FA Laboratory

Raluca Laura Portase, Eugen-Richard Ardelean

September 28, 2025

Contents

1 Laboratory Guide **6**
- 1.1 **About the laboratory** . . . 6
- 1.2 **Laboratory Format** . . . 6
  - 1.2.1 **Rules** . . . 6
  - 1.2.2 **Grading** . . . 6
  - 1.2.3 **Assignments Delivery** . . . 7
  - 1.2.4 **Algorithm complexity evaluation** . . . 7
  - 1.2.5 **Delivery deadline** . . . 8
  - 1.2.6 **Attempted Fraud** . . . 8
- 1.3 **Laboratory session transfer** . . . 8
- 1.4 **Suggested literature** . . . 9
- 1.5 **Profiler** . . . 9

2 Intro session **10**
- 2.1 Microsoft Visual Studio . . . 10
  - 2.1.1 Install . . . 10
  - 2.1.2 Use . . . 10
- 2.2 JetBrains CLion . . . 13
  - 2.2.1 Install . . . 13
  - 2.2.2 Use . . . 14
- 2.3 C/C++ . . . 16
  - 2.3.1 Reading/writing files . . . 16
  - 2.3.2 Generating test cases . . . 16
  - 2.3.3 Generating plots . . . 16
  - 2.3.4 Microsoft Office Excel . . . 17
  - 2.3.5 Exercise . . . 19

3 Assignment No. 1: Analysis & Comparison of Direct Sorting Methods **20**
- 3.1 Implementation . . . 20
- 3.2 Requirements . . . 20
  - 3.2.1 Implementation of direct sorting method (5.5p) . . . 20

3.2.2 Evaluate algorithms for the average case (1.5p – 0.5p for each algorithm) . . . 21
3.2.3 Evaluate algorithm for best and worst case (3p - 0.5p for each case of each algorithm) . . . 21
3.2.4 Bonus: Binary insertion sort (0.5p) . . . 21

4 Assignment No. 2: Analysis & Comparison of Bottom-up and Top-down Build Heap Approaches 21
4.1 Implementation . . . 22
4.2 Minimal requirements for grading . . . 22
4.3 Requirements . . . 22
4.3.1 Comparative analysis of *one* of the sorting algorithms from L1 (you choose) in *iterative* vs *recursive* version. The analysis should be performed based on the *number of operations and the runtime* (2p) . . . 22
4.3.2 Implement bottomup build heap procedure (2p) . . . 23
4.3.3 Implement topdown build heap procedure (2p) . . . 23
4.3.4 Comparative analysis of the two build heap methods in the average case (2p) . . . 23
4.3.5 Comparative analysis of the two build heap methods in the worst case (1p) . . . 23
4.3.6 Implement and exemplify correctness of heapsort (1p) . . . 23

5 Assignment No. 3: Analysis & Comparison of Advanced Sorting Methods – Heapsort and Quicksort / QuickSelect 24
5.1 Implementation . . . 24
5.2 Minimal requirements for grading . . . 24
5.3 Requirements . . . 24
5.3.1 QuickSort: implementation (2p) . . . 24
5.3.2 QuickSort: average, best and worst case analysis (3p) . . . 25
5.3.3 Quicksort and Heapsort: comparative analysis of average case (2p) . . . 25
5.3.4 Implementation of quicksort hybridization (1p) . . . 25
5.3.5 Determination of an optimal threshold used in hybridization + proof (graphics/ measurements) (1p) . . . 25
5.3.6 Comparative analysis (between *quicksort and quicksort hibridization*) from the operations and runtime perspective (1p) . . . 26
5.3.7 Bonus: QuickSelect - Randomized-Select (0.5p) . . . 26

6 Assignment No. 4: Merge k Ordered Lists Efficiently 26
6.1 Implementation . . . 26
6.2 Minimal requirements for grading . . . 27
6.3 Requirements . . . 27

6.3.1 Generate k randomly sized sorted lists (having n elements in total, n and k given as parameters) and the merging of 2 lists (5p) . . . 27
6.3.2 Adapt *min-heap* operations to work on the new structure and the merging of k lists (3p) . . . 27
6.3.3 Evaluation of the algorithm in average case (2p) . . . 27

7 Assignment No. 5: Search Operation in Hash Tables 28
7.1 Implementation . . . 28
7.1.1 Hashing (refers to the hash table) . . . 28
7.1.2 Addressing (refers to the final position of the element with respect to its initial position) . . . 28
7.2 Minimal requirements for grading . . . 29
7.3 Requirements . . . 29
7.3.1 Implementation of the insert and search operations using the required data structure (5p) . . . 29
7.3.2 Evaluate the search operation for a single fill factor 95% (2p) . . . 29
7.3.3 Complete evaluation for all fill factors (2p) . . . 30
7.3.4 Implement delete operation in a hash table, *demo (size 10)* and evaluation of the search operation after deletion of some elements (1p) . . . 31

8 Assignment No. 6: Multi-way Trees 31
8.1 Implementation . . . 31
8.2 Requirements . . . 31
8.2.1 Implementation of *iterative* and *recursive* binary tree traversal in *O(n)* and *with constant additional memory* (3p) . . . 31
8.2.2 Implementation of transforms between different representations . . . 32
8.2.3 Correct implementation for Pretty-print for R1 (2p) . . . 32
8.2.4 Correct implementation for *T1 (from R1 to R2)* and pretty-print for *R2* (1p) + *T1* in linear time (1p) . . . 32
8.2.5 Correct implementation for *T2 (from R2 to R3)* and pretty-print for *R3* (2p) + *T2* in linear time (1p) . . . 32

9 Assignment No. 7: Dynamic Order Statistics 32
9.1 Implementation . . . 32
9.2 Requirements . . . 33
9.2.1 BUILD_TREE: correct and efficient implementation (5p) 33
9.2.2 OS_SELECT: correct and efficient implementation (1p) . 33
9.2.3 OS_DELETE: correct and efficient implementation (2p) . 34
9.2.4 Management operations evaluation - BUILD, SELECT, DELETE (2p) . . . 34
9.2.5 Bonus: Implementation using AVL / Red black tree (1p) 34

10 Assignment No. 8: Disjoint Sets **34**
10.1 Implementation . . . 34
10.2 Requirements . . . 35
10.2.1 Correct implementation of MAKE_SET, UNION and FIND_SET (5p) . . . 35
10.2.2 Correct and efficient implementation for Kruskal's algorithm (2p) . . . 35
10.2.3 Evaluate the disjoint sets operations (MAKE, UNION, FIND) using Kruskal's algorithm (3p) . . . 35

11 Assignment No. 9. BFS: Breadth-First Search **36**
11.1 Introduction . . . 36
11.2 The structure of the boiler plate code . . . 36
11.2.1 Project setup for Windows, with Visual Studio . . . 36
11.2.2 Project setup for Linux and Mac . . . 37
11.3 Running the program . . . 37
11.3.1 Example: command `neighb` . . . 38
11.3.2 Example: commands `bfs` and `bfs_step` . . . 39
11.3.3 Example: command `bfs_tree` . . . 39
11.3.4 Example: command `path` . . . 39
11.3.5 Employed data structures . . . 40
11.4 Requirements . . . 41
11.4.1 Determine the neighbors of a cell (2p) . . . 41
11.4.2 BFS algorithm implementation (3p) . . . 42
11.4.3 Pretty printing the BFS tree (2p) . . . 42
11.4.4 Evaluate the performance of BFS (3p) . . . 42
11.4.5 Bonus: Determine the shortest path (0.5p) . . . 42
11.4.6 Bonus: Where can a knight end up on the board? (0.5p) . 43

12 Assignment 9: Tutorial **43**
12.1 Visual Studio Project Setup . . . 43
12.2 Visual Studio 'unsafe' error . . . 46
12.3 Visual Studio 'Assertion' Error . . . 51
12.4 CLion undefined Error . . . 52
12.5 CLion Visual Studio – Option 1 (slower) . . . 53
12.6 CLion MinGW – Option 2 (faster, requires external console) . . . 56
12.6.1 CLion Clear error . . . 56
12.6.2 CLion not showing grid . . . 58
12.7 Mac run command . . . 59

13 Assignment 10: Depth-first search (DFS) **59**
13.1 Implementation . . . 59
13.2 Requirements . . . 60
13.2.1 DFS (5p) . . . 60
13.2.2 Topological sort (1p) . . . 60
13.2.3 Tarjan (2p) . . . 60

13.2.4 Analysis of the DFS performance (2p) 60

14 Error Fix **61**
14.1 Profiler + VS Code Error 61

References **63**

1 Laboratory Guide

1.1 About the laboratory

This semester, you will implement a series of algorithms and you will analyze the *correctness* and the *efficiency* of your own implementations. The implementations will have a starting point the pseudocode that is provided at the **course** and **seminary** sessions. You can write your code in C or C++ or another language that you are familiar with as long as you implement all the data structures that are required.

1.2 Laboratory Format

1.2.1 Rules

- *Attendance* is **mandatory**
- *One* absence can be recovered (the corresponding assignment MUST be *delivered* and *verified* the following week)
- A second absence can be recovered in the special laboratory session at the end of the semester (surcharge) by solving additional assignments (solving the corresponding assignment of the missed session will NOT be considered)
- If you miss more than 2 sessions, **you will not be able to attend the exam in the regular exam session**
- **IN EXCEPTIONAL CASES**, you can attend another laboratory sessions in the same week with another group *if* you announce your laboratory assistant on email and you receive his agreement; the assignment will be delivered to and verified by the laboratory assistant of the sessions you are signed up for (the assignment has to be uploaded on time and will be presented the following week at the laboratory session)

1.2.2 Grading

- The laboratory grade is equal to **30%** of the final grade
- The laboratory grade is composed of two parts: **assignments grade** - 2/3 of lab grade (that is **20%** of final grade) and **colloquy/lab exam** - 1/3 of lab grade (that is **10%** of final grade)
- Assignments:
  - Each assignment has the same weight in the calculation of the assignments grade - unweighted arithmetic mean formula is used
  - Each assignment has 4 grading thresholds for the following grades: 5, 7, 9 and 10 (the requirements for each grade can be viewed in the laboratory assignment documentation)

- Colloquy:
  - The colloquy consists of a closed book lab test in the last week of the semester (W14)
  - The colloquy must be held by each student on the computers available in the laboratory room (no personal laptops / computers)
- In order to pass the laboratory and be granted participation in the exam you need **both**:
  - Assignments grade ≥ 5
  - Colloquy grade ≥ 5

1.2.3 Assignments Delivery

- At the assessment discussion you will present: **source code, charts and runnable demo code**
- **Source code** (program.cpp) and the **charts MUST** be uploaded to Moodle, in the archive (program.zip), **before the lab session**
- *Unindented* code will **not be evaluated**
- If you cannot explain the used algorithms, the assignment will **not be evaluated**
- Each source file must contain at the beginning a comment of the following format:

```
/**
* @author John Smith
* @group 30xyz
*
* Assignment requirements, ex: Compare the sorting methods X and Y
*
* Personal interpretation of the (time and space) complexity of the average, best and worst testing Interpretarea
* cases. For example: "Method X has a time complexity of Y in case Z because ..."
*/
```

1.2.4 Algorithm complexity evaluation

- For the average case, you need to repeat the measurements *at least 5 times*
- Measure the number of operations made by the algorithm (the assignments and the comparisons of the input data or the auxiliary variables that contain input data)

- Vary the dimensions of the input data consistent with the requirements of each assignment
- Apply the same input data on each algorithm when making comparative evaluations (for the average case)
- Generate evaluation charts (either in **Excel** or by using **Profiler**)
- Analyze the charts and add your personal observations at the beginning of the source code using the above mentioned format

1.2.5 Delivery deadline

Assignments can be delivered:

- During the laboratory sessions that they are presented in. At the end of the laboratory you MUST upload to Moodle a draft version of the current assignment (containing what you managed to implement during the session). A lack of a submission at the end of the session (or a lack of relevant code) will receive of penalty of up to 2 points of the grade of that assignment
- **Extension_1** (E1): at the beginning of the next laboratory session
- **Extension_2** (E2): specific assignments can be delivered at the beginning of the second laboratory after the assignment was presented (with a penalty of -2).
- Starting with the third laboratory after the assignment was presented, the assignment cannot be delivered anymore (will be graded as 0)

You can find a planning of the assignment and their extensions on Moodle (file name: Planificare Saptamanala (bachelor))

Each assignment MUST be uploaded to Moodle before the beginning of the laboratory session during which it will be delivered.

1.2.6 Attempted Fraud

For the first uncovered attempt at fraud (copying someone else's code or using code generated by AI tools) you will receive a penalty of 10 points of the total accumulated points (until that moment). For a subsequent attempt, you will have to **retake the course next year**.

1.3 Laboratory session transfer

If you wish to participate at laboratory sessions with another assistant you must follow these rules:

- Student S1 from group G1 can transfer to group G2 if and only if a student S2 from group G2 can be found that is willing to participate with G1 at the assigned laboratory session hours of G1.

To formalize the transfer you must send an email in which you mention with whom are you making the transfer/exchange:

- An email from S1 to both laboratory assistants
- An email from S2 to both laboratory assistants

The laboratory session transfer deadline is the **end of the second week** of the semester.

1.4 Suggested literature

- Cormen, T. H. et al (2009). Introduction to algorithms. MIT press
- J. Kleinberg, E. Tardos (2005). Algorithm Design. Addison Wesley
- C/C++ Tutorials
  - http://www.cprogramming.com/begin.html
  - http://www.learn-c.org
  - Accelerated C++: Practical Programming by Example
- Coding Styles
  - http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html
  - http://www.cs.swarthmore.edu/~newhall/unixhelp/c_codestyle.html
  - http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

1.5 Profiler

The library that will be utilized for the generation of plots, each student must go through the given example and tutorial.

The most recent version can be found here:
https://github.com/cypryoprisa/utcn-fa-profiler

Profiler Tutorial:
- Part 1
- Part 2
- Part 3

2 Intro session

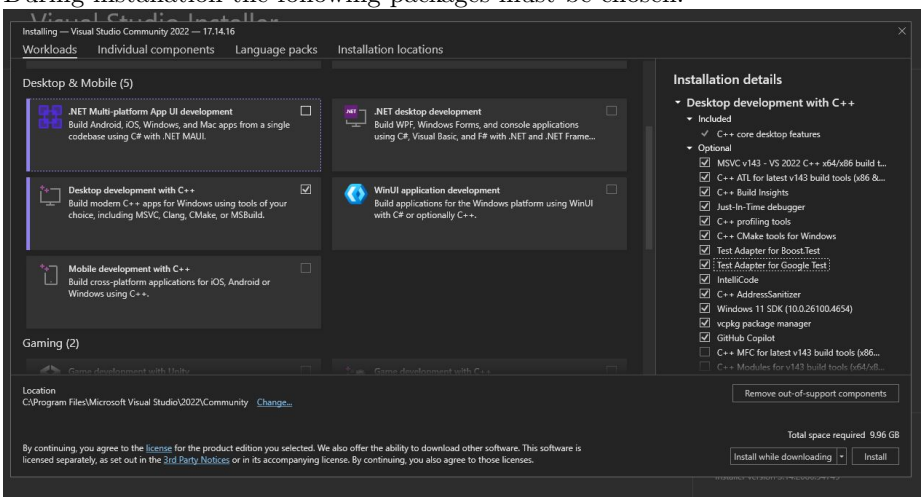
To begin with, make sure you have read the Laboratory Guide on Moodle. In this lab, you will learn how to write a C/C++ program in Microsoft Visual Studio or JetBrains CLion. You will also learn how to generate data for evaluating algorithms and how to create graphs in Microsoft Office Excel.

