ALGEBRA

DATA STRUCTURES AND ALGORITHMS

Lecture 03

Learning outcome 1

ALGORITHM COMPLEXITY ANALYSIS

ALGEBRA

Introduction

- An algorithm is a well-defined set of steps that takes inputs and converts them into outputs
  - The algorithm can be written on paper, in a fictional language (pseudocode), a programming language, it can be drawn, …

```
BUBBLESORT(A)
1. for i = i to A.length – 1
2.     for j = A.length downto i + 1
3.         if A[j] < A[j – 1]
4.             exchange A[j] with A[j – 1]
```

Example taken from Introduction to Algorithms, 3rd Edition (Cormen et al)

  - The algorithm is implemented in a programming language
- The goal of algorithm complexity analysis is to say for each algorithm a) how fast it is and b) how it behaves when the number of elements to be processed increases
  - Based on that, we then make a decision on its use

An example

- How fast is the following algorithm:

```
for (int i = 0; i < n; i++) {
    if (numbers[i] == val) {
        cout << "Found it!" << endl;
        break;
    }
}
```

- Two ways of analysis:
  1. *A priori* analysis: estimating / predicting execution speed
     - An algorithm on paper is enough for this
  2. *A posteriori* analysis: measuring the duration of execution
     - For this we need a ready-made computer program

A PRIORI ANALYSIS

Running time

- **Running time** is the time required for the algorithm to run to its completion
  - We will denote it by **T(n)**, where *n* is the number of input data
  - We will express it in the number of operations, not seconds
    - Fewer operations = higher speed
  - For example, T(1000) = 2981 means that the duration of the algorithm for 1000 input data is 2981 operations

Counting the number of operations (1/2)

- To be able to estimate the execution time, we need to count how many operations will need to be performed
- We will take the previous example and start counting:

1. Initialization of a variable in a for loop: 1 operation
2. Check that $i < n$: 1 operation
3. Check if condition: 1 operation
4. If the condition is met:
   - Display: 1 operation
   - Exit from the loop: 1 operation
5. If the condition is not met:
   - Increase variable *i*: 1 operation
   - Go to step 2

Counting the number of operations (2/2)

- So what is the total operations count? What is missing to be able to determine that?
- We are missing information on how many times the loop will be executed
  - It matters if it is being executed 6 or 6,000,000 times…

Types of analysis

- In order to be able to count the operations, we need to decide what type of analysis we want to do:
  - Best case scenario
  - worst case scenario
  - Average case scenario

Best case analysis

- Our algorithm looks for the number `val` in the array
- What data must be in the array to find the `val` as quickly as possible?
  - `val` must be immediately on index 0
- Whatever the input data and whatever their number is, our algorithm will never work faster than the best case
  - We say that this is the upper limit of the speed of our algorithm, i.e. our algorithm cannot be faster than that
- In the best case, the execution speed will be: $T(n) = 5$
  - Note that it does not depend at all on the size of the array *n*
- Best case analysis is less commonly used because best practice rarely happens in practice

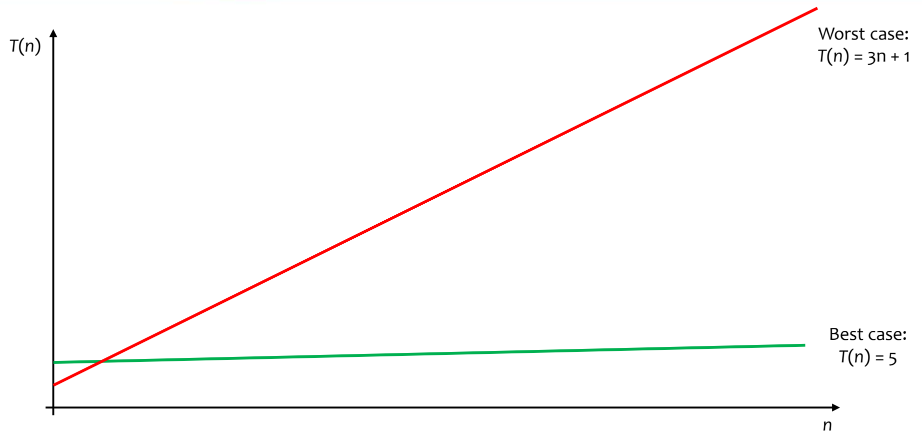
Worst case analysis

- What data must be in the array to find the `val` number at the latest?
  - The `val` number must not exist in the array at all
- Whatever the input data and whatever their number is, our algorithm will never run slower than the worst case
  - We say that this is the lower limit of the speed of our algorithm
  - Our algorithm cannot be slower than that
- In the worst case, the execution speed will be: $T(n) = 3n + 1$
  - Note the linear dependence on the array size *n*
- Worst case analysis is very often used in practice
  - If you are the seller of your algorithm, this is a guarantee to the buyer

