# Homework: Streams and Files

This document defines the homework assignments from the ["Advanced C#" Course @ Software University](https://softuni.bg/courses/advanced-csharp). Please submit as homework a single zip / rar / 7z archive holding the solutions (source code) of all below described problems. The solutions should be written in C#.

## Odd Lines

Write a program that reads a text file and prints on the console its odd lines. Line numbers starts from 0. Use **StreamReader**.

## Line Numbers

Write a program that reads a text file and inserts line numbers in front of each of its lines. The result should be written to another text file. Use **StreamReader** in combination with **StreamWriter**.

## Word Count

Write a program that reads a list of words from the file **words.txt** and finds how many times each of the words is contained in another file **text.txt**. Matching should be **case-insensitive**.

Write the results in file **results.txt**. Sort the words by frequency in descending order. Use **StreamReader** in combination with **StreamWriter**.

|  |  |  |
| --- | --- | --- |
| **words.txt** | **text.txt** | **result.txt** |
| quick  is  fault | -I was quick to judge him, but it wasn't his fault.  -Is this some kind of joke?! Is it?  -Quick, hide here…It is safer. | is - 3  quick - 2  fault - 1 |

## Copy Binary File

Write a program that copies the contents of a binary file (e.g. image, video, etc.) to another using **FileStream**. You are **not allowed** to use the **File** class or similar helper classes.

## Slicing File

Write a program that takes any file and slices it to **n** parts. Write the following methods:

* **Slice(string sourceFile, string destinationDirectory, int parts)** - slices the given source file into **n** parts and saves them in **destinationDirectory**.

|  |  |
| --- | --- |
| **Source File** | **Destination Directory** |
| parts = 5C:\Users\Jamal\AppData\Local\Microsoft\Windows\INetCache\Content.Word\solid-logger.png | C:\Users\Jamal\AppData\Local\Microsoft\Windows\INetCache\Content.Word\sliced-png.png |

* **Assemble(List<string> files, string destinationDirectory)** -combines all files into one, in the order they are passed, and saves the result in **destinationDirectory**.

|  |  |
| --- | --- |
| **Source Files** | **Destination Directory** |
| C:\Users\Jamal\AppData\Local\Microsoft\Windows\INetCache\Content.Word\sliced-png.png | C:\Users\Jamal\AppData\Local\Microsoft\Windows\INetCache\Content.Word\assembled.png |

Use **FileStreams**. You are **not allowed** to use the **File** class or similar helper classes.

## Zipping Sliced Files

Modify your previous program to also **compress** the bytes while slicing parts and **decompress** them when assembling them back to the original file. Use **GzipStream**.

**Tip**: When getting files from directory, make sure you only get files with **.gz** extension (there might be hidden files).

|  |  |  |
| --- | --- | --- |
| **Source File** | **Compressed & Sliced** | **Decompressed & Assembled** |
| parts = 5C:\Users\Jamal\AppData\Local\Microsoft\Windows\INetCache\Content.Word\solid-logger.png |  | C:\Users\Jamal\AppData\Local\Microsoft\Windows\INetCache\Content.Word\assembled.png |

## Directory Traversal

Traverse a given directory for all files with the given extension. Search through the first level of the directory only and write information about each found file in **report.txt**.

The files should be **grouped** by their **extension**. Extensions should be ordered by the **count of their files** (from most to least). If two extensions have equal number of files, order them by **name**.

Files under an extension should be ordered by their **size**.

**report.txt** should be saved on the **Desktop**. Ensure the desktop path is always valid, regardless of the user.

|  |  |  |
| --- | --- | --- |
| **Input** | **Directory View** | **report.txt** |
| ../../ |  | .cs  --Mecanismo.cs - 0.994kb  --Program.cs - 1.108kb  --Nashmat.cs - 3.967kb  --Wedding.cs - 23.787kb  --Program - Copy.cs - 35.679kb  --Salimur.cs - 588.657kb  .txt  --backup.txt - 0.028kb  --log.txt - 6.72kb  .asm  --script.asm - 0.028kb  .config  --App.config - 0.187kb  .csproj  --01. Writing-To-Files.csproj - 2.57kb  .js  --controller.js - 1635.143kb  .php  --model.php - 0kb |

## \* Full Directory Traversal

Modify your previous program to **recursively traverse** the **sub-directories** of the starting directory as well.

# Problems for Champions

## \* Disk

**This problem is from the C# Basics Exam (28 April 2014). You can test your solution** [**here**](https://judge.softuni.bg/Contests/Practice/Index/10#2)**.**

In geometry, a **disk** is the region in a plane bounded by a circle (it also **includes** the circle itself). Your task is to **print a disk on the console** by a given **radius R** in a square **field of size** **N** x **N** (see the examples below).

### Input

The input data should be read from the console.

* On the first line of the input you will be given the size of the field **N**. On the second line of the input you will be given the radius of the disk **R**.
* The disk’s center **is the center point** of the field (it will always exist, because N is odd).

The input data will always be valid and in the format described. There is no need to check it explicitly.

### Output

The output should be printed on the console. You should print the disk on the console following the examples below.