2.1 Microsoft Visual Studio

2.1.1 Install

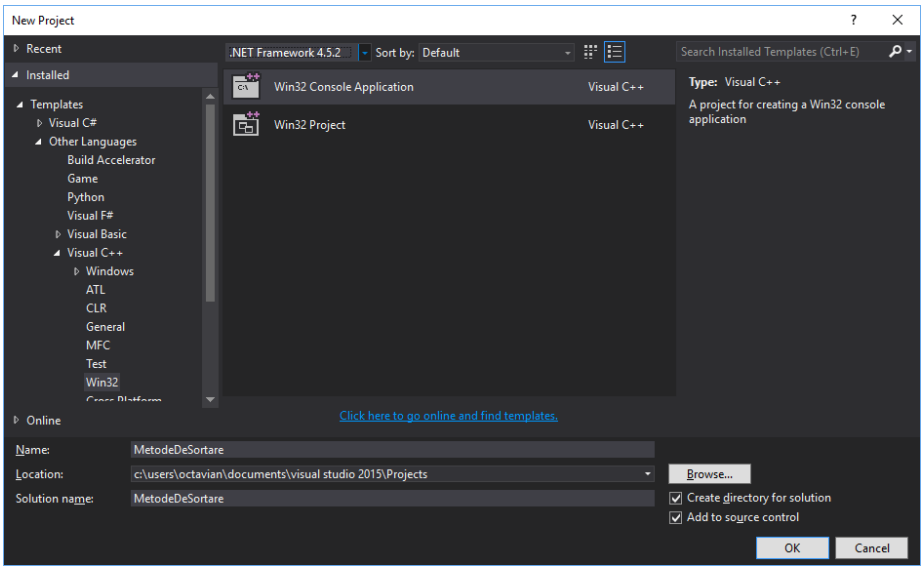
Free edition: `https://visualstudio.microsoft.com/vs/community/`

During installation the following packages must be chosen:

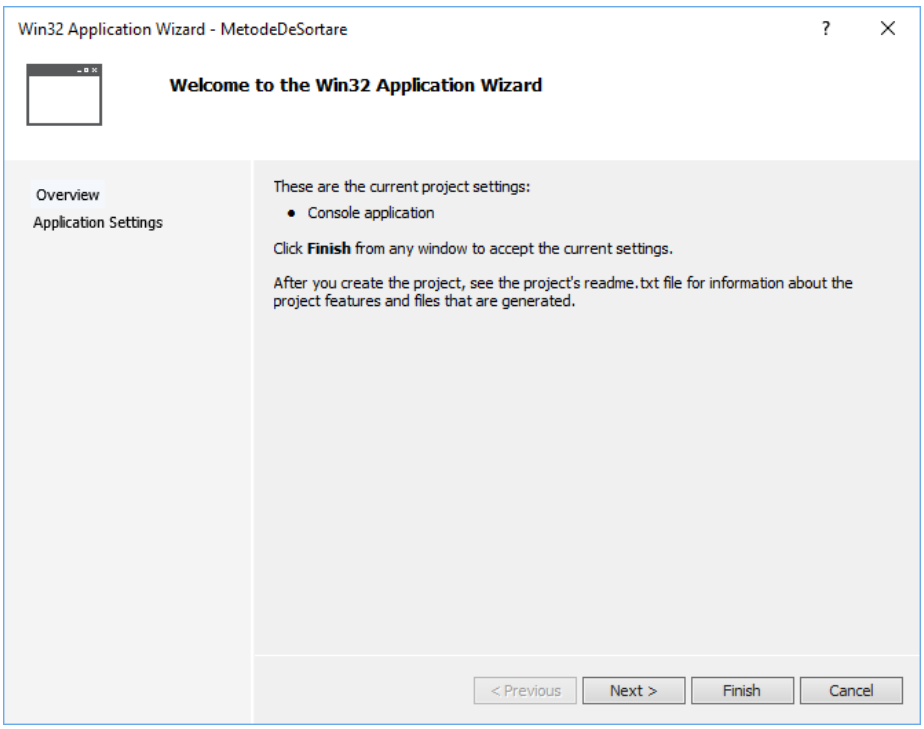


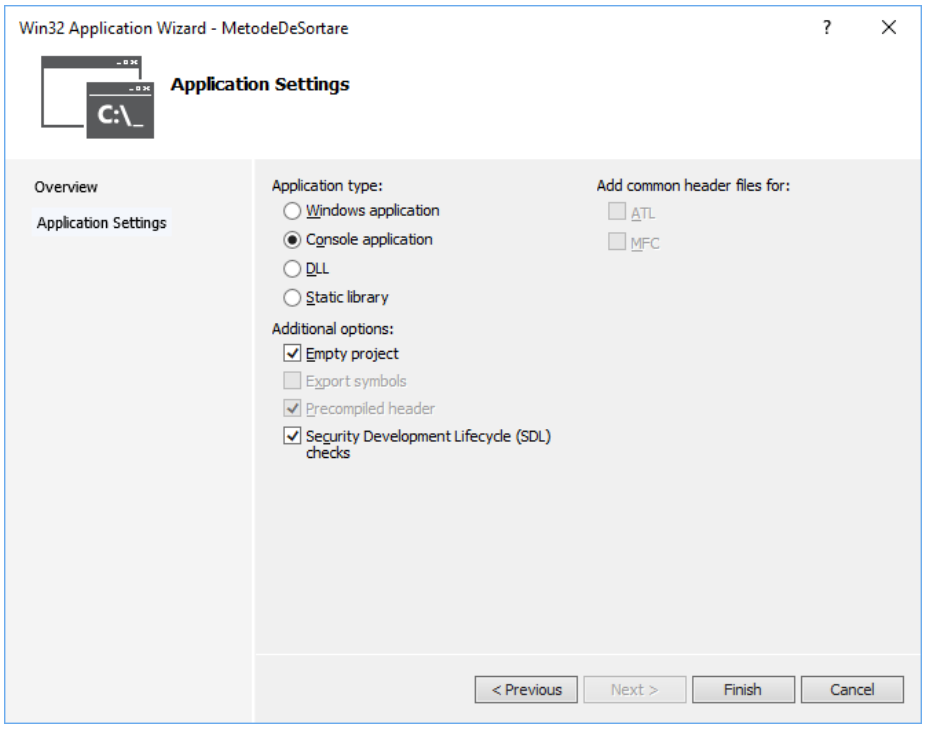
2.1.2 Use

1. Create project: *File -> New -> Project...*
2. Project type: *Templates -> Other Languages -> Visual C++ -> Win32 -> Win32 Console App*

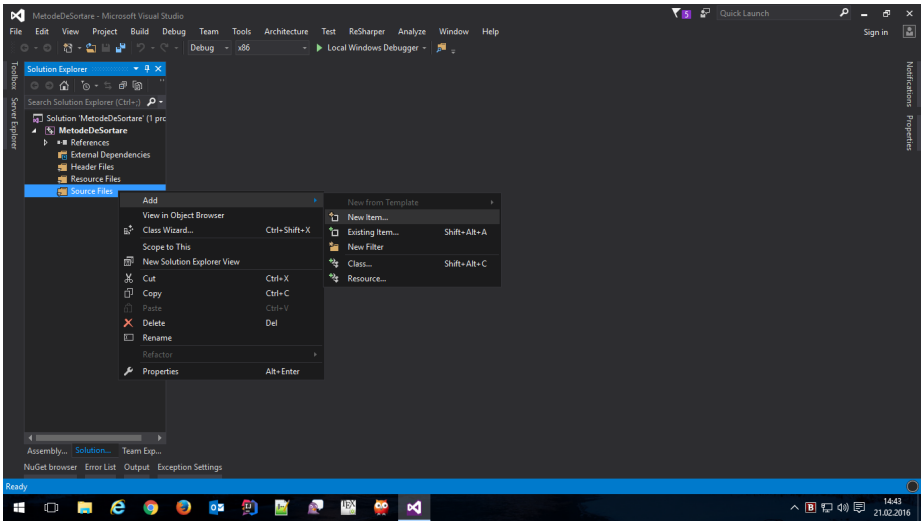


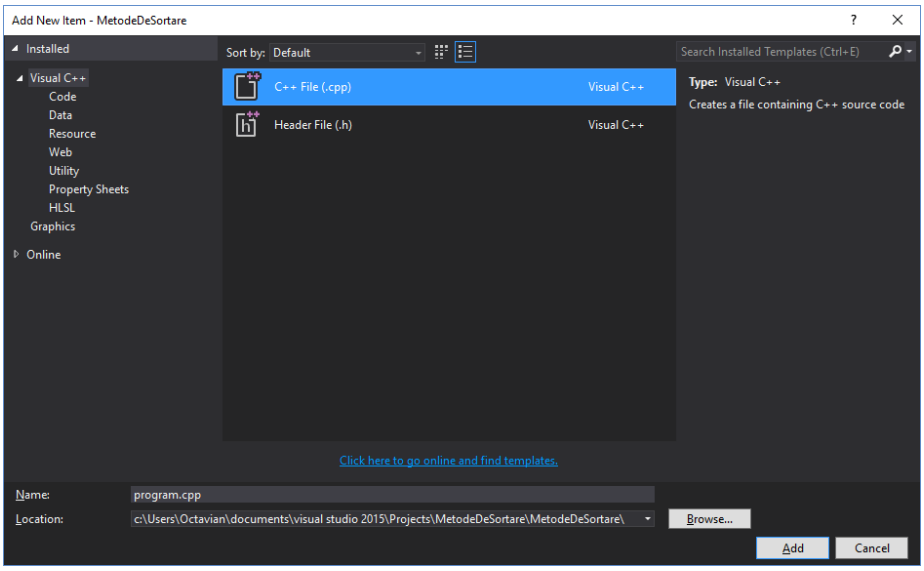
3. Project properties: choose "*Console application*" and select "*Empty project*"



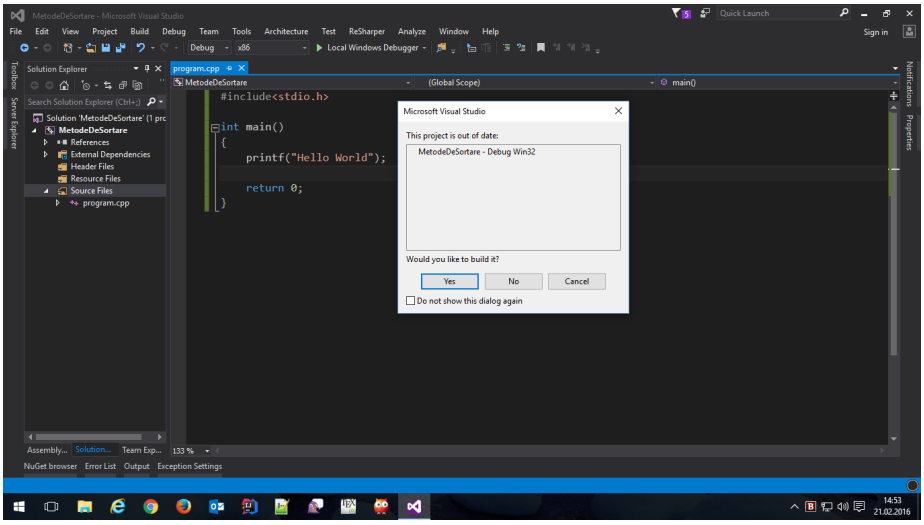


4. Create a **.cpp* file





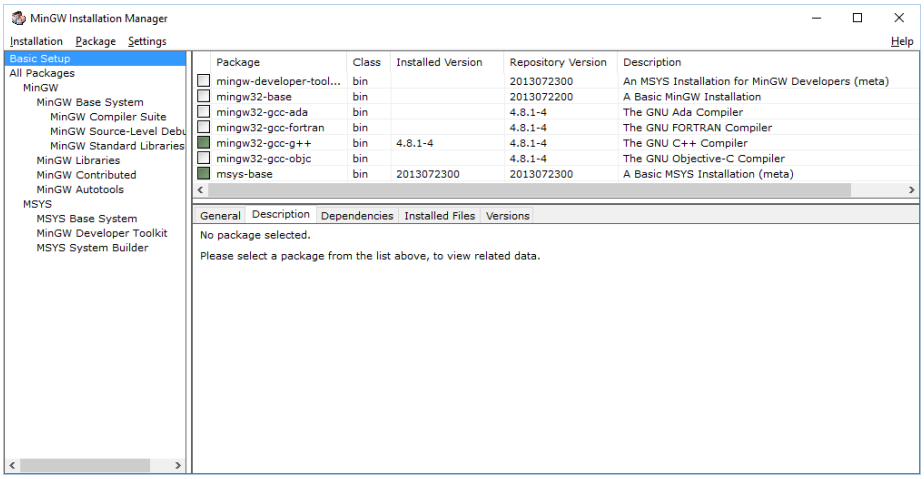
5. Compile and execute the program (in **DEBUG** mode)



2.2 JetBrains CLion

2.2.1 Install

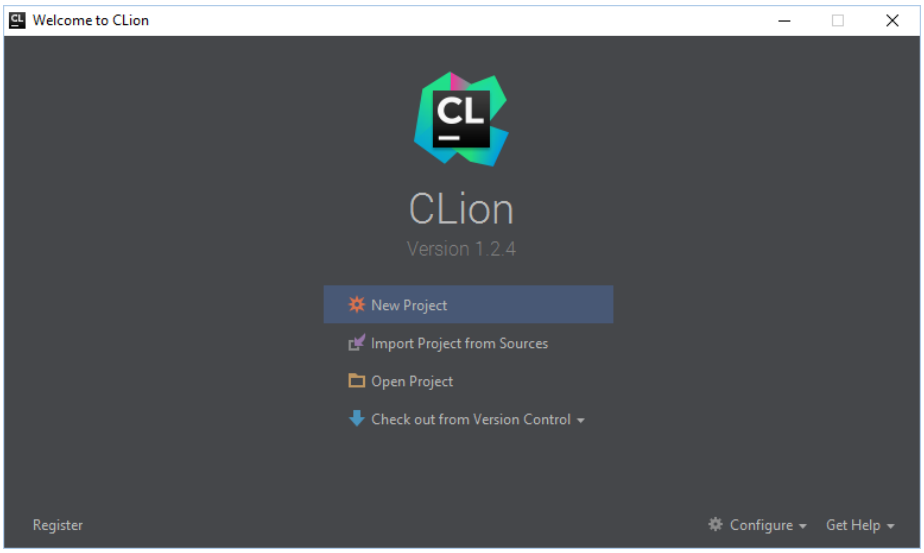
Register with your student e-mail address: *@student.utcluj.ro* on https://www.jetbrains.com/student/
Download and install MinGW: https://sourceforge.net/projects/mingw/



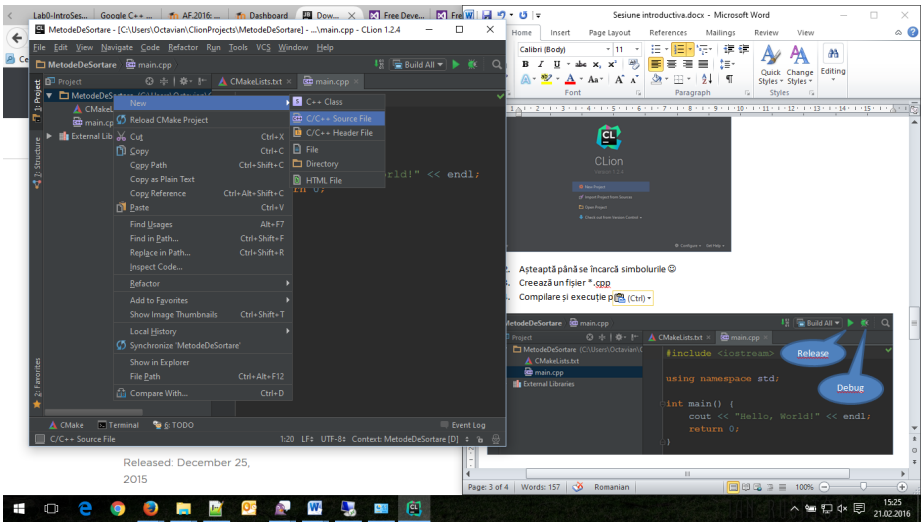
Download and install CLion: `https://www.jetbrains.com/clion/download/#section=windows`

2.2.2 Use

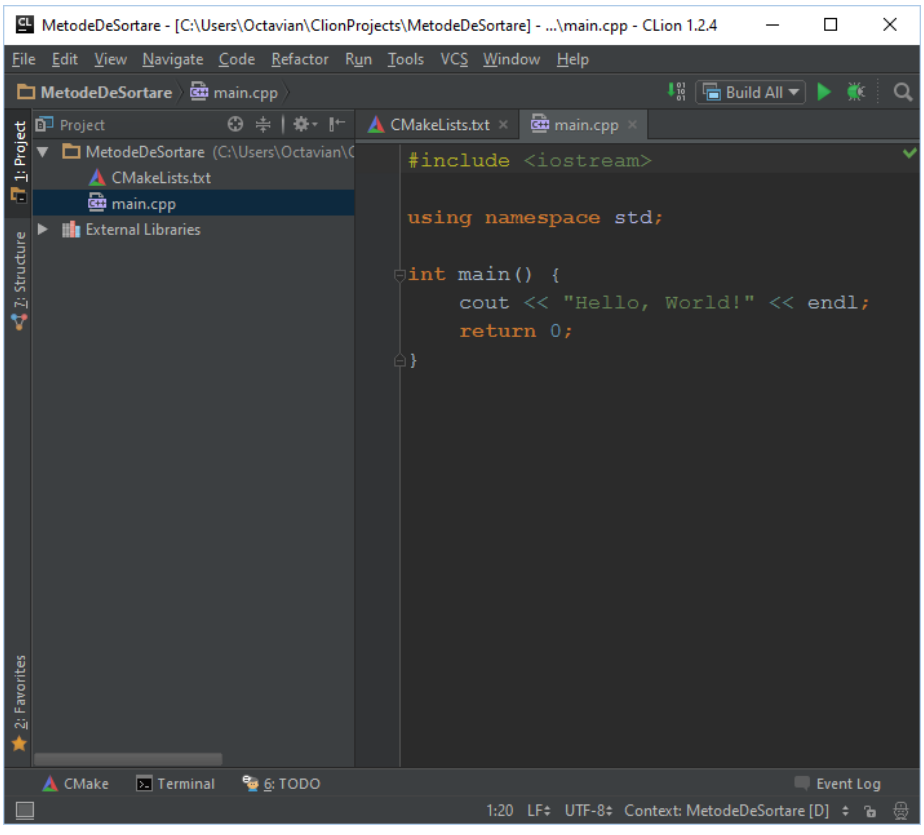
1. Create project: *New Project*



2. Wait until symbols are loaded
3. Create a **.cpp* file



4. Compile and execute program



2.3 C/C++

2.3.1 Reading/writing files

Exercise - steps:

- Declare an array v of length *MAX_SIZE* (a constant defined by you)
- Read an n from the keyboard
- Open the *input.txt* file, read n numbers from it and save them in array v
- Save the n numbers in the *output.txt* file in **reverse** order

2.3.2 Generating test cases

To test the algorithms you will implement, you will need to use a series of input data: sorted ascending arrays, sorted descending arrays, random arrays, etc. Generating ascending/descending arrays should be straightforward. For generating random arrays, you can use the following:

- *Profiler* library from Moodle (or `https://github.com/cypryoprisa/utcn-fa-profiler`)
- *rand*(), *srand*() methods, read:
  - `http://www.cplusplus.com/reference/cstdlib/rand/`
  - `http://www.cplusplus.com/reference/cstdlib/srand/`
  - `http://www.cplusplus.com/reference/cstdlib/RAND_MAX/`