Average case analysis

- It might be reasonable to assume that we will find the number `val` on average in the middle of the array
  - Sometimes we will find it closer to the beginning, sometimes closer to the end or we will not find it at all
- In the average case, the execution speed is: $T(n) = 3n / 2 + 3$
  - Note again the linear dependence on the array size *n*
- However, if our initial assumption is not correct, neither is the average case accurately calculated
  - In practice, it is usually difficult to calculate the average case
  - It often has the same complexity as the worst case
  - We will not consider the average case in the rest of the lecture

Graphical representation of best and worst case



- Our algorithm is guaranteed to be below the top and above the bottom line, no matter how big *n* is and what the input data is (after a certain point n_0)

Simplifying the counting of operations (1/2)

- For the complexity analysis, it is not important to know the exact number of operations, but to understand their dependence on *n*
- Let's analyze the best and worst case of the algorithm by an approximating the number of operations and draw a graph for *n* equal to 1, 10 and 100:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        cout << i << " * " << j << " = " << i * j << endl;
    }
}
```

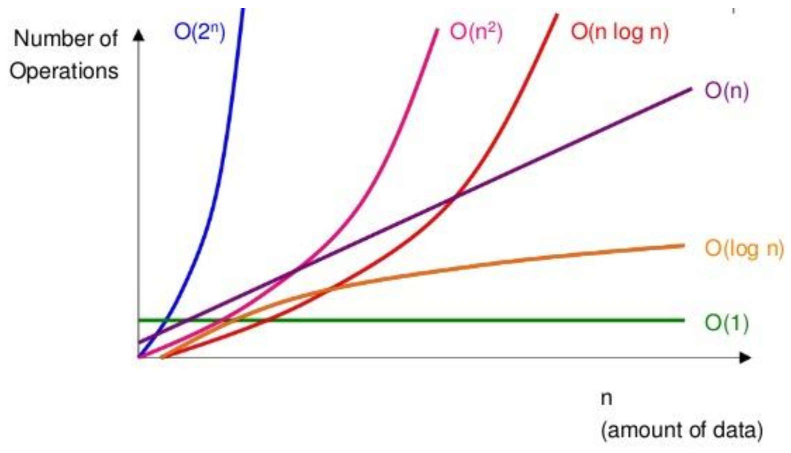
Simplifying the counting of operations (2/2)

- Note that the execution time increases squarely with the number of input data
  - We do not care whether it is T(1000) = 1,000,000 or T(1000) = 1,000,094

Predefined functions

- If one algorithm has in the worst case equal to $\mathrm{T}(n) = \left(\frac{n+4}{2n-7}\right)^{n+1}$, and the other $\mathrm{T}(n) = \frac{3n+\sqrt{n^2-4n-7}}{2n+3}$, which one is faster?
- To make it easier to compare algorithms, we will use a few simple predefined functions:
  - $f(n) = 1$
  - $f(n) = \log n$
  - $f(n) = n$
  - $f(n) = n \log n$
  - $f(n) = n^2$ (or n^3, n^4, …)
  - $f(n) = 2^n$
  - $f(n) = n!$

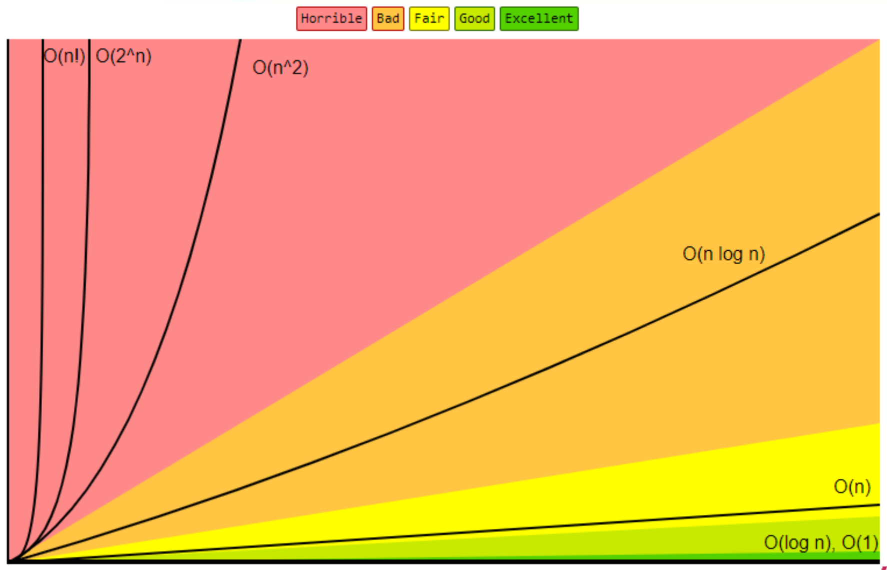
Functions relationships – view 1



(C) 2010 Thomas J Cortina, Carnegie Mellon University

Strana ▪ 17

Functions relationships – view 2



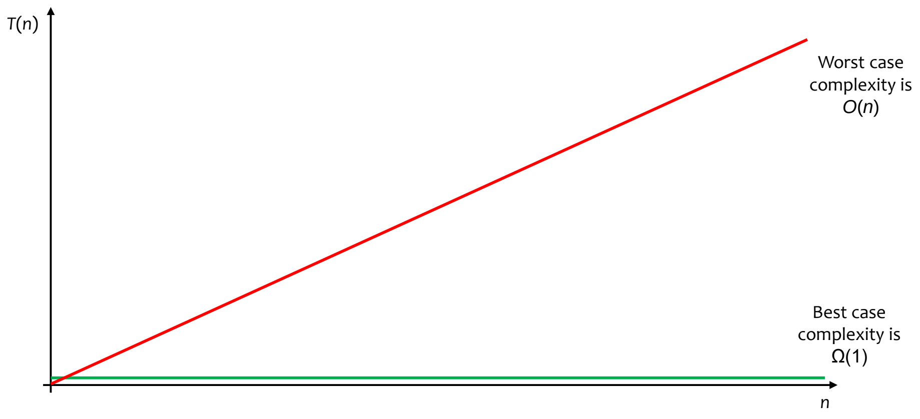
Strana ▪ 18

Taken from bigocheatsheet.com

Marking the best and worst case

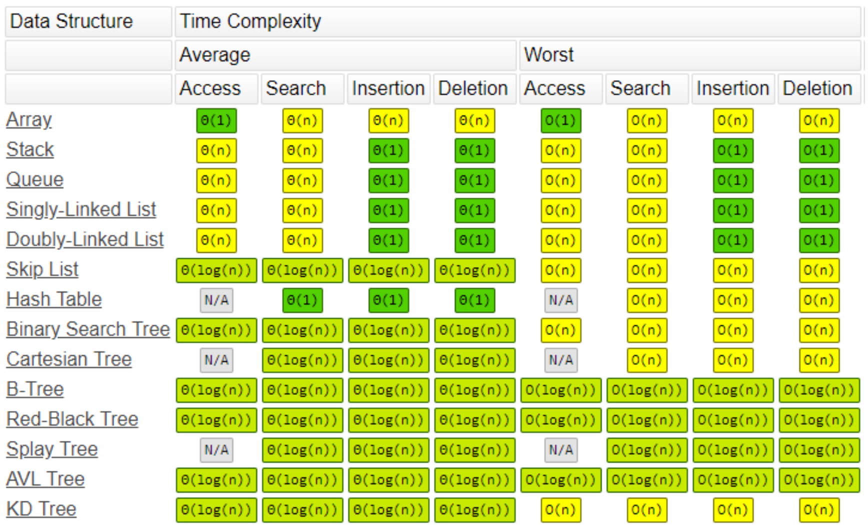
- To describe our algorithm in an understandable, standard way, we proceed as follows:
  - Best case: we take the closest predefined function f_1 that is still below our execution time and say that our algorithm has complexity $\Omega(f_1)$
  - Worst case: we take the closest predefined function f_2 that is still above our execution time and say that our algorithm has complexity $O(f_2)$
- We say we use a group of notations known as Bachmann-Landau notations or asymptotic notations

Standard presentation of best and worst case



- Our algorithm is guaranteed to be below the top and above the bottom line, no matter how big *n* is and what the input data is (after a certain point n_0)

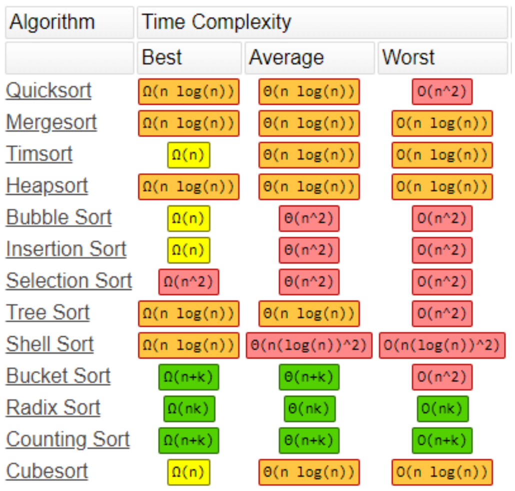
Complexity of standard container operations



Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)

Taken from bigocheatsheet.com

Complexity of standard sorting algorithms



Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))

Taken from bigocheatsheet.com

Problems

- Problems from the *a priori* analysis of the algorithm complexity will be based on the interpretation of the previous two tables
  - Practically speaking, this is expected of every developer
  - Frequently asked question at a job interview
- Examples of problems (argument your answer):
  1. If you want the fastest average insertion, which container will you choose?
  2. Is BUBBLESORT or HEAPSORT faster in the worst case?
  3. Which container finds the required value the fastest on average?

ALGEBRA

***A POSTERIORI* ANALYSIS**

ALGEBRA

Introduction

- We designed the algorithm and implemented it in the program
  - We also assessed its complexity
- *A posteriori* analysis is an analysis of the execution of a program on a real computer
- We use time instead of logical operations
- We measure the length of program execution
  - The shorter the execution time, the better we consider the program

Namespaces

- Data types can be logically organized into namespaces
  - Data type `string` is in the namespace `std`
  - Data type `Rectangle` is not in the namespace
- Namespaces can be nested in each other
- When using a data type we have two options:

1. Using "`using namespace`", we import the entire namespace and then use the data type by name
2. We omit "`using namespace`" and put the namespace prefix in front of the data type name, i.e. `std::cout`

Example

- Option 1:

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string name = "Mirko";
    cout << name << endl;
    return 0;
}
```