* Your output must consist of **N** rows, each containing **N** characters. Each character represents a point in the field. For every point you must output one of two possible states – dot '**.**' if the point lies outside of the disk and asterisk '**\***' if the point lies within the disk.

**Hint:** In order to check whether a point is inside or outside of a circle, you may calculate the distance from the point to the center of the field by the **Pythagorean** **Theorem** (see <http://goo.gl/HwqOuU>).

### Constraints

* The number **N** is a positive **odd** integer in the range [3 … 39], inclusive.
* The number **R** is a positive integer between 1 and N/2 (floor (N/2)), inclusive. This means that the disk will always fit in the field, without crossing its sides.
* Allowed working time for your program: 0.1 seconds.
* Allowed memory: 16 MB.

### Examples

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |  | **Input** | **Output** |
| 5  2 | ..\*..  .\*\*\*.  \*\*\*\*\*  .\*\*\*.  ..\*.. | 9  3 | .........  ....\*....  ..\*\*\*\*\*..  ..\*\*\*\*\*..  .\*\*\*\*\*\*\*.  ..\*\*\*\*\*..  ..\*\*\*\*\*..  ....\*....  ......... | 11  1 | ...........  ...........  ...........  ...........  .....\*.....  ....\*\*\*....  .....\*.....  ...........  ...........  ...........  ........... | 19  6 | ...................  ...................  ...................  .........\*.........  ......\*\*\*\*\*\*\*......  .....\*\*\*\*\*\*\*\*\*.....  ....\*\*\*\*\*\*\*\*\*\*\*....  ....\*\*\*\*\*\*\*\*\*\*\*....  ....\*\*\*\*\*\*\*\*\*\*\*....  ...\*\*\*\*\*\*\*\*\*\*\*\*\*...  ....\*\*\*\*\*\*\*\*\*\*\*....  ....\*\*\*\*\*\*\*\*\*\*\*....  ....\*\*\*\*\*\*\*\*\*\*\*....  .....\*\*\*\*\*\*\*\*\*.....  ......\*\*\*\*\*\*\*......  .........\*.........  ...................  ...................  ................... |

## \* Paint Ball

**This problem is from the C# Basics Exam (19 December 2014). You can test your solution** [**here**](https://judge.softuni.bg/Contests/Practice/Index/52#4)**.**

You are given a painting canvas of size 10 x 10, divided into 100 cells. Initially, the canvas is white (all cells have a value of **1**). You shoot black and white paint balls with different sizes at the canvas. White is represented by **1**s and black is represented by **0**s. You alternate between black and white paint after each shot; the first shot is always with black paint (**0**s), the second is white (**1**s), the third is black again and so on. You will be given each shot's impact row and column coordinates as well as the ball's radius. The impact area is a square, its center is the impact cell; all cells in the impact area change values to either **0** or **1**, depending on the color of the paint.

After you run out of ammo (when you receive the string "**End**" from the console) the canvas will be some combination of 1s and 0s. Each row of the canvas represents a binary integer number. Your task is to find the **sum of the 10 numbers** and print it to the console. An example where a single shot with parameters "4 5 2" was fired is shown below. The impact cell is shaded black, the splashed cells in the impact area are shaded grey.

**Input**

The input data is read from the console.

* It consists of a **random number of lines**. The input **ends with the string "End"**.
* Each line will hold **three numbers** – the **row and column** of the cell where the ball lands and the **radius of the ball**, all separated from each other by a single space.

The input data will always be valid and in the format described. There is no need to check it explicitly.

**Output**

The output data must be printed on the console.

* On the only output line you must print the **sum of the 10 rows of the canvas in decimal format.**

**Constraints**

* The **number of shots** will be in the range [1…25].
* The **rows** and **columns** are integer numbers in the range [0…9].
* The **radius of the ball** will be an integer between 0 (single cell) and 10 (large splash area damage).
* Time limit: 0.25 seconds. Allowed memory: 16 MB.

**Examples**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Number |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1023 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1023 |
| 2 | 1 | 1 | **0** | **0** | **0** | **0** | **0** | 1 | 1 | 1 | 775 |
| 3 | 1 | 1 | **0** | **0** | **0** | **0** | **0** | 1 | 1 | 1 | 775 |
| 4 | 1 | 1 | **0** | **0** | **0** | **0** | **0** | 1 | 1 | 1 | 775 |
| 5 | 1 | 1 | **0** | **0** | **0** | **0** | **0** | 1 | 1 | 1 | 775 |
| 6 | 1 | 1 | **0** | **0** | **0** | **0** | **0** | 1 | 1 | 1 | 775 |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1023 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1023 |
| 9 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1023 |
|  |  |  |  |  |  |  |  | **sum =** | | | **8990** |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 5 2  End | 8990 |
|  |  |
| **Input** | **Output** |
| 1 2 5 | 5118 |
| 3 3 1 |  |
| 0 6 4 |  |
| 0 0 0 |  |
| 8 9 2 |  |
| 1 7 2 |  |
| End |  |

## \* Couples Frequency

**This problem is from the Java Basics Exam (26 May 2014). You can test your solution** [**here**](https://judge.softuni.bg/Contests/Practice/