Exercise - steps:

- Read n, *min* and *max* from the keyboard
- Generate a random array of n elements with values bounded within *min* and *max*
- The array must be different for each execution of the program
- Add the array to the *output.txt* file

2.3.3 Generating plots

For the generation of the charts, you can use:

- *Profiler* library from Moodle (or `https://github.com/cypryoprisa/utcn-fa-profiler`)
- Microsoft Office Excel

2.3.4 Microsoft Office Excel

You will need to create a file with the *.csv* (comma-separated values) extension. The file should have a structure similar to the following:

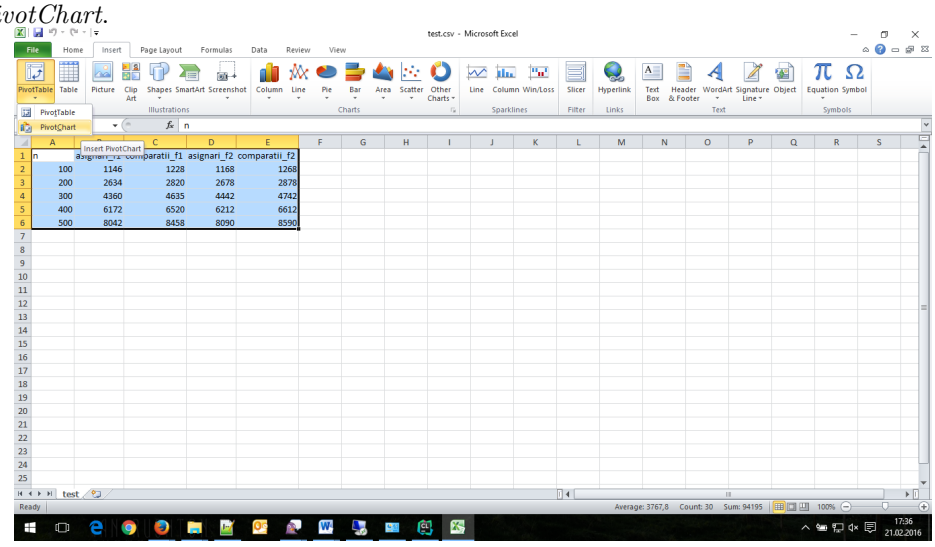
```
n,assignments_f1,comparisons_f1,assignments_f2,comparisons_f2
100,1146,1228,1168,1268
200,2634,2820,2678,2878
300,4360,4635,4442,4742
400,6172,6520,6212,6612
500,8042,8458,8090,8590
```

Legend:

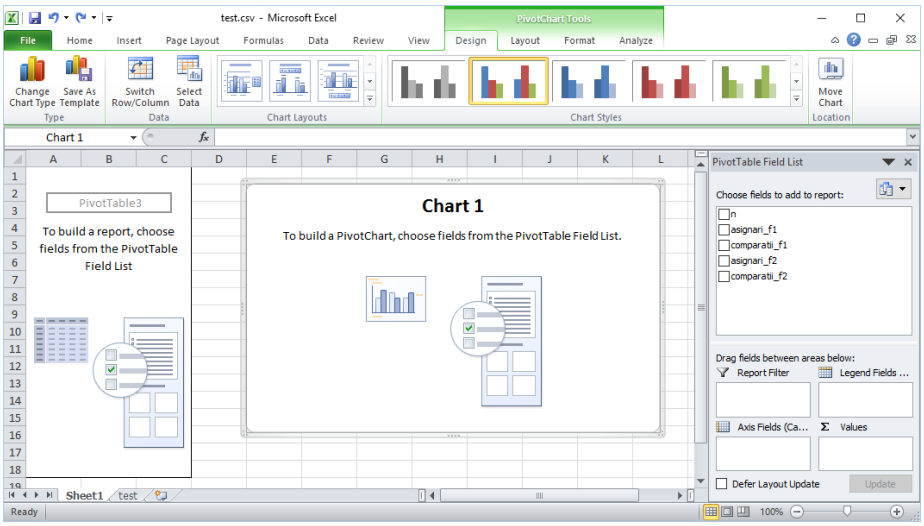
- n= the size of the problem (ex: lungimea șirului de intrare)
- assignments_f1= the number of assignments for the best-case scenario of method 1
- comparisons_f2= the number of assignments for the best-case scenario of method 2

Caution: If you open the CSV file in Excel and the values appear in a single column, it means you should use a different delimiter character (e.g., use a semicolon ";").

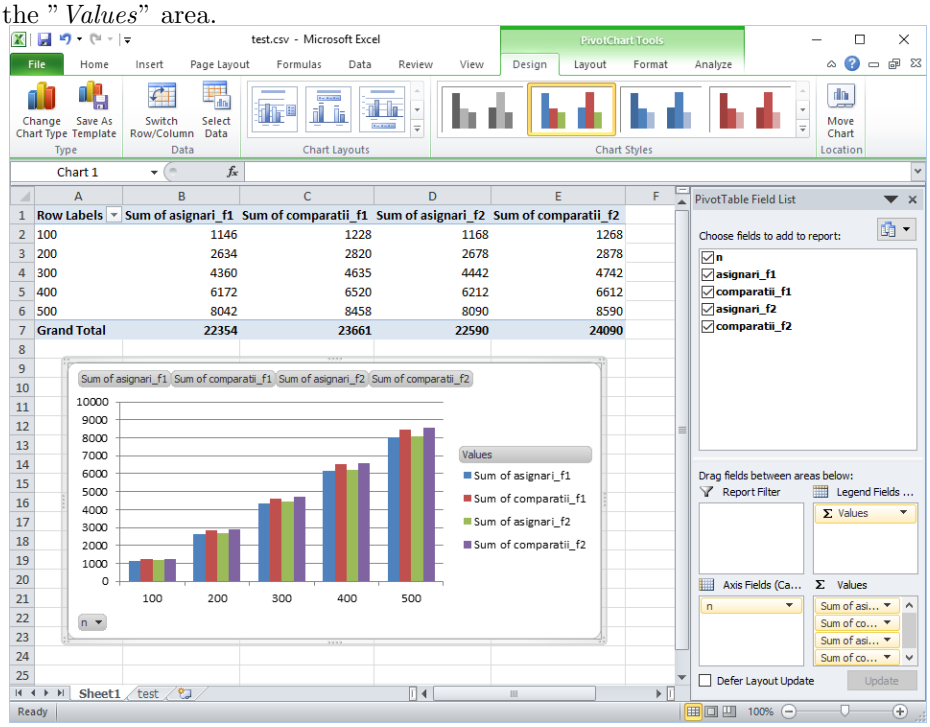
After you've opened the CSV file in Excel, select all the values and create a *PivotChart*.



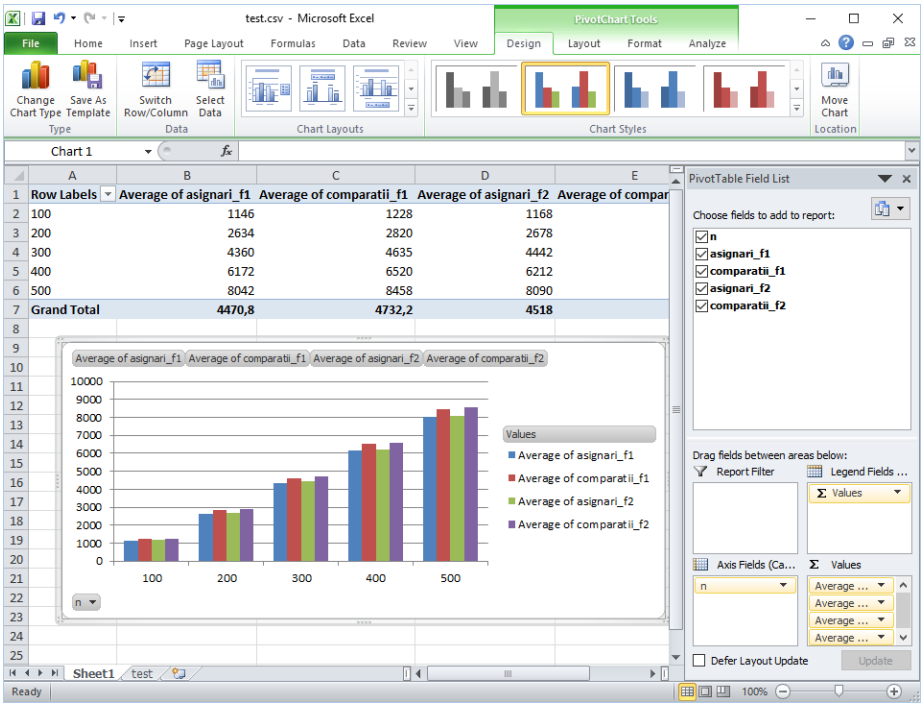
After you click "*Ok*," the window should look like the picture below.



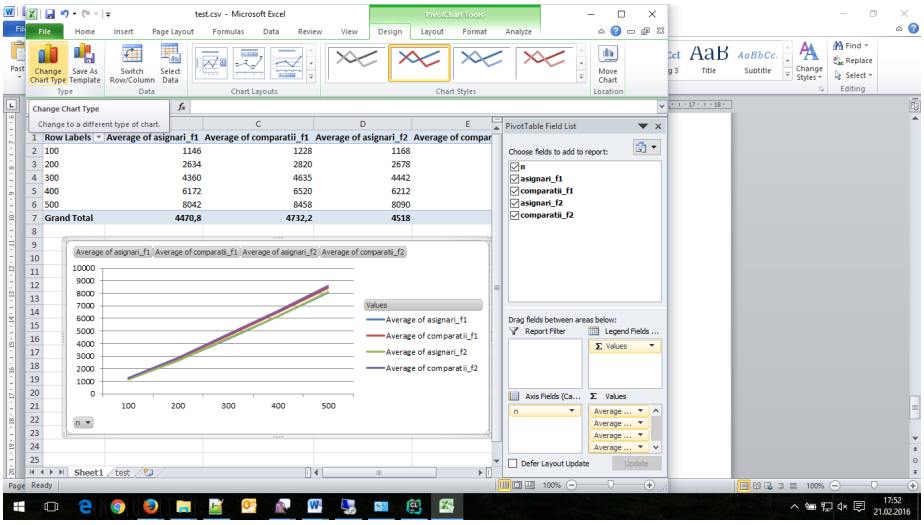
In the left panel, drag "*n*" to the "*Axis Fields*" area and the other columns to the "*Values*" area.



Change the aggregation function from "*Sum*" to "*Average*": Click on the black arrow in each row in the "*Values*" area, then choose "*Value Field Settings*" and select "*Average.*" If you've changed them correctly, the window should look like the picture below.



The final step is to change the chart type to a line chart from "*Change Chart Type*." The final result should look like this:



2.3.5 Exercise

Write a program that for each value n of the interval $\{100, 200, \ldots, 10.000\}$ computes and adds into a file the following values:

n, 100*log(n), 10*n, n*log(n), 0.1*n^2, 0.01*n^3
Use the values from the files to generate a chart depending on n.

3 Assignment No. 1: Analysis & Comparison of Direct Sorting Methods

Allocated time: 2 hours

3.1 Implementation

You are required to implement **correctly** and **efficiently** 3 direct sorting methods (*Bubble Sort*, *Insertion Sort* – using either linear or binary insertion and *Selection Sort*)

- Input: sequence of numbers $< a_1,\ a_2,\ \ldots,\ a_n >$
- Output: an ordered permutation of the input sequence $< a'_1 \leq a'_2 \leq \ldots \leq a'_n >$

You may find any necessary information and pseudo-code in the **Seminar no. 1 notes** (Insertion Sort is also presented in the **book[1]– Section 2.1**). Make sure that you implement the efficient version for each of the required sorting methods (if more than one version has been provided to you).

Minimal requirements for grading

- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- Prepare a demo for each algorithm implemented.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- ***The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and the correct answer to the questions from the teacher.***

3.2 Requirements

3.2.1 Implementation of direct sorting method (5.5p)

- Bubble sort (1.5p)
- Insertion sort (2p)
- Selection sort (2p)

You will have to prove your algorithm(s) work, so you should also prepare a demo on a small-sized input (which may be hard-coded in your *main* function).

3.2.2 Evaluate algorithms for the average case (1.5p – 0.5p for each algorithm)

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm!

You are required to compare the three sorting algorithms, in the **best**, **average** and **worst** cases. Remember that for the **average** case you have to repeat the measurements **m** times (m=5 should suffice) and report their average; also, for the **average** case, make sure you always use the **same** input sequence for all three sorting methods. To make the comparison fair, make sure you know how to generate the **best/worst** case input sequences for all three methods.

This is how the analysis should be performed for a sorting method, in any of the three cases (**best**, **average** and **worst**):

- vary the dimension of the input array (n) between $[100 \ldots 10000]$, with an increment of maximum 500 (we suggest 100).

- for each dimension, generate the appropriate input sequence for the sorting method; run the sorting method counting the operations (i.e., number of assignments, number of comparisons and their sum).

> **!** Only the assignments (=) and comparisons (<, ==, >, !=) which are performed on the input structure and its corresponding auxiliary variables matter.

3.2.3 Evaluate algorithm for best and worst case (3p - 0.5p for each case of each algorithm)

For each analysis case (**best**, **average,** and **worst**), generate charts which compare the three methods; use different charts for the number of comparisons, number of assignments and total number of operations. If one of the curves cannot be visualized correctly because the others have a larger growth rate (e.g., a linear function might seem constant when placed on the same chart with a quadratic function), place that curve on a separate chart as well. Name your charts and the curves on each chart appropriately.

3.2.4 Bonus: Binary insertion sort (0.5p)

You will have to prove your algorithm(s) work, so you should also prepare a demo on a small-sized input (which may be hard-coded in your main function).

You will have to compare binary insertion sort against all other sorting methods (bubble, insertion, selection) on all cases (best, average, worst).

4 Assignment No. 2: Analysis & Comparison of Bottom-up and Top-down Build Heap Approaches

Allocated time: 2 hours

4.1 Implementation

You are required to implement **correctly** and **efficiently** two methods for building a heap, namely the *bottom-up* and the *top-down* strategies. Moreover, you are required to implement the *heapsort* algorithm.

You may find any necessary information and pseudo-code in your course notes, or in the book[1]:

- *Bottom-up*: section 6.3 (Building a heap)
- *Heapsort*: chapter 6.4 (The heapsort algorithm)
- *Top-down*: section 6.5 (Priority queues) and problem 6-1 (Building a heap using insertion)

4.2 Minimal requirements for grading

- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- Prepare a demo for each algorithm implemented.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- ***The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and the correct answer to the questions from the teacher.***

4.3 Requirements

4.3.1 Comparative analysis of *one* of the sorting algorithms from L1 (you choose) in *iterative* vs *recursive* version. The analysis should be performed based on the *number of operations* and the *runtime* (2p)

For the comparative analysis of the iterative vs recursive version pick one of the 3 algorithms from Assignment 1 (bubble sort, insertion or selection). Use the iterative version that you already implemented (corrected, if needed, based on the feedback received from the teacher) and implement the same algorithm in the recursive version.

You must measure the total effort and the running time of the two versions (iterative and recursive) => two charts, each of them comparing the two versions of the algorithm.

For measuring the runtime, you can use Profiler similar to the example below.

```
profiler.startTimer("your_function", current_size);
for(int test=0; test<nr_tests; ++test) {
```

your_function(array, current_size);
}
profiler.stopTimer("your_function", current_size);

The number of tests (*nr_tests* from the above example) has to be chosen based on your processor and the compile mode used. We suggest bigger values such as 100 or 1000.

4.3.2 Implement bottomup build heap procedure (2p)

You will have to prove your algorithm(s) work on a small-sized input.

4.3.3 Implement topdown build heap procedure (2p)

You will have to prove your algorithm(s) work on a small-sized input.

4.3.4 Comparative analysis of the two build heap methods in the average case (2p)

! Before you start to work on the algorithms evaluation code, make sure you have a *correct* algorithm!

You are required to compare the two build heap procedures in the **average** case. Remember that for the **average** case you have to repeat the measurements m times (m=5) and report their average; also, for the **average** case, make sure you always use the **same** input sequence for the two methods – to make the comparison fair.

This is how the analysis should be performed:

- vary the dimension of the input array (n) between $[100 \dots 10000]$, with an increment of maximum 500 (we suggest 100).

- for each dimension (n), generate the appropriate input sequence for the method; run the method, counting the operations (assignments and comparisons, may be counted together for this assignment).

> **!** Only the assignments and comparisons performed on the input structure and its corresponding auxiliary variables matter.
>
> Generate a chart which compares the two methods under the total number of operations, in the **average** case. If one of the curves cannot be visualized correctly because the other has a larger growth rate, place that curve on a separate chart as well. Name your chart and the curves on it appropriately.

4.3.5 Comparative analysis of the two build heap methods in the worst case (1p)

4.3.6 Implement and exemplify correctness of heapsort (1p)

You will have to prove your algorithm(s) work on a small-sized input.

5 Assignment No. 3: Analysis & Comparison of Advanced Sorting Methods – Heapsort and Quicksort / QuickSelect

Allocated time: 2 hours

5.1 Implementation

You are required to implement **correctly** and **efficiently** *Quicksort, Hybrid Quicksort* and *Quick-Select (Randomized-Select)*. You are also required to analyze comparatively of the complexity of *Heapsort* (implemented in Assignment No. 2) and *Quicksort*.