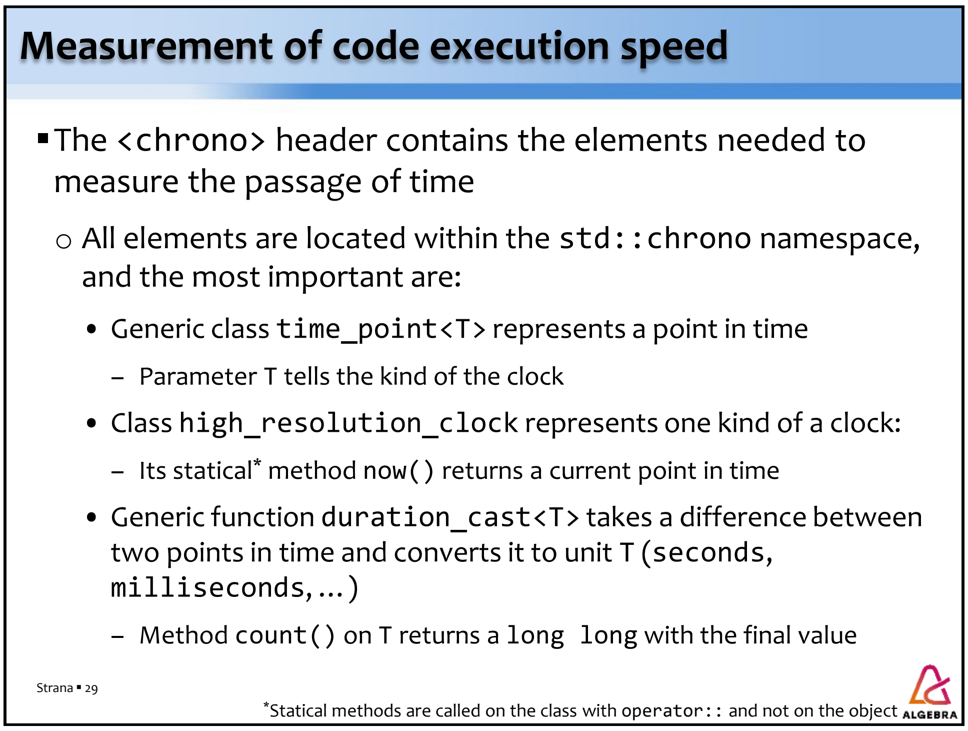
- Option 2:

```
#include <iostream>
#include <string>

int main() {
    std::string name = "Mirko";
    std::cout << name << std::endl;
    return 0;
}
```

