Comparison:
The empirical times fit well with the asymptotic predictions. Absolute times differ depending on hardware and compiler, but the *growth rates* confirm the theoretical Big-Oh analysis.

Effect of -O optimization: Using the -O flag does not change the complexity class of any fragment. It only reduces the constant factors, so the program executes faster in practice, but the asymptotic behavior (O(N), O(N²), O(N³)) remains the same.

Data:
Below we have documented the runtime in big O and noted down our actual runtimes in four tables.

1) O(N)

N values	10.000	100.000	1.000.000
RT in Big-O	10.000	100.000	1.000.000
Actual RT	0.000035 s	0.000371 s	0.003193 s
RT w/ -o	0.000034 s	0.000381 s	0.003418 s

The runtime grows linearly with N, matching the predicted **O(N)** complexity (10× input → ~10× time).

2) O(N^2)

N values	100	1.000	10.000
RT in Big-O	10.000	1.000.000	100.000.000
Actual RT	0.000036 s	0.003774 s	0.129951 s
RT w/ -o	0.000011 s	0.003544 s	0.125271 s

The runtime grows quadratically with N, matching the predicted **O(N^2)** complexity (10× input → ~100× time).

3) O(N^3)

N values	100	1.000	5.000
RT in Big-O	1000000	1 000 000 000	125 000 000 000
Actual RT	0.002025 s	0.976280 s	117.518805 s
RT w/ -o	0.000962 s	0.978737 s	117.654196 s

The runtime grows cubically with N, matching the predicted **O(N^3)** complexity (10× input → ~1000× time).

4) O(N^2)

N values	100	1.000	10.000
RT in Big-O	10.000	1.000.000	100.000.000
Actual RT	0.000024 s	0.001594 s	0.079370 s
RT w/ -o	0.000008 s	0.000492 s	0.052773 s

The runtime grows quadratically with N, matching the predicted **O(N^2)** complexity (10× input → ~100× time).

(4) (a) (Old exam question, week 1) Write an implementation of the **Abstract Data Type** `MaxHeap` described below. The implementation **must** be done by using the standard class `vector` from C++ **and it does not have to be efficient**. If you need to iterate through the `vector`, you **must** use and iterator and its methods and operators `begin()`, `end()`, `++` and `*`

```
class MaxHeap
{
public:
    // is the heap empty?
    virtual bool isEmpty() const = 0;

    // number of elements in the heap
    virtual int size() = 0;

    // add an element to the heap
    virtual void insert(const int x) = 0;

    // find the maximum element in the heap
    virtual const int findMax() const = 0;

    // delete and return the maximum element of the heap
    virtual int deleteMax() = 0;
};
```

We interpreted this task by first copy pasting the code snippet from the task description into our 4a.cpp file. Then we created a second class called VectorMaxHeap where we used the standard class vector from C++ by initializing a vector called data in private and then, in public, we reimplemented all the functions from class MaxHeap in our new class VectorMaxHeap. We used an iterator to iterate through our vector in our findMax and deleteMax functions, to make sure we found and deleted the correct element.

(b) (Week 2) What is the time complexity of the five operations of `MaxHeap`? You must argue for your answers

The complete line by line Big-O analysis can be found in 4a.cpp

The heap is implemented on top of std::vector<int> and uses a linear scan (std::max_element) to find the maximum. Let N be the current number of elements. We report asymptotic time; printing does not affect complexity.

Big-O overview

- isEmpty() → checks vector::empty()
  Time: O(1).

- size() → returns vector::size()
  Time: O(1).

- insert(x) → vector::push_back(x)
  Time: amortized O(1); worst case O(N) on reallocation.

- findMax() → std::max_element(begin, end) + O(1) work
  Time: O(N) (full linear scan).

- deleteMax() → max_element (O(N)) + vector::erase(it) (O(N) shift)
  Time: O(N) overall.

Total complexity in main

The shown sequence performs three inserts (amortized O(1) each), one isEmpty() O(1), one size() O(1), then findMax() O(N), deleteMax() O(N), and findMax() O(N). Dominant terms are the three linear operations:

$$T(N) = O(N) + O$$

(5) (a) Create and implement a C++ class called *Library* that manages books. Each book (which should be a class as well) has an identifier (an integer provided when creating the book) and a category (an integer between 0 and 15 inclusive, provided when constructing the book). The library should allow adding new books, borrowing books, returning borrowed books, and displaying books available as well as borrowed. There should be a function that returns the number of books in a given category. When adding a new book, a warning (to `std::cout`) should be issued if the number of books in any **valid category** exceeds twice the average number of books per **valid category**, to ensure the user is balanced. A **valid category** is category for which books have been added to. Create a main function that tests all features of the library. The code below is provided as a **starting point that you can modify**.

Class is called 5.cpp.

(b) Justify the choice of the data structures you chose to maintain the books in the library.

We employ three different data structures to optimize for efficiency and clarity.

- An **array of fixed size (16)** tracks the number of books in each category. The number of categories is constant and known in advance, making an array the most efficient choice with O(1) access.
- An **unordered set** stores the IDs of borrowed books. A set is sufficient here since only membership (whether a book is borrowed) matters, not additional values. Lookups and inserts are O(1) on average.
- An **unordered map** associates each book ID with its corresponding `Book` object. This allows constant-time average lookups, insertions, and updates, while keeping all relevant book information (ID, category, borrowed status) together in one structure.

This combination balances fixed, predictable storage for categories with flexible, hash-based structures for fast book and borrow tracking.

(c) Provide the complexity of each operation in the library, and justify.

Line-by-line annotation of 200+ lines is unnecessary since most statements are trivial O(1). The meaningful complexity lies at the function and loop level, where operations scale with input size. Summarizing at this level avoids clutter and highlights the dominant costs, so that is what we did.

Below you can see an example of one the dominant parts of the code that dominated growth and an example of the analysis we did for all 200+ lines of code:

Example of analysis:
void addBook(int id, int category) {

```
    // Validate category logic
    if (category < 0 || category > 15) {        // O(1)  simple comparison
        cout<<"Invalid category...\n";          // O(1)  print
        return;                                 // O(1)  exit function
    }

    if (booksById.find(id) == booksById.end()){  // O(1) avg, O(N) worst
        cout<<"Book is not found...\n";          // O(1)
        Book b{ id, category};                   // O(1)  construct Book
        booksById.insert({id,b});                // O(1) avg, O(N) worst (rehash/collisions)
        incCategory(category);                   // O(1)  updates array & counter
        warnIfUnbalanced(category);              // O(1)  calls helpers
    }
    else {
        cout<<"The book is already...\n";        // O(1)
    }
}
```

Worst-case total (in `main`):

- 6 × addBook → O(N)
- 2 × borrowBook → O(N)
- 1 × returnBook → O(N)
- 3 displays → O(N)
- everything else → O(1)

Total: 12·O(N) + c·O(1) → Complexity: T(N) = O(N)

Notice we do not take into account the constants when looking at the growth asymptotically, so in the end we get a linear growth as the single dominating part of the library.cpp growth.