You may find any necessary information and pseudo-code in your course notes, or in the book[1]:

- *Heapsort*: chapter 6 (Heapsort)
- *Quicksort*: chapter 7 (Quicksort)
- *Hybridization for quicksort using iterative insertion sort* - in quicksort, for array sizes < threshold, insertion sort should be used (use insertion sort from Assignment No. 1)
- *Randomized-Select*: chapter 9

5.2 Minimal requirements for grading

- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- Prepare a demo for each algorithm implemented.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- ***The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and the correct answer to the questions from the teacher.***

5.3 Requirements

5.3.1 QuickSort: implementation (2p)

You will have to prove your algorithm(s) work on a small-sized input.

5.3.2 QuickSort: average, best and worst case analysis (3p)

! Before you start to work on the algorithms evaluation code, make sure you have a **correct algorithm**!

This is how the analysis should be performed:

- vary the dimension of the input array (n) between [100...10000], with an increment of maximum 500 (we suggest 100).

- for each dimension, generate the appropriate input sequence for the method; run the method, counting the operations (assignments and comparisons, may be counted together).

> **!** Only the assignments and comparisons performed on the input structure and its corresponding auxiliary variables matter.

5.3.3 Quicksort and Heapsort: comparative analysis of average case (2p)

You are required to compare the two sorting procedures in the **average** case. Remember that for the **average** case you have to repeat the measurements m times (m=5) and report their average; also for the **average** case, make sure you always use the **same** input sequence for the two methods – to make the comparison fair.

Generate a chart which compares the two methods under the total number of operations, in the **average** case.

If one of the curves cannot be visualized correctly because the other has a larger growth rate, place that curve on a separate chart as well. Name the chart and curves appropriately.

5.3.4 Implementation of quicksort hybridization (1p)

You will have to prove your algorithm(s) work on a small-sized input.

5.3.5 Determination of an optimal threshold used in hybridization + proof (graphics/ measurements) (1p)

You should vary the threshold value of quicksort hibridization for which insertion sort is applied.

Compare the results from the performance (number of operations and execution time) perspective for determination of the optimum threshold. You can use 10.000 as the fixed size of the vector that is being sorted and vary the threshold between [5,50] with an increment of 1 to 5.

The number of tests (nr_tests from the example) has to be chosen based on your processor and the compile mode used. We suggest bigger values such as 100 or 1000.

5.3.6 Comparative analysis (between *quicksort and quicksort hibridization*) from the operations and runtime perspective (1p)

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm!

For quicksort hybridization, you have to use the iterative insertion sort from the first assignment if the size of the vector is small (we suggest using insertion sort if the vector has less than 30 elements). Compare *runtime and the number of operations* (assignments + comparisons) for quicksort implemented in the third assignment with the hybrid one.

For measuring the runtime you can use Profiler similar to the example below.

```
profiler.startTimer("your_function", current_size);
for(int test=0; test<nr_tests; ++test) {
   your_function(array, current_size);
}
profiler.stopTimer("your_function", current_size);
```

When you are measuring the execution time make sure all the processes that are not critical are stopped.

5.3.7 Bonus: QuickSelect - Randomized-Select (0.5p)

You will have to prove your algorithm(s) work on a small-sized input.

For QuickSelect (Randomized-Select) no explicit complexity analysis needs to be performed, only the correctness needs to be demonstrated on sample inputs.

6 Assignment No. 4: Merge k Ordered Lists Efficiently

Allocated time: 2 hours

6.1 Implementation

You are required to implement **correctly** and **efficiently** an O(nlogk) method for **merging k sorted sequences**, where n is the total number of elements. (Hint: use a heap, see *Seminar no. 2* notes).

Implementation requirements:

- Use linked lists to represent the k sorted sequences and the output sequence

Input: k lists of numbers $< a_1^i, a_2^i, \ldots, a_{m_i}^i >$, $\sum_{i=1}^{k} m_i = n$

Output: a permutation of the union of the input sequences: $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

6.2 Minimal requirements for grading

- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- Prepare a demo for each algorithm implemented.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- ***The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and the correct answer to the questions from the teacher.***

6.3 Requirements

6.3.1 Generate *k* randomly sized sorted lists (having *n* elements in total, *n* and *k* given as parameters) and the merging of 2 lists (5p)

You will have to show your algorithm (*generation and merging*) works on a small-sized input (e.g. k=4, n=20).

6.3.2 Adapt *min-heap* operations to work on the new structure and the merging of k lists (3p)

You will have to show your algorithm (*merging*) works on a small-sized input (e.g. k=4, n=20).

6.3.3 Evaluation of the algorithm in average case (2p)

! Before you start to work on the algorithm's evaluation code, make sure you have a correct algorithm!

We will make the **average case** analysis of the algorithm. Remember that, in the average case, you have to repeat the measurements several times. Since both **k** and **n** may vary, we will make each analysis in turn:

- Choose, in turn, 3 constant values for k (**k1=5, k2=10, k3=100**); generate k **random** sorted lists for each value of k so that the number of elements in all the lists n varies between **100 and 10000**, with a maximum increment of 400 (we suggest 100); run the algorithm for all values of n (for each value of k); generate a chart that represents the **sum of assignments and comparisons** done by the merging algorithm for each value of k as a curve (total 3 curves).
- Set ***n = 10.000***; the value of k must vary between 10 and 500 with an increment of 10; generate k **random** sorted lists for each value of k so that the number of elements in all the lists is 10000; test the merging

algorithm for each value of k and generate a chart that represents the **sum of assignments and comparisons** as a curve.

7 Assignment No. 5: Search Operation in Hash Tables

Open Addressing with Quadratic Probing
Allocated time: 2 hours

7.1 Implementation

You are required to implement **correctly** and **efficiently** the *insert* and *search* operations in a hash table using *open addressing* and *quadratic probing*.

You may find relevant information and pseudo-code in your course notes, or in the book (Cormen), in section *11.4 Open addressing*.

The notions of closed/open specify whether you are compelled to use a certain position or a data structure.

7.1.1 Hashing (refers to the hash table)

- Open Hashing
  - Free to leave the hash table to hold more elements at a certain index (e.g. chaining)
- Closed Hashing
  - Not more than one element can be stored at a certain index (e.g. linear/quadratic probing)

7.1.2 Addressing (refers to the final position of the element with respect to its initial position)

- Open Addressing
  - The final address is not completely determined by the hash code, it also depends on the elements which are already in the hash table (e.g. linear/quadratic probing)
- Closed Addressing
  - The final address is always the one initially calculated (there is no probing, e.g. chaining)

For the purpose of this assignment, the hash table will not contain integers, but a custom data structure defined as follows:

```
typedef struct {
    int  id;
    char name[30];
} Entry;
```

The *position* of each Entry in the Hash Table will be calculated by applying the required hash function on the *id* member of the struct. The *name* member of the struct will be used only to exemplify the correctness of the search and delete operations and is not needed when evaluating the performance (i.e., the *name* member will be printed to the console if the search operation finds the *id*, otherwise print "not found").

7.2 Minimal requirements for grading

- Interpret the chart and write your observations in the header (block comments) section at the beginning of your *main.cpp* file.
- Prepare a demo for each algorithm implemented.
- We do not accept assignments without code indentation and with code not organized in functions (for example where the entire code is in the main function).
- ***The points from the requirements correspond to a correct and complete solution, quality of interpretation from the block comment and the correct answer to the questions from the teacher.***

7.3 Requirements

7.3.1 Implementation of the insert and search operations using the required data structure (5p)

You will have to prove your algorithm(s) work on a small-sized input *(ex. 10)*.

7.3.2 Evaluate the search operation for a single fill factor 95% (2p)

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm!

You are required to evaluate the *search* operation for hash tables using open addressing and quadratic probing, in the **average case** (remember to perform 5 runs for this). You will do this in the following manner:

1. Select N, the size of your hash table, as a prime number around 10000 (e.g., 9973, or 10007);
2. For each of several values for the filling factor α **=0.95** do:
    a. Insert n random elements, such that you reach the required value for α ($\alpha = n/N$)

b. Search, in each case, m random elements (m ~ 3000), such that approximately half of the searched elements will be *found* in the table, and the rest will *not* be *found* (in the table). *Make sure that you sample uniformly the elements in the found category, i.e., you should search elements which have been inserted at different moments with equal probability (there are several ways in which you could ensure this – it is up to you to figure this out)*

c. Count the operations performed by the search procedure (i.e., the number of cells accessed during the search)

d. Pay attention to the values that you search for, they should be in random order of introduction. *If you look for the first 1500 values introduced in the table, implicitly the average found effort will be 1.*

3. Output a table in the following form:

Table 1: Effort measurements at various filling factors

Filling factor	**Avg. Effort *(found)***	**Max Effort *(found)***	**Avg. Effort *(not-found)***	**Max Effort *(not-found)***
0.95	...	...	...	...

Avg. Effort = total_effort / no_elements
Max. Effort = maximum number of accesses performed by one search operation

7.3.3 Complete evaluation for all fill factors (2p)

Respecting the requirements of point 2 with $\alpha \in \{0.8, 0.85, 0.9, 0.95, 0.99\}$, output a table in the following form:

Table 2: Effort measurements at various filling factors

Filling factor	**Avg. Effort *(found)***	**Max Effort *(found)***	**Avg. Effort *(not-found)***	**Max Effort *(not-found)***
0.80				
0.85				
...	...		...	

7.3.4 Implement delete operation in a hash table, *demo (size 10)* and evaluation of the search operation after deletion of some elements (1p)

For the evaluation of the search operation after deletion, fill the hash table until a fill factor of 0.99. Delete elements from the table until you get a filling factor of 0.8 and afterwards search m random elements (m ~ 3000) such that approximately half of the searched elements will be *found* in the table, and the rest will *not be found* (in the table). Count the operations performed by the *search* and add it in the previous table.

8 Assignment No. 6: Multi-way Trees

Transforms between different representations **Allocated time:** 2 hours

8.1 Implementation

1. You are required to implement **correctly** and **efficiently** *iterative* and *recursive* binary tree traversal. You may find any necessary information and pseudo-code in your course and seminar notes.

2. Moreover, the **correct** and **efficient** implementation of *linear* complexity algorithms is required for transforming multi-way trees between the following representations:

    1. **R1**: *Parent representation*: for each index, the value in the vector represents the parent's index, e.g.: $\Pi = \{2, 7, 5, 2, 7, 7, -1, 5, 2\}$
    2. **R2**: *Multi-way tree representation*: each node contains the key and a vector of child nodes.
    3. **R3**: *Binary representation*: each node contains the key and two pointers, one to the first child and the second to the right sibling (e.g., the next sibling).

Therefore, you need to define transformation **T1** from the *parent representation* (**R1)** to the *multi-way tree representation* (**R2**), and then the transformation **T2** from the *multi-way tree representation* (**R2**) to the *binary representation* (**R3**). For all representations (**R1**, **R2**, **R3**), you need to implement the Pretty Print (**PP**) display (see page 2).

Define the data structures. You can use intermediate structures (e.g., additional memory).

8.2 Requirements

8.2.1 Implementation of *iterative* and *recursive* binary tree traversal in *O(n)* and *with constant additional memory* (3p)

You will have to prove your algorithm(s) work on a small-sized input.

8.2.2 Implementation of transforms between different representations

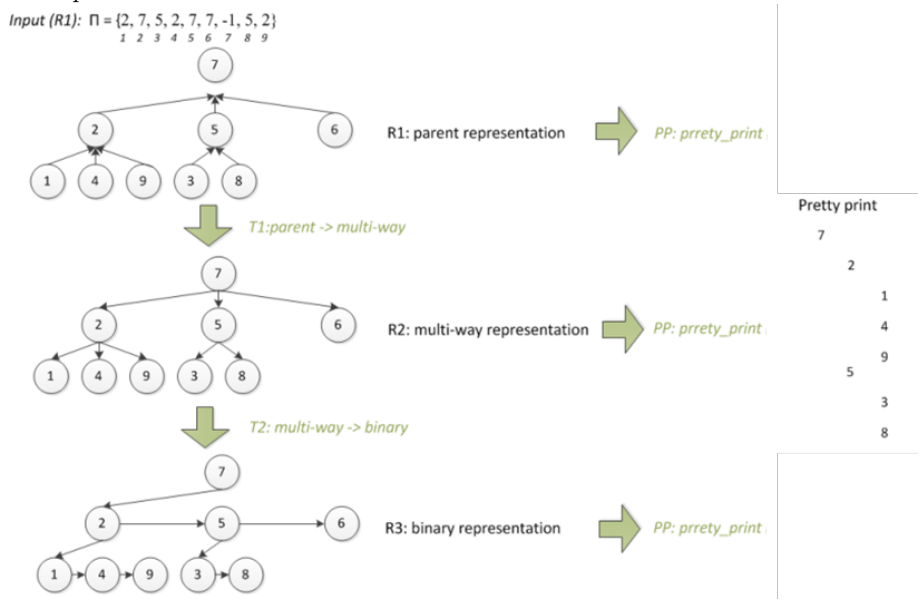
8.2.3 Correct implementation for Pretty-print for R1 (2p)

8.2.4 Correct implementation for *T1 (from R1 to R2)* and pretty-print for *R2* (1p) + *T1* in linear time (1p)

8.2.5 Correct implementation for *T2 (from R2 to R3)* and pretty-print for *R3* (2p) + *T2* in linear time (1p)

The correctness of the algorithms should be demonstrated using the example $\Pi = \{2, 7, 5, 2, 7, 7, -1, 5, 2\}$.

Use Pretty Print for all three representations. *Each representation (R1,R2,R3) should have a pretty print of its own with a different implementation but with the same print.*



Analyse the time and space efficiency of the two transformations. Did you achieve *O(n)*? Did you use additional memory?

9 Assignment No. 7: Dynamic Order Statistics

Allocated time: 2 hours

9.1 Implementation

You are required to implement **correctly** and **efficiently** the management operations of an **order statistics tree** (*chapter 14.1 from the book[1]*).

You have to use a balanced, augmented Binary Search Tree. Each node in the tree holds, besides the necessary information, also the *size* field (i.e. the size of the sub-tree rooted at the node).

The management operations of an **order statistics tree** are:

- BUILD_TREE(n)
  - *builds* a **balanced** BST containing the keys 1,2,...n (*hint:* use a divide and conquer approach)
  - make sure you initialize the size field in each tree node
- OS-SELECT(tree, i)
  - selects the element with the i-th smallest key
  - the pseudo-code is available in *chapter 14.1 from the book[1]*
- OS-DELETE(tree, i)
  - you may use the deletion from a BST, without increasing the height of the tree (why don't you need to rebalance the tree?)
  - keep the size information consistent after subsequent deletes
  - there are several alternatives to update the size field without increasing the complexity of the algorithm (it is up to you to figure this out).

Does OS-SELECT resemble anything you studied this semester?

9.2 Requirements

9.2.1 BUILD_TREE: correct and efficient implementation (5p)

You will have to prove your algorithm(s) work on a small-sized input (11)

- pretty-print the initially built tree

9.2.2 OS_SELECT: correct and efficient implementation (1p)

You will have to prove your algorithm(s) work on a small-sized input (11)

- execute OS-SELECT for a few elements (at least 3) by a randomly selected index

9.2.3 OS_DELETE: correct and efficient implementation (2p)

You will have to prove your algorithm(s) work on a small-sized input (11)

- execute OS-SELECT followed by OS-DELETE for a few elements (at least 3) by a randomly selected index *and pretty-print the tree after each execution.*

9.2.4 Management operations evaluation - BUILD, SELECT, DELETE (2p)

! Before you start to work on the algorithms evaluation code, make sure you have a **correct algorithm**!

Once you are sure your program works correctly:

- vary n from 100 to 10000 with a step of 100;
- for each n (don't forget to repeat 5 times),
  - BUILD a tree with elements from *1 to n*
  - perform n sequences of OS-SELECT and OS-DELETE operations using a randomly selected index based on the remaining number of elements in the BST,
  - Evaluate the number of operations needed for each management operation (BUILD, SELECT, DELETE – *resulting in a plot with 3 series*). Evaluate the computational effort as the sum of the comparisons and assignments performed by each individual management operation ofr each value of n.

9.2.5 Bonus: Implementation using AVL / Red black tree (1p)

10 Assignment No. 8: Disjoint Sets

Allocated time: 2 hours

10.1 Implementation

You are required to implement **correctly** and **efficiently** the base operations for **disjoint set** (*chapter 21.1*[1]) and the **Kruskal's algorithm** (searching for the minimum spanning tree, *chapter 23.2*[1]) using disjoint sets.

You have to use a tree as the representation of a disjoint set. Each tree holds, besides the necessary information, also the *rank* field (i.e. the height of the tree).

The base operations on **disjoints sets** are:

[1]Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*

- MAKE_SET (x)
  - creates a set with the element x
- UNION (x, y)
  - makes the union between the set that contains the element x and the set that contains the element y
  - the heuristic *union by rank* takes into account the height of the two trees so as to make the union
  - the pseudo-code can be found in *chapter 21.3*[1]
- FIND_SET (x)
  - searches for the set that contains the element x
  - the heuristic *path compression* links all nodes that were found on the path to x to the root node

10.2 Requirements

10.2.1 Correct implementation of MAKE_SET, UNION and FIND_SET (5p)

The correctness of the algorithm must be proved on a small-sized input

- create (MAKE) 10 sets + show the contents of the sets
- execute the sequence UNION and FIND_SET for 5 elements + show the contents of the sets

10.2.2 Correct and efficient implementation for Kruskal's algorithm (2p)

The correctness of the algorithm must be proved on a small-sized input

- create a graph of 5 nodes and 9 edges + **print the edges**
- apply Kruskal's algorithm + **print the chosen edges**

10.2.3 Evaluate the disjoint sets operations (MAKE, UNION, FIND) using Kruskal's algorithm (3p)

! Before you start to work on the algorithms evaluation code, make sure you have a correct algorithm!