* Example of own namespaces

```
namespace my1 {
   namespace my2 {
        class Rectangle {
        private:
             int width;
             int height;
        public:
             Rectangle(int width, int height);
             void multiply(int scalar);
             int area();
             int perimeter();
             double diagonal();
             void draw();
        };
   }
}
```

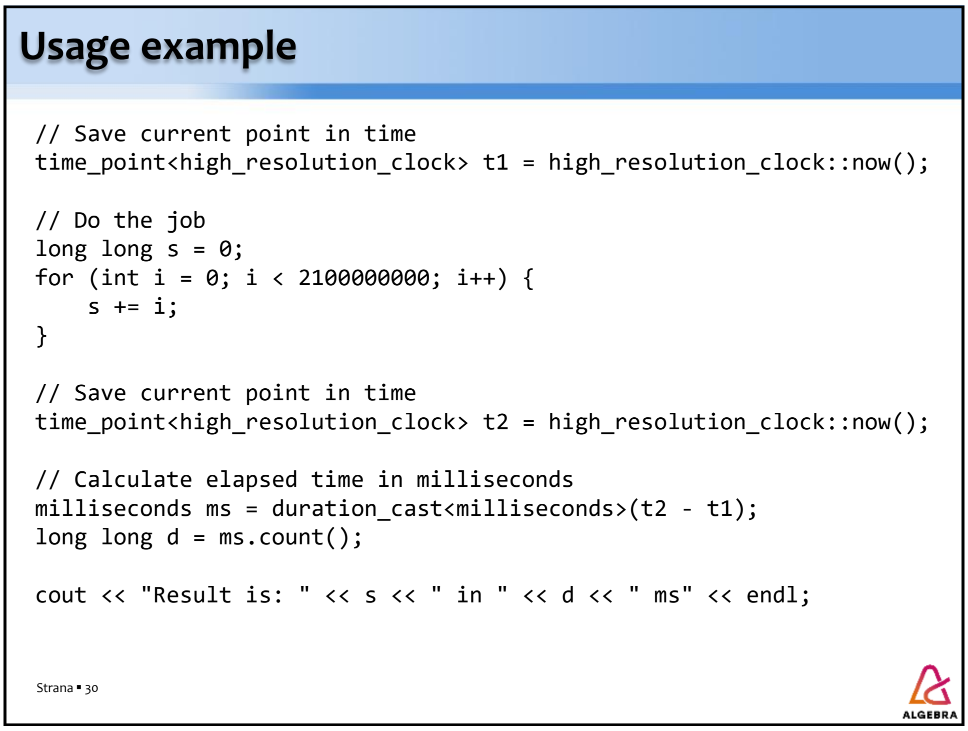


Measurement of code execution speed

- The `<chrono>` header contains the elements needed to measure the passage of time
  - All elements are located within the `std::chrono` namespace, and the most important are:
    - Generic class `time_point<T>` represents a point in time
      - Parameter T tells the kind of the clock
    - Class `high_resolution_clock` represents one kind of a clock:
      - Its statical* method `now()` returns a current point in time
    - Generic function `duration_cast<T>` takes a difference between two points in time and converts it to unit T (`seconds`, `milliseconds`, …)
      - Method `count()` on T returns a `long long` with the final value

Strana ▪ 29

*Statical methods are called on the class with `operator::` and not on the object