Once you are sure your program works correctly:

- vary n from 100 to 10000 with a step of 100
- for each n

- build an **undirected**, **connected**, and **random** graph with random weights on edges (**n** nodes, **n*4** edges)
- find the minimum spanning tree using Kruskal's algorithm
  - evaluate the computational effort of each individual base operation (MAKE, UNION, FIND– *resulting in a plot with 3 series*) on disjoint sets as the sum of the comparisons and assignments performed; thus, there should be **3 series in the plot**, one for each operation.

11 Assignment No. 9. BFS: Breadth-First Search

11.1 Introduction

In this session you are required to implement and analyse BFS (Breadth-First Search), as presented in subsection 22.2 of Introduction to Algorithms by Th. Cormen et al.

11.2 The structure of the boiler plate code

You are already provided with several source files:

- `main.cpp` - the main source file, which makes the calls to the implemented functions and provides the visualization code
- `bfs.h` - contains definitions of data structures and functions used in the project
- `bfs.cpp` - will contain the implementations of the required algorithms
- `grid.txt` - the maze which represents the graph for the demo
- `Profiler.h` - the library used for algorithm evaluation and chart generation

!!! You should only make changes to `bfs.cpp`.

For a user-friendly visualization, `main.cpp` displays an ASCII-like interface, in which the maze (i.e. the graph) is displayed (black cells are free, white cells represent walls).

11.2.1 Project setup for Windows, with Visual Studio

Make a new Project in Visual Studio. Make sure to check the *"Empty Project"* checkbox. Then, copy all the files provided in the project folder.

Then, in Visual Studio, add the two files `.h` to the *"Header Files"* subsection of your project, and the two `.cpp` files to the *"Source Files"* subsection (right click on corresponding project subsection → *"Add"* → *"Existing Item"*, see Figure 1.

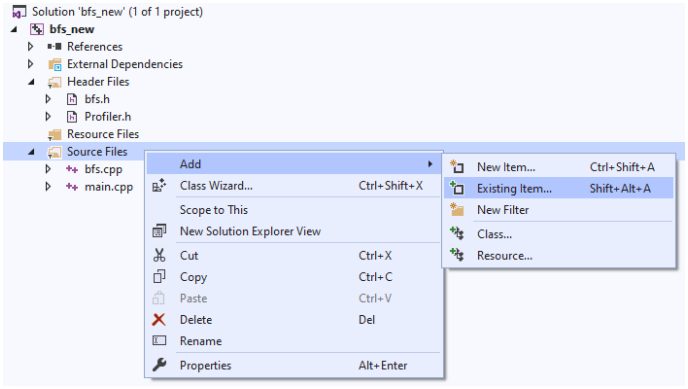


Figure 1: The *"Solution Explorer"* window in Visual Studio

11.2.2 Project setup for Linux and Mac

You may edit the source files using your editor of choice. The project comes with a `Makefile`, so it is enough to run `make` in a terminal to compile it and generate the executable. The resulting `.exe` will be called `main`, and can be run in the terminal by executing `./main`.

11.3 Running the program

When you run the program, it will display the maze, similarly to what you can observe in Figure 2.

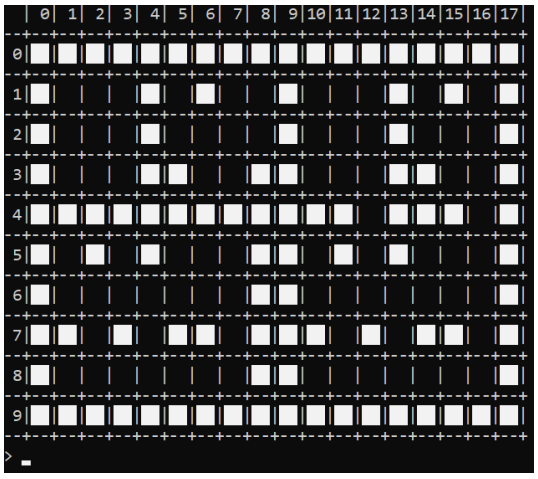


Figure 2: Interface of the program

The user may input one of the following commands:

- `exit`
  program termination
- `clear`
  clear the previous information from the grid
- `neighb <row> <col>`
  display the neighbors of cell on row `<row>` and column `<col>`.
- `bfs <row> <col>`
  execute the BFS traversal, starting from the cell on row `<row>` and column `<col>`.
- `bfs_step <row> <col>`
  same as `bfs`, but the result is displayed step by step, depending on the distance from the source node
- `bfs_tree <row> <col>`
  same as `bfs`, but it will also display the output tree under the grid
- `path <row1> <col1> <row2> <col2>`
  displays the shortest path between `(<row1> <col1>)` and `(<row2> <col2>)`
- `perf`
  generates the charts for the algorithm evaluation

11.3.1 Example: command `neighb`

If you run:
`neighb 2 3` you should get the output displayed in Figure 3.

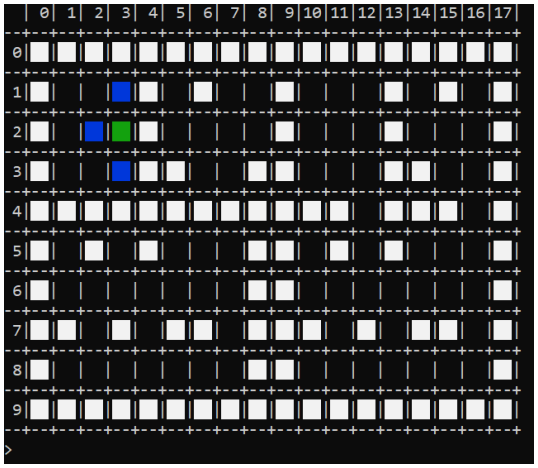


Figure 3: The result of running command `neighb 2 3`

The starting cell will be coloured green, with its neighbors coloured blue.

Since `get_neighbors()` is not yet implemented (you will be required to do the implementation), you will not get this answer if you run the command. Once you implement the function, you may use this command to check the correctness of your implementation. Each cell can have at most 4 neighbors (up, down, left, right); cells outside the grid or wall cells must not appear as neighbors.

11.3.2 Example: commands `bfs` and `bfs_step`

Upon running the command:
`bfs 6 3` the program should output the result shown in Figure 4.

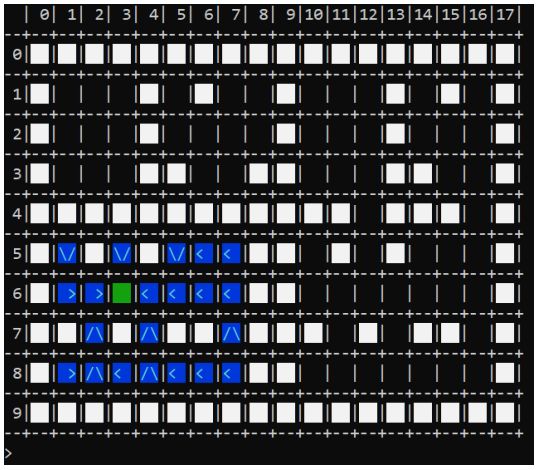


Figure 4: The result of running command `bfs 6 3`

The source cell will be coloured green, and the other cells traversed will be coloured blue. Each blue cell will have an arrow on it, indicating the direction of the parent in the BFS tree.

Currently, the `bfs()` function is not implemented. Once implemented, use this command to check the correctness of your implementation.

11.3.3 Example: command `bfs_tree`

By running:
`bfs 2 6` you should get the image in Figure 5.

The root of the tree is the source node, i.e. `(2, 6)`. The children of this node in the tree are: `(2, 5)`, `(2, 7)` and `(3, 6)` (the order might differ, according to the implementation).

11.3.4 Example: command `path`

By running:
`path 5 10 3 15` you should obtain the image in Figure 6.

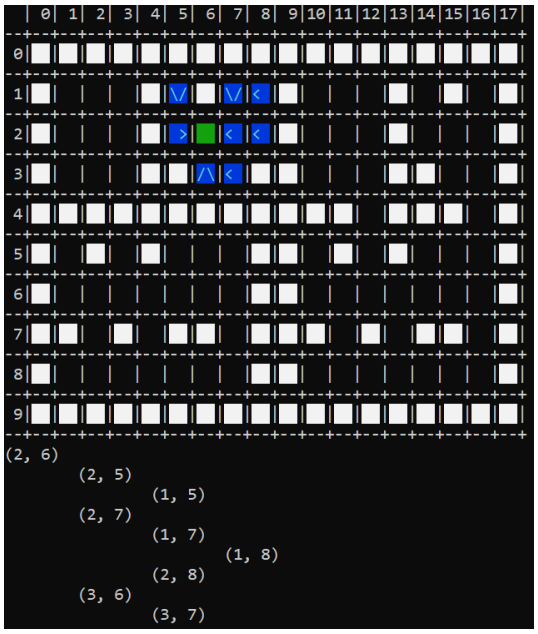


Figure 5: The result of running command `bfs_tree 2 6`

The source cell will be coloured green, the destination cell red, and the rest of the cells on the path - blue. Each blue cell will also have an arrow indicating the direction of the traversal.

Currently, `shortest_path()` and `bfs()` are not implemented, so you will not see this result when running the `path` command. Once the functions implemented, use this command to check the correctness of your implementations.

11.3.5 Employed data structures

The file `bfs.h` contains definitions of data structures used in the framework.

The `Grid` structure models a grid, having `rows` lines and `cols` columns, the grid elements are stored in the matrix `mat`. An empty cell has value 0 in the matrix, and a wall cell will have value 1.

The `Point` structure models a point (i.e. a grid cell) the `row` and `col` fields representing the location, i.e. the row and the column of the point in the grid.

The `Node` structure models a node in the graph, and it contains:

- `position` having type `Point` represents the cell corresponding to the graph node.
- `adjSize` - the number of neighbors of the node
- `adj` - the neighbor array, containing `adjSize` neighbours

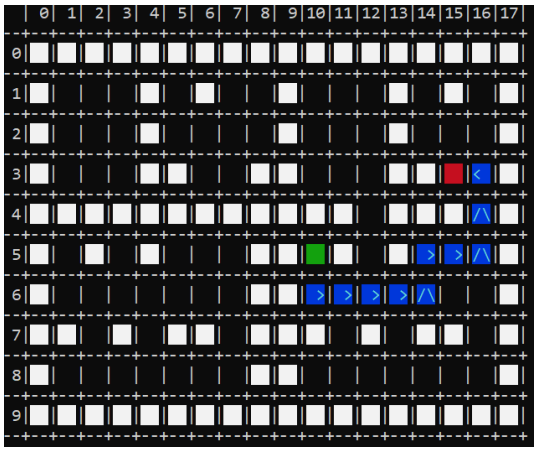


Figure 6: The result of running command `path 5 10 3 15`

- `color` - the color of the node; initially, all nodes are colored `COLOR_WHITE`, i.e. the color has value 0
- `dist` - the distance from the source node, in BFS
- `parent` - pointer to the parent node, in the BFS tree

The `Graph` structure models the graph; it has as members the number of nodes, `nrNodes` and the array `v` containing pointers to the neighbor arrays of each node.

11.4 Requirements

11.4.1 Determine the neighbors of a cell (2p)

In `bfs.cpp`, tyou have to complete the `get_neighbors()` function which receives a pointer to a `Grid` structure, a point `p` of type `Point` and an array `neighb` of points, which the function will fill with the neighbors of point `p`. The function returns the number of neighbors filled in the `neighb` array.

A point on the grid can have maximum 4 neighbors (up, down, left, right). Not all neighbors are necessarily valid: some may end up outside the grid (negative, or out of bounds coordinates) or inside a wall.Therefore, after computing the position of a potential neighbor, you should check that it is situated inside the grid, on a free value cell (the value in the matrix of the corresponding cell should be 0).

The valid neighbors will be added to the array `neighb`. It is guaranteed that the array has at most 4 elements, so you may not exceed this capacity. Because the number of neighbors could be less than 4, you should also return this information.

11.4.2 BFS algorithm implementation (3p)

In `bfs.cpp`, you have to complete the function `bfs()` which receive as arguments a pointer to a structure of type `Graph` and the source node `s` of type `Node*`. The function will implement BFS, as specified in the algorithm from the book (see subsection 22.2 from Cormen).

Initially, the nodes of the graph are colored white, i.e. `COLOR_WHITE`, and the `dist` and `parent` fields are initialized with 0 and `NULL`, respectively. At the end of the traversal, all nodes that can be reached from the cource node are colored `COLOR_BLACK`, the distance `dist` has as value the number of steps from the source node to that node, and the `parent` pointer should idicate the parent in the BFS tree.

11.4.3 Pretty printing the BFS tree (2p)

In `bfs.cpp`, t you have to complete the implementation for `print_bfs_tree()` which receives as parameter a pointer to a `Graph` structure on which the BFS algorithm has already been run, so the node colors and the parent information is already set.

In the function, you already have the construction of the parent array `p`, in which the nodes colored black in the BFS traversal will be numbered from 0 to n. Also, it builds the `repr` array, which contains the coordinates of each node (in the grid).

To display this tree, you have to adapt the code from the multiway trees assignment.

11.4.4 Evaluate the performance of BFS (3p)

The `performance()` function evaluates the BFS algorithm, by varying, in turn, the number of edges, then the number of nodes (and always keeping the other as constant). For each value , you have to implement the generation of a random, connected, graph, having a given number of nodes and edges, respectively.

Inside the `bfs()` function, you will have to actually count the operations performed, using the optional argument `op`. Because this parameter is optional, sometimes `bfs()` will be called by the framework with this parameter set to `NULL`. Consequently, whenever counting an operation, you must first check that `op` is a valid pointer, i.e.:

```
if(op != NULL) op->count();
```

11.4.5 Bonus: Determine the shortest path (0.5p)

In `bfs.cpp`, you have to fill in the code for the `shortest_path()` function, which receives as argument a pointer to a `Graph` structure, a source node and a destination node `start` and `end` of type `Node*`, and a `path` array, y - where the result will be stored, i.e. the nodes on the path, in order. The function returns the number of nodes stored in `path`.

To determine the shortest path between two nodes, you should use the already implemented BFS, and reconstruct the path from the parent array computed by BFS.

The `path` array has a capacity of 1000 elements (upon the call). The function should return the number of elements that it contains – i.e. the path length – or -1 in case there is no path from `start` to `end`.

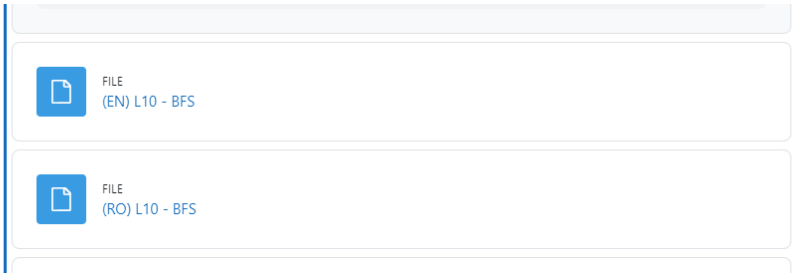
11.4.6 Bonus: Where can a knight end up on the board? (0.5p)

Using this framework, show that a knight starting from the up-left corner can end up in any position of an empty chess board. Give examples of empty chessboards that contains positions unreachable by a knight.

12 Assignment 9: Tutorial

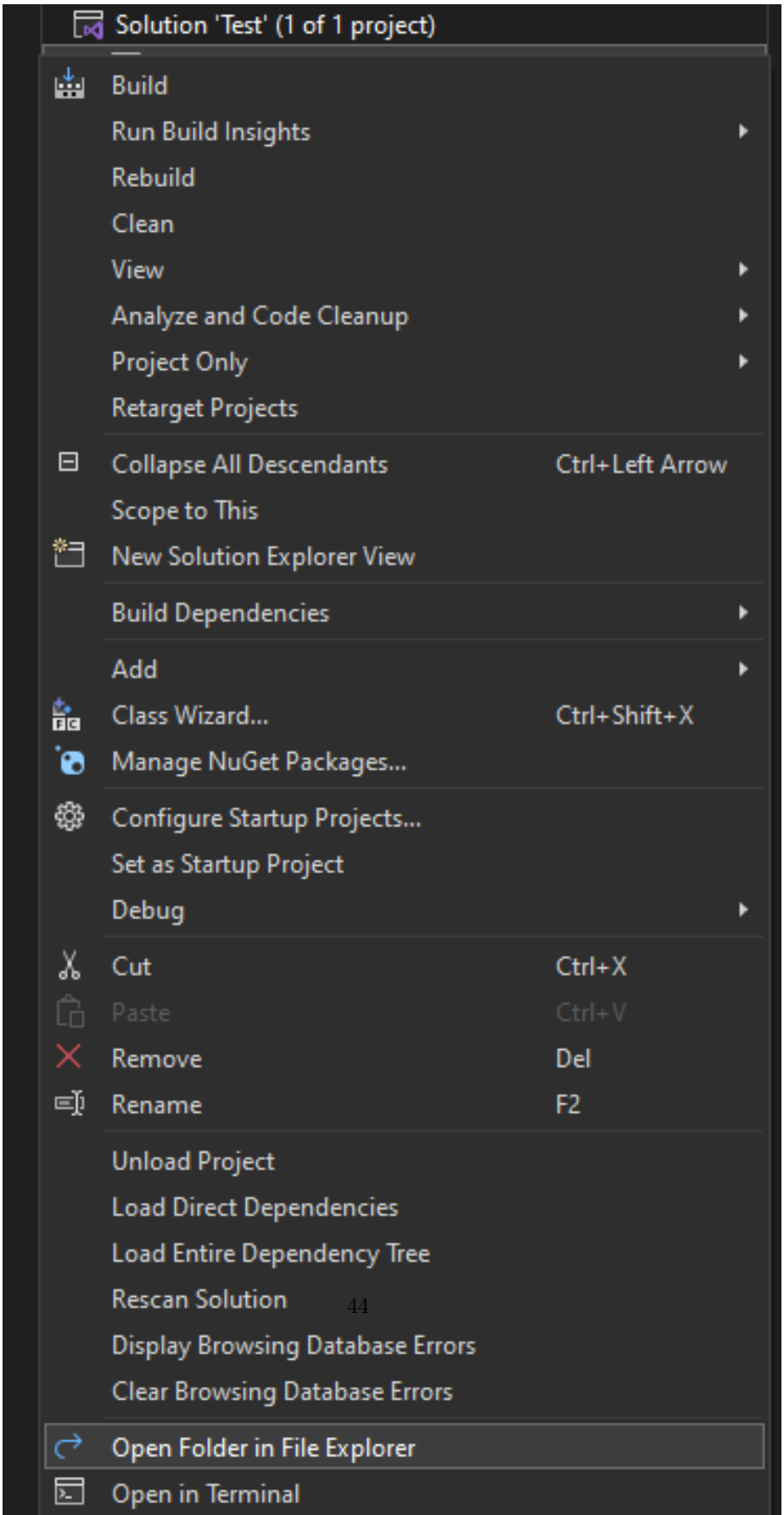
12.1 Visual Studio Project Setup

1. Go to moodle and select one of the following:

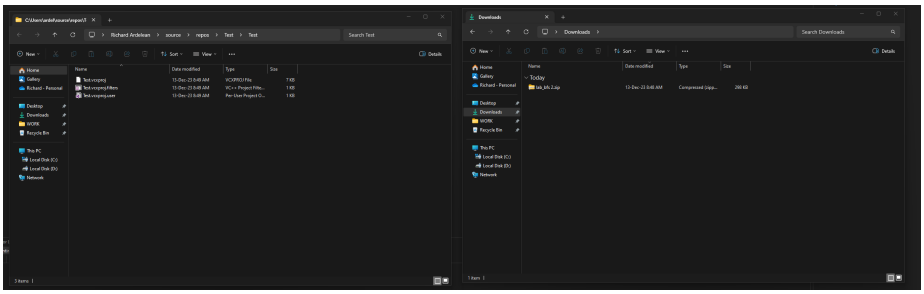


   This will result in a '.zip' file being downloaded.

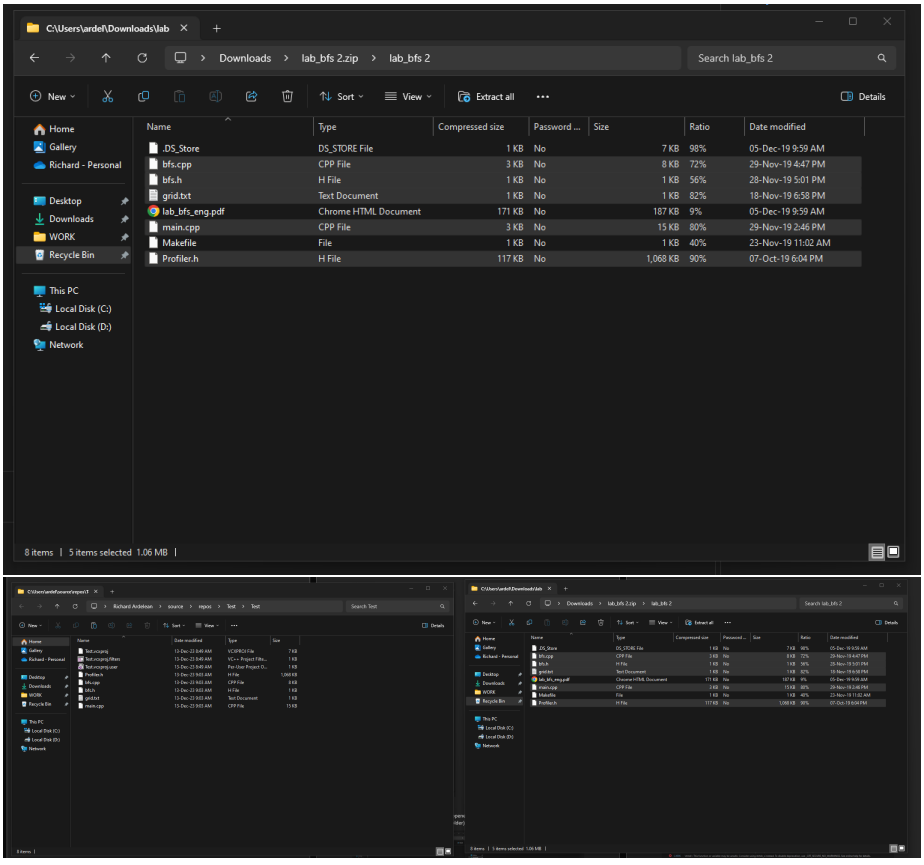
2. Create an Empty C++ Project and by right-clicking on the project (not the solution) you will be able to select 'Open Folder in File Explorer'.


Solution 'Test' (1 of 1 project)
Build
Run Build Insights
Rebuild
Clean
View
Analyze and Code Cleanup
Project Only
Retarget Projects
Collapse All Descendants Ctrl+Left Arrow
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Configure Startup Projects...
Set as Startup Project
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename F2
Unload Project
Load Direct Dependencies
Load Entire Dependency Tree
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Open Folder in File Explorer
Open in Terminal

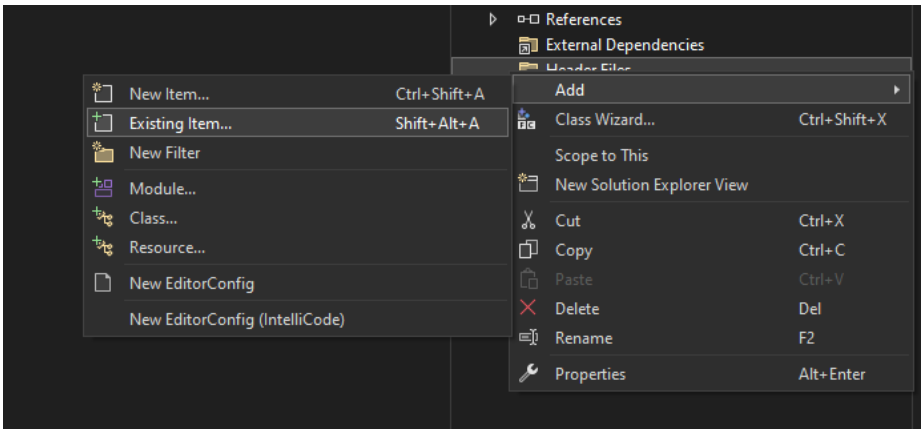
3. At this point, a 'File Explorer' window will be opened. Open another 'File Explorer' with the location of your downloads (most likely the Downloads folder).



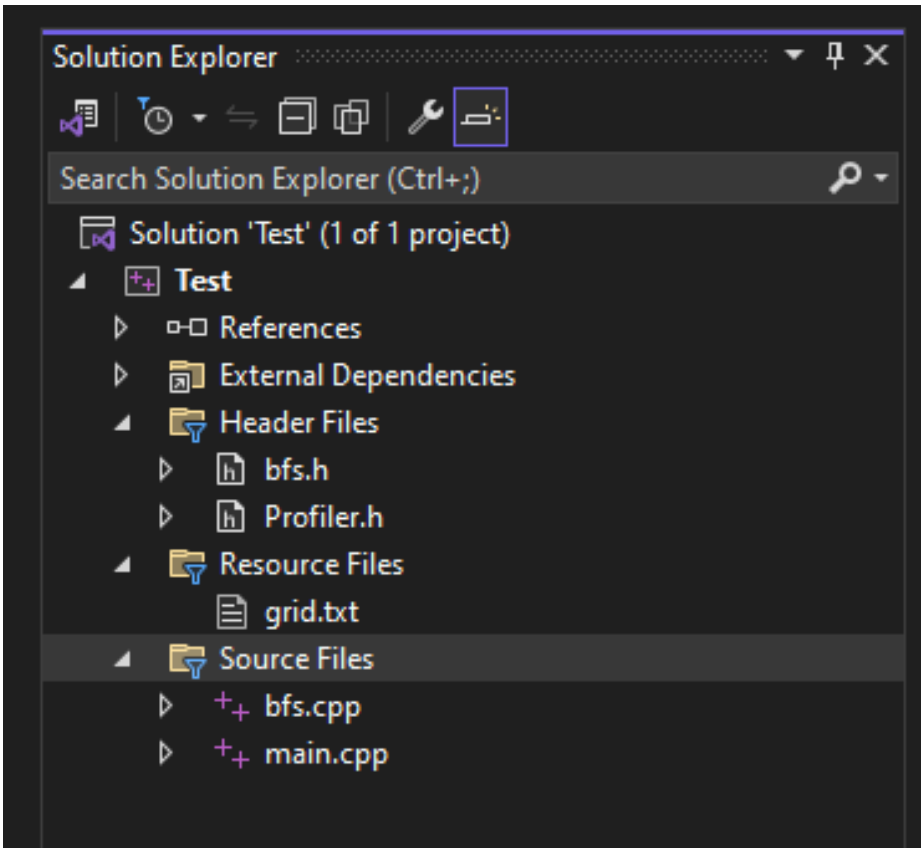
4. Open the '.zip' file and copy as shown below the files from the zip file to the folder of the project.



5. By selecting the folders from the Visual Studio Solution Explorer 'Header Files / Resources / Source Files' use the following options:

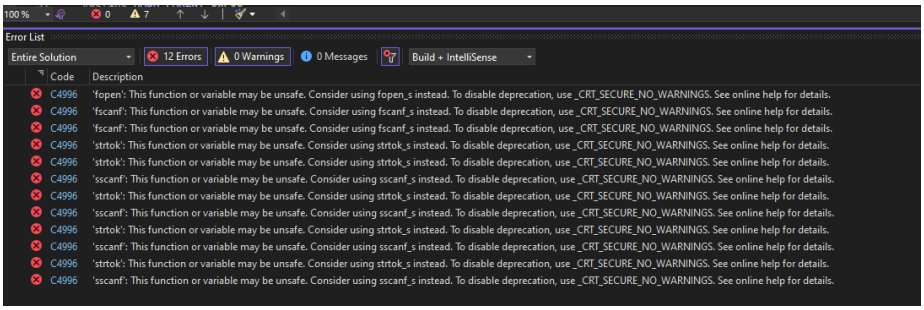


6. Such that, the following setup is accomplished:



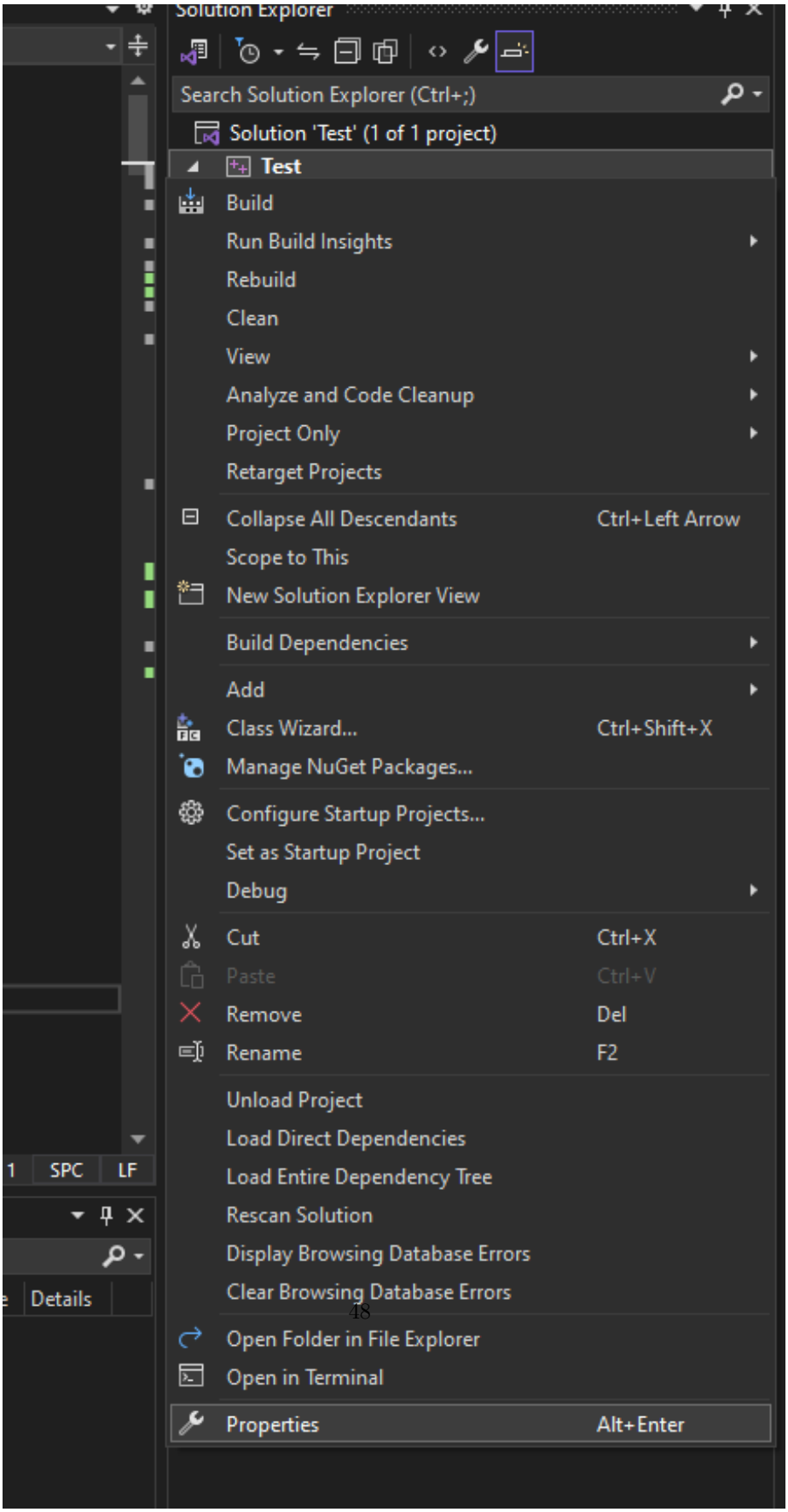
12.2 Visual Studio 'unsafe' error

Example:



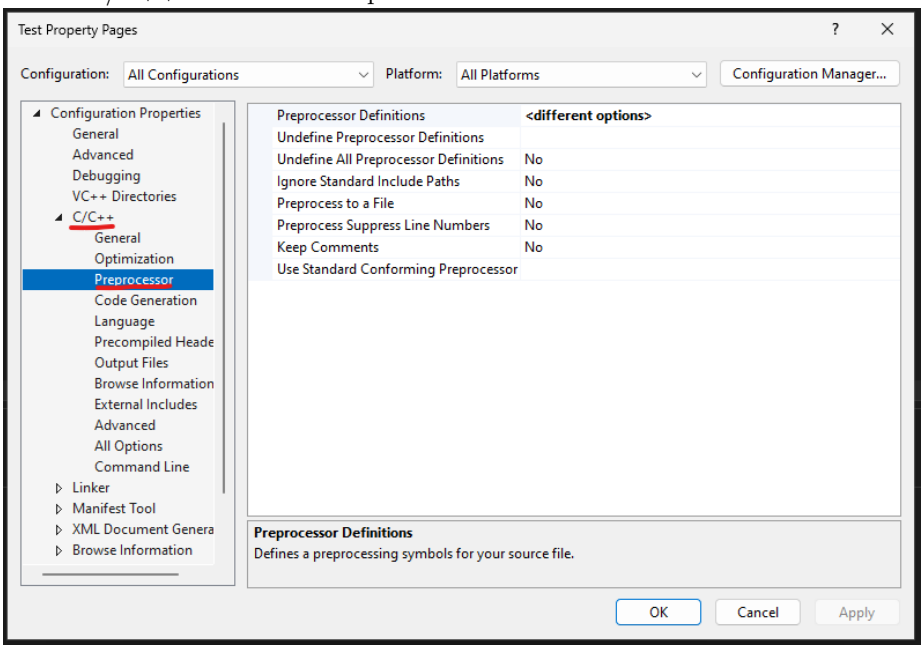
Solution:

Right-click on the project (not the Solution which will most likely have the same name) and select ‘Properties’.