Usage example

```
// Save current point in time
time_point<high_resolution_clock> t1 = high_resolution_clock::now();

// Do the job
long long s = 0;
for (int i = 0; i < 2100000000; i++) {
    s += i;
}

// Save current point in time
time_point<high_resolution_clock> t2 = high_resolution_clock::now();

// Calculate elapsed time in milliseconds
milliseconds ms = duration_cast<milliseconds>(t2 - t1);
long long d = ms.count();

cout << "Result is: " << s << " in " << d << " ms" << endl;
```

Strana ▪ 30

Wrapping it up in the class (1/3)

- Stopwatch.h

```
#pragma once
#include <chrono>

class Stopwatch {
private:
    std::chrono::time_point<std::chrono::high_resolution_clock> t1;
    std::chrono::time_point<std::chrono::high_resolution_clock> t2;

public:
    void start();
    void stop();
    long long get_elapsed_milliseconds();
};
```

Wrapping it up in the class (2/3)

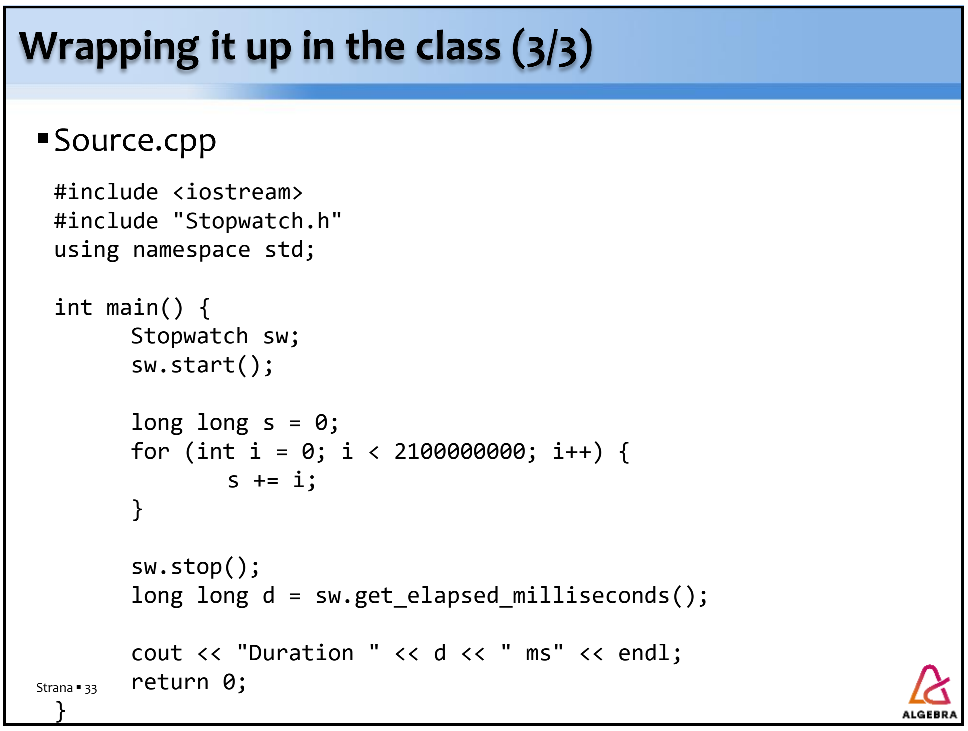
- Stopwatch.cpp

```
#include "Stopwatch.h"

void Stopwatch::start() {
    t1 = std::chrono::high_resolution_clock::now();
}

void Stopwatch::stop() {
    t2 = std::chrono::high_resolution_clock::now();
}

long long Stopwatch::get_elapsed_milliseconds() {
    return
        std::chrono::duration_cast<std::chrono::milliseconds>(t2-t1)
                                                          .count();
}
```



Wrapping it up in the class (3/3)

- Source.cpp

```
#include <iostream>
#include "Stopwatch.h"
using namespace std;

int main() {
      Stopwatch sw;
      sw.start();

      long long s = 0;
      for (int i = 0; i < 2100000000; i++) {
              s += i;
      }

      sw.stop();
      long long d = sw.get_elapsed_milliseconds();

      cout << "Duration " << d << " ms" << endl;
      return 0;
}
```