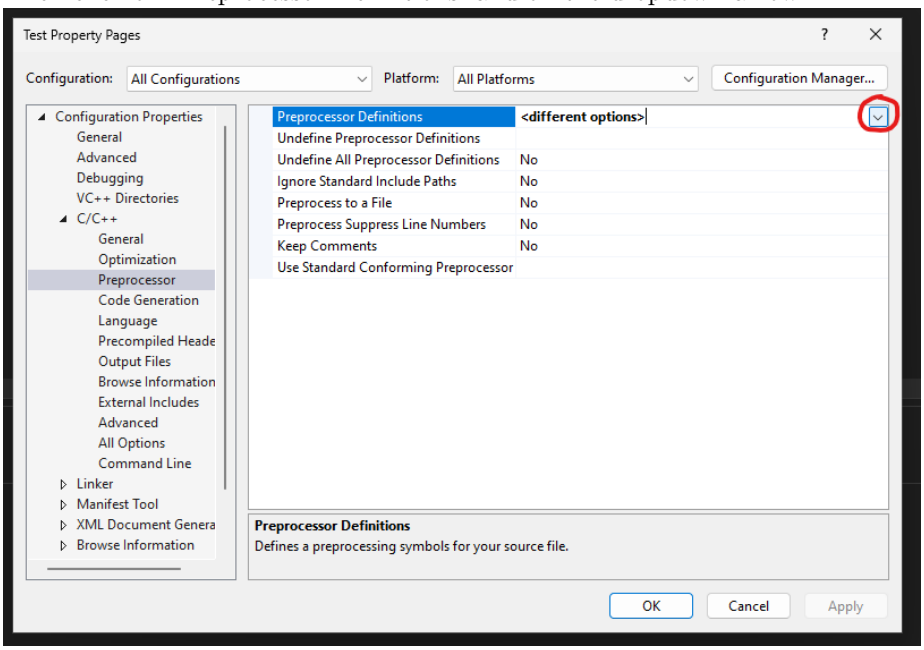

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Test' (1 of 1 project)
Test
Build
Run Build Insights
Rebuild
Clean
View
Analyze and Code Cleanup
Project Only
Retarget Projects
Collapse All Descendants Ctrl+Left Arrow
Scope to This
New Solution Explorer View
Build Dependencies
Add
Class Wizard... Ctrl+Shift+X
Manage NuGet Packages...
Configure Startup Projects...
Set as Startup Project
Debug
Cut Ctrl+X
Paste Ctrl+V
Remove Del
Rename F2
Unload Project
Load Direct Dependencies
Load Entire Dependency Tree
Rescan Solution
Display Browsing Database Errors
Clear Browsing Database Errors
Open Folder in File Explorer
Open in Terminal
Properties Alt+Enter
SPC
LF
Details

Update the ‘Configuration’ and ‘Platform’ in the upper side of the window to ‘All Configurations’ and ‘All Platforms’, respectively.

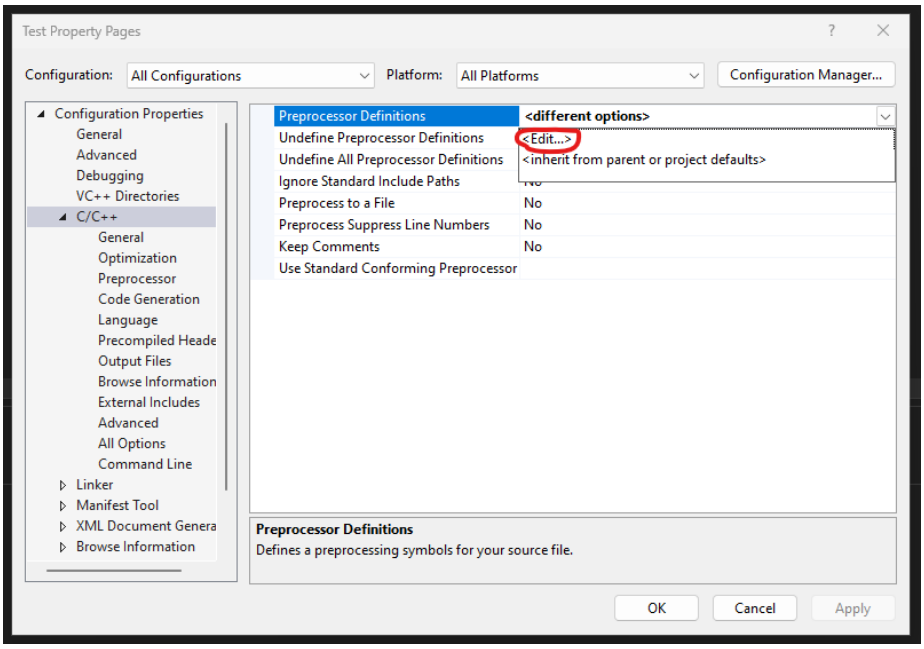
Go to ‘C/C++’ and select ‘Preprocessor’.



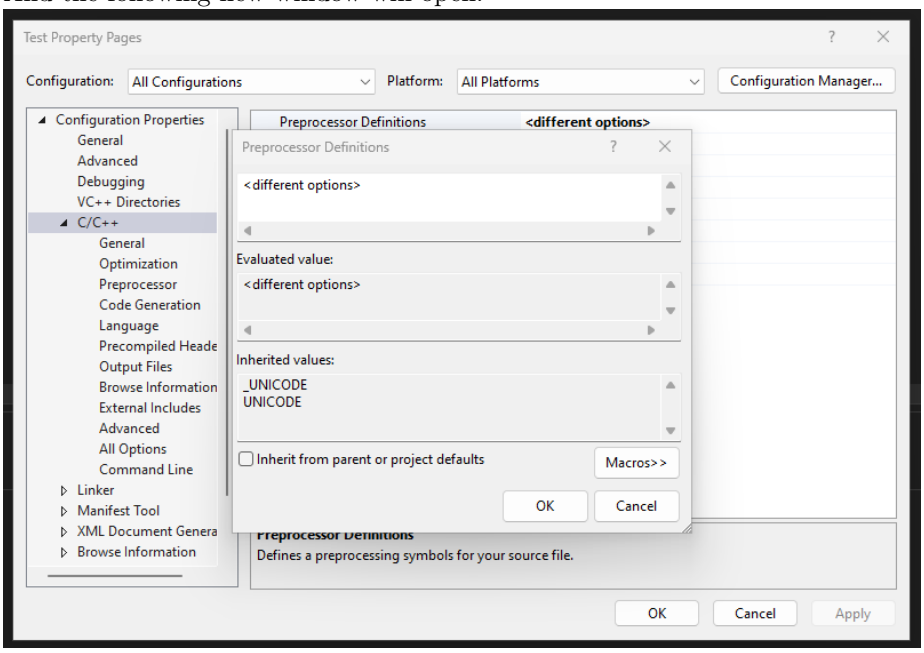
Then click on ‘Preprocessor Definitions’ and on the dropdown arrow.



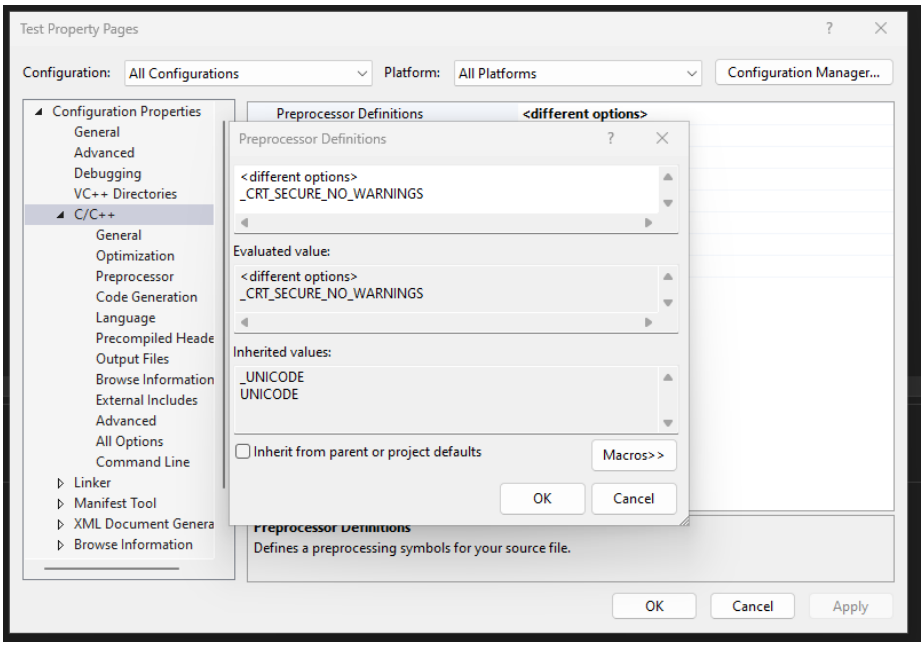
Select ‘Edit’



And the following new window will open:



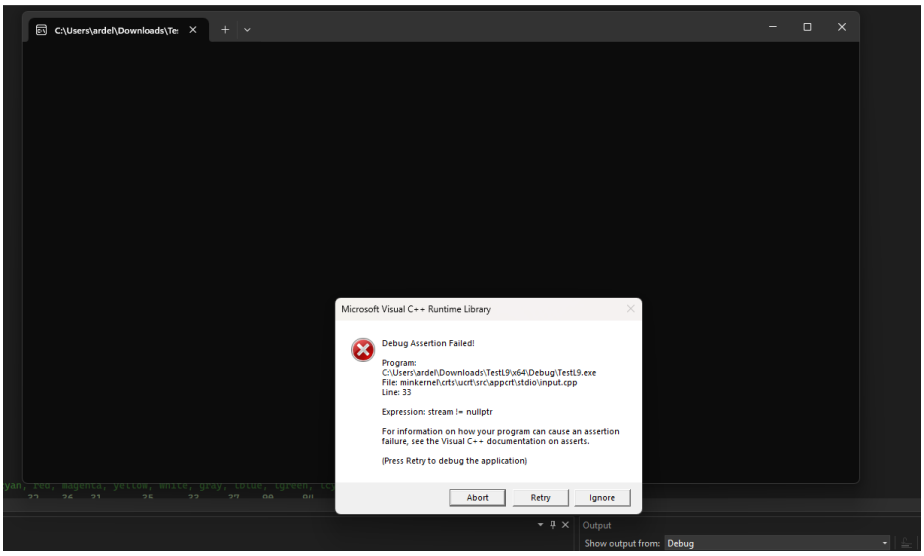
Introduce '_CRT_SECURE_NO_WARNINGS' under '<different options>' as shown below.



Click on 'OK' for all opened windows until you are returned to the main Visual Studio window of the project, and you will be able to run the project.

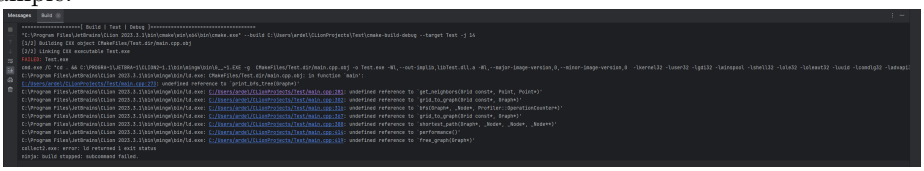
12.3 Visual Studio 'Assertion' Error

This indicates that you have not followed the tutorial. Go back to the first page and make sure that you have moved the files from the 'Downloads' folder to the 'Project' folder. You might also need to *remove* all the files from the Visual Studio IDE and *re-add* them by hand.

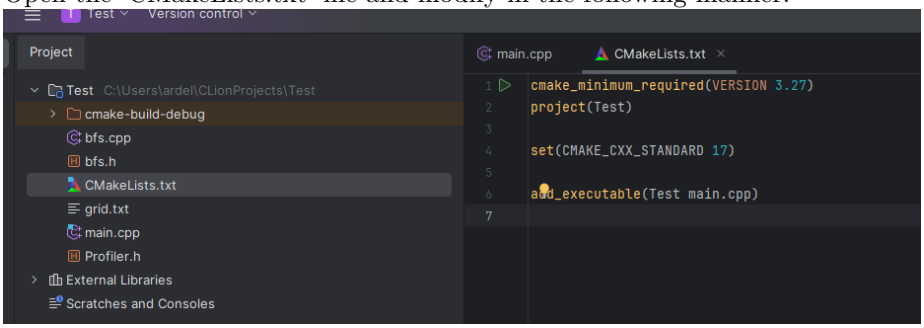


12.4 CLion undefined Error

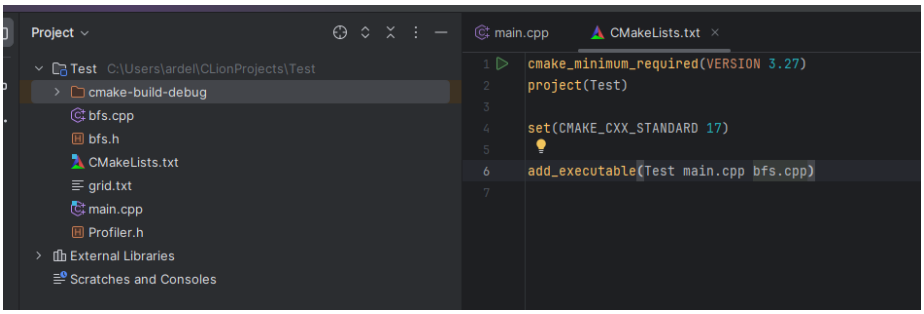
Example:



Solution:
Open the 'CMakeLists.txt' file and modify in the following manner:

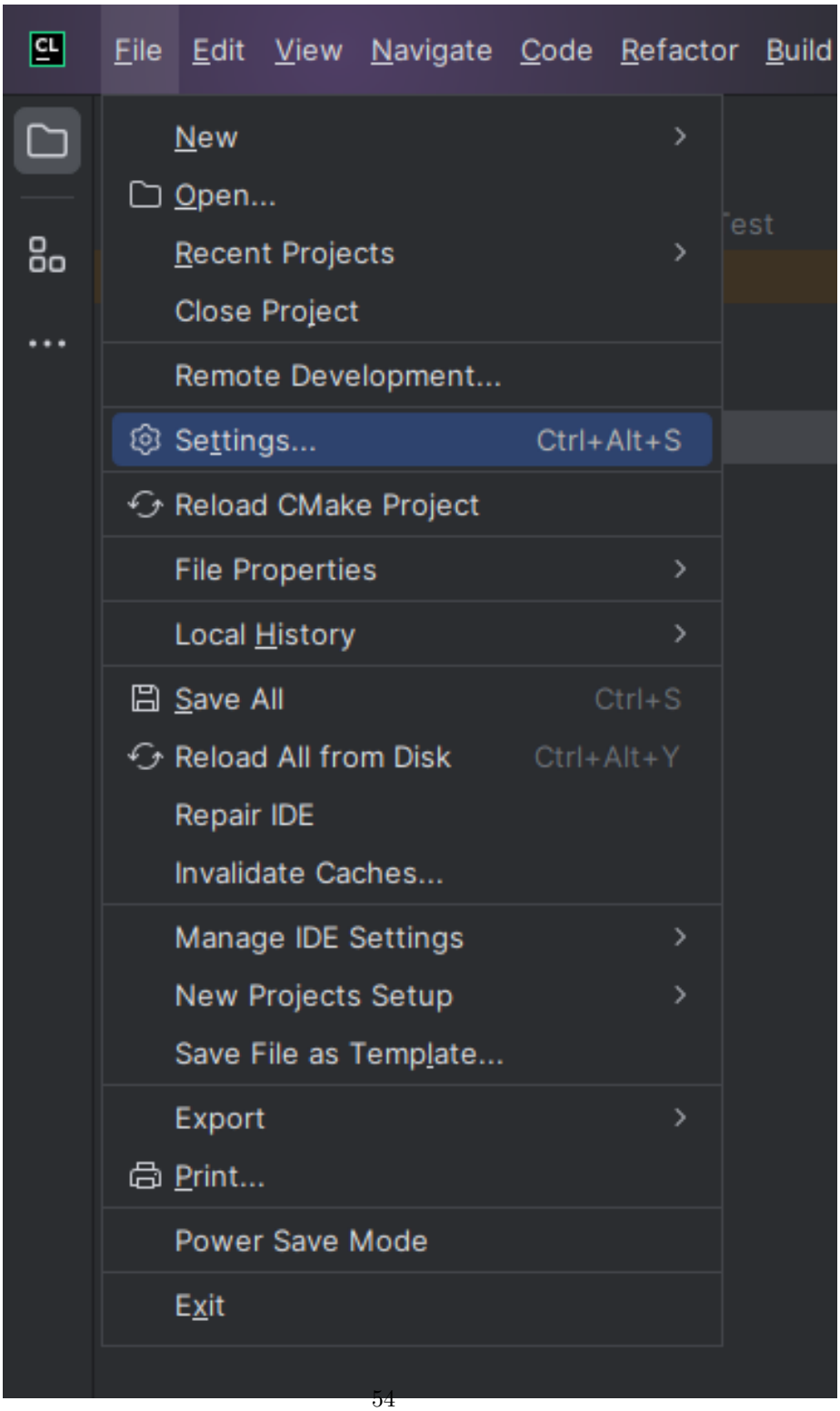


add_executable(Test main.cpp)
into
add_executable(Test main.cpp bfs.cpp)

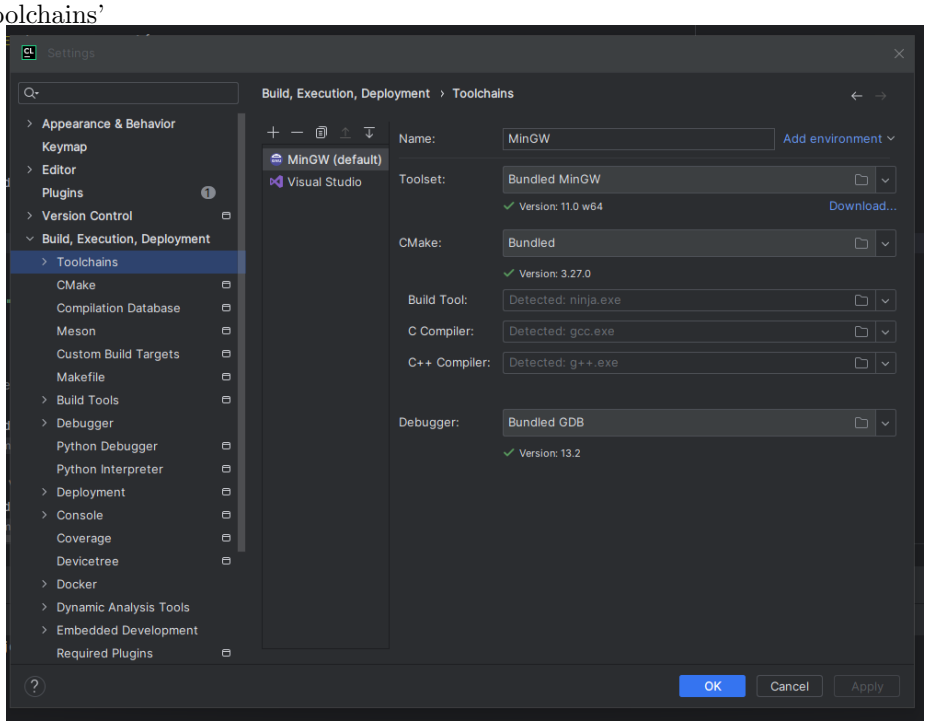


12.5 CLion Visual Studio – Option 1 (slower)

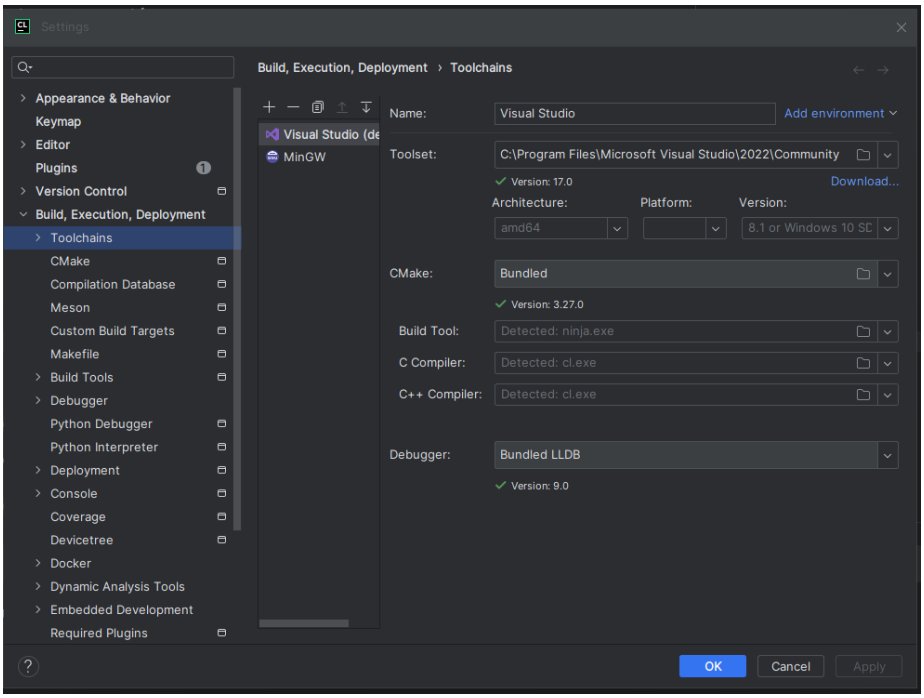
Go to File -> Settings:


File
Edit
View
Navigate
Code
Refactor
Build
New
Open...
Recent Projects
Close Project
Remote Development...
Settings... Ctrl+Alt+S
Reload CMake Project
File Properties
Local History
Save All Ctrl+S
Reload All from Disk Ctrl+Alt+Y
Repair IDE
Invalidate Caches...
Manage IDE Settings
New Projects Setup
Save File as Template...
Export
Print...
Power Save Mode
Exit

In the newly opened window, go to ‘Build, Execution, Deployment’ -> ‘Toolchains’



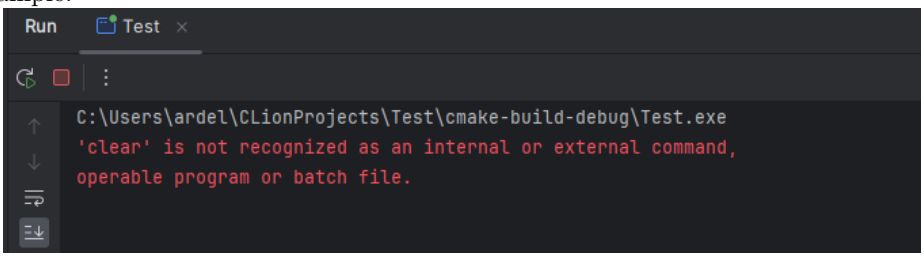
Using the arrows, move Visual Studio as default:



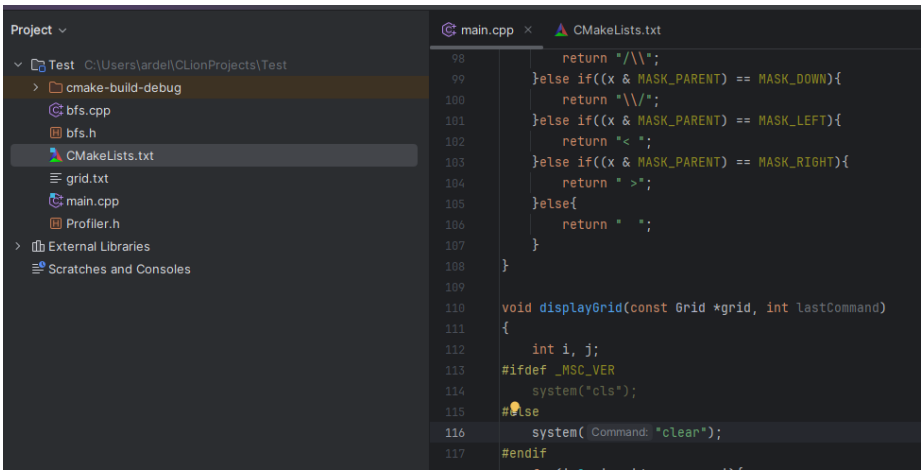
12.6 CLion MinGW – Option 2 (faster, requires external console)

12.6.1 CLion Clear error

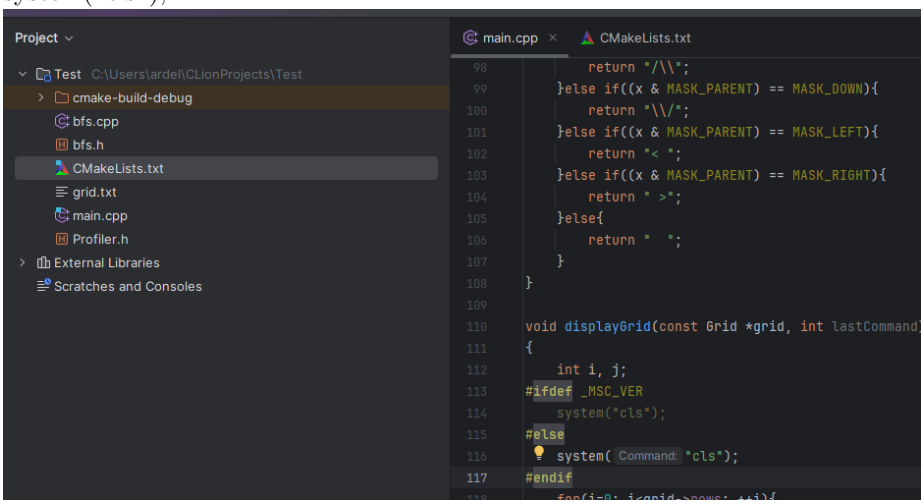
Example:



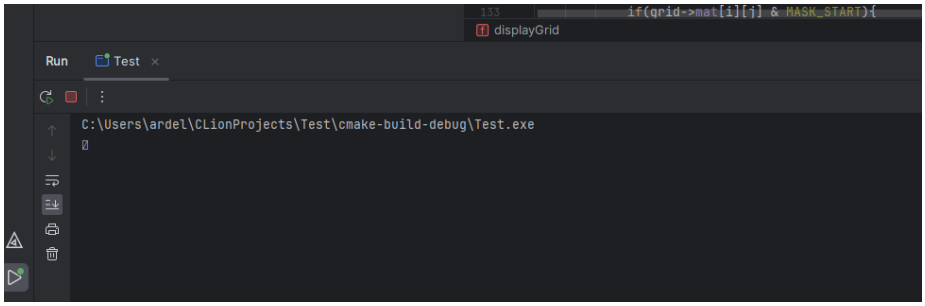
Solution:
Go to the main.cpp file and scroll to lines 113-117 in the displayGrid function.



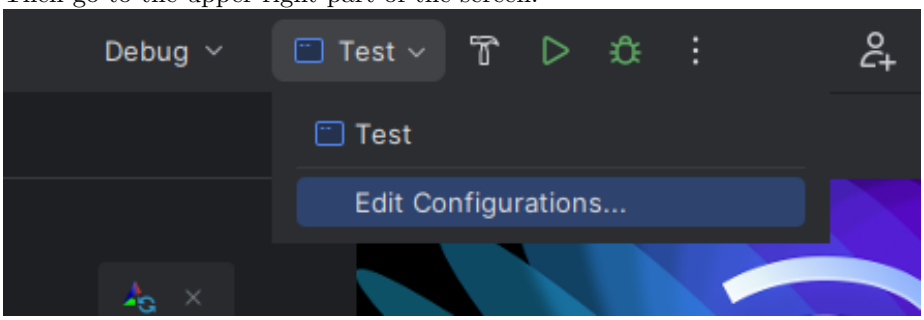
Modify in the following manner:
In the else branch from line 116
system(”clear”);
to
system(”cls”);



12.6.2 CLion not showing grid

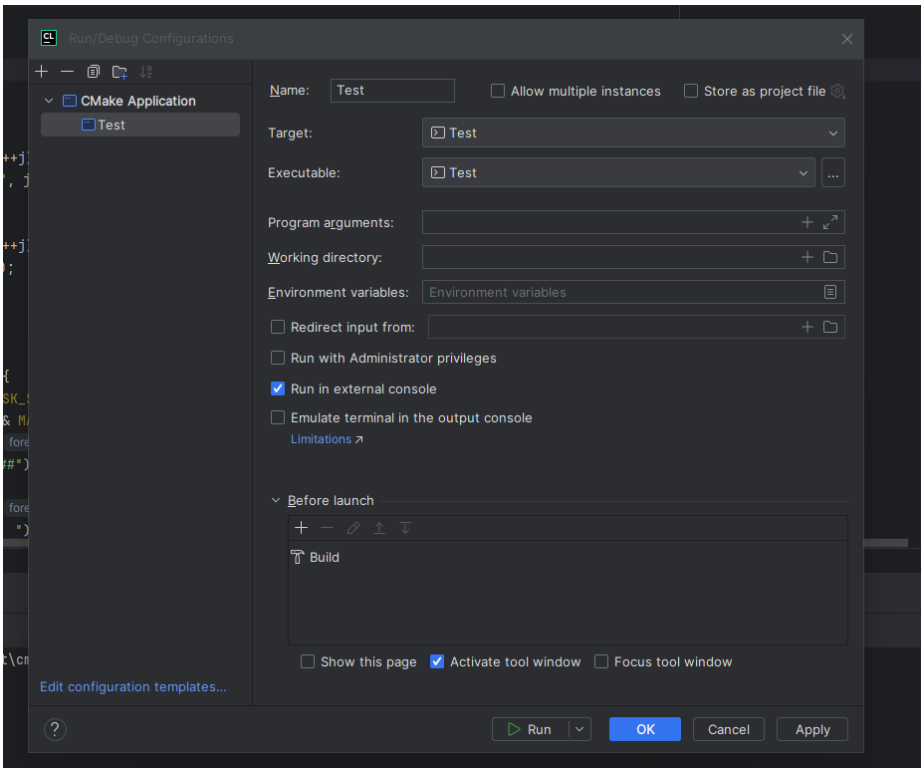


Copy the 'grid.txt' file into the 'cmake-build-debug' folder.
Then go to the upper right part of the screen:



Select 'Edit Configurations' and check the following boxes:

- Run in external console



12.7 Mac run command

g++ main.cpp bfs.cpp -std=c++11 && ./a.out

13 Assignment 10: Depth-first search (DFS)

Allocated time: 2 hours

13.1 Implementation

You are required to correctly and efficiently implement the depth first search algorithm (DFS) (*Chapter 22.3[1]*). For graph representation, you should use adjacency lists. You also have to:

- Implement the Tarjan algorithm for detecting strongly connected components
- Implement topological sorting (*Chapter 22.4[1]*)

13.2 Requirements

13.2.1 DFS (5p)

Exemplify the correctness of your algorithm/implementation by running it on a smaller graph:

- print the initial graph (the adjacency lists)
- print the tree resulted from DFS

13.2.2 Topological sort (1p)

Exemplify the correctness of your algorithm/implementation by running it on a smaller graph:

- print the initial graph (the adjacency lists)
- print a list of nodes sorted topologically (should this list be nonempty/if it is why so?)

13.2.3 Tarjan (2p)

Exemplify the correctness of your algorithm/implementation by running it on a smaller graph:

- print the initial graph (the adjacency lists)
- print all strongly connected components of the graph

13.2.4 Analysis of the DFS performance (2p)

! Before you start to work on the algorithm evaluation code, make sure you have a correct algorithm!

Since, for a graph, both |V| and |E| may vary, and the running time of DFS depends on both, we will make each analysis in turn:

1. Set |V| = 100 and vary |E| between 1000 and 4500, using a 100 increment. Generate the input graphs randomly – make sure you don't generate the same edge twice for the same graph. Run the DFS algorithm for each graph and count the number of operations performed; generate the corresponding chart (i.e., the variation of the number of operations with |E|).
2. Set |E| = 4500 and vary |V| between 100 and 200, using an increment equal to 10. Repeat the procedure above to generate the chart which gives the variation of the number of operations with |V|.

14 Error Fix

14.1 Profiler + VS Code Error

In the file `Profiler.h`, you have the following lines starting at line 11:

```
// OS detection
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)
    #define PROFILER_WINDOWS
#elif __APPLE__
    #define PROFILER_OSX
#elif __linux__
    #define PROFILER_LINUX
#endif

#ifdef PROFILER_WINDOWS
    #include <Windows.h>
    #include <Shellapi.h>
#else
    #include <unistd.h>
#endif
```

You should modify them as follows:

```
// OS detection
#if defined(__MINGW32__) || defined(__CYGWIN__)
    #define PROFILER_VSCODE
#elif defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)
    #define PROFILER_WINDOWS
#elif defined(__APPLE__)
    #define PROFILER_OSX
#elif defined(__linux__)
    #define PROFILER_LINUX
#endif

#if defined(PROFILER_WINDOWS) || defined(PROFILER_VSCODE)
    #include <Windows.h>
    #include <Shellapi.h>
#else
    #include <unistd.h>
#endif
```

Below are the screenshots (approximately 80% of text width), framed as figures:

```
 (DWORD, PCVOID, DWORD, DWORD, LPWSTR, DWORD, va_list *);
                                                ^~~~~~~

In file included from c:\users\ardel\mingw\include\windows.h:48:0,
                 from C:\Users\ardel\Downloads\Test\Profiler.h:21,
                 from C:\Users\ardel\Downloads\Test\example1.cpp:1:
c:\users\ardel\mingw\include\winuser.h:4351:50: error: 'va_list' has not been declared
 WINUSERAPI int WINAPI wvsprintfA (LPSTR, LPCSTR, va_list arglist);
                                                  ^~~~~~~
c:\users\ardel\mingw\include\winuser.h:4352:52: error: 'va_list' has not been declared
 WINUSERAPI int WINAPI wvsprintfW (LPWSTR, LPCWSTR, va_list arglist);
                                                    ^~~~~~~
In file included from C:\Users\ardel\Downloads\Test\example1.cpp:1:0:
C:\Users\ardel\Downloads\Test\Profiler.h: In member function 'int Profiler::showReport()':
C:\Users\ardel\Downloads\Test\Profiler.h:11263:35: error: 'localtime_s' was not declared in this scope
         localtime_s(&now, &crtTime);
                                   ^
C:\Users\ardel\Downloads\Test\Profiler.h:11282:9: error: '_snprintf_s' was not declared in this scope
         );
         ^
C:\Users\ardel\Downloads\Test\Profiler.h:11284:40: error: 'fopen_s' was not declared in this scope
         fopen_s(&fout, reportName, "wb");
                                        ^

Build finished with error(s).
```

Figure 7: Initial error in VS Code Profiler

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#endif

#define _CRT_SECURE_NO_DEPRECATE
#define _CRT_SECURE_NO_WARNINGS

//OS detection
#if defined(__MINGW32__) || defined(__CYGWIN__)
#   define PROFILER_LINUX
#elif defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)
#   define PROFILER_WINDOWS
#elif __APPLE__ ||
#   define PROFILER_OSX
#elif __linux__
#   define PROFILER_LINUX
#endif

#ifdef PROFILER_WINDOWS
#   include <windows.h>
#   include <Shellapi.h>
#else
#   include <unistd.h>
#endif

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

Figure 8: Original screenshot (before fix)

```
Profiler.h   example1.cpp
Profiler.h > ...

#ifndef _PROFILER_H
#define _PROFILER_H

#ifndef _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
#endif

#define _CRT_SECURE_NO_DEPRECATE
#define _CRT_SECURE_NO_WARNINGS

//OS detection
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)
#   define PROFILER_WINDOWS
#elif __APPLE__ ||
#   define PROFILER_OSX
#elif __linux__
    define PROFILER_LINUX
#endif

#ifdef PROFILER_WINDOWS
#   include <windows.h>
#   include <Shellapi.h>
#else
#   include <unistd.h>
#endif

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#include <typeinfo>
#include <map>
#include <vector>
#include <algorithm>
#include <functional>
#include <string>
```

Figure 9: Screenshot after update (after fix)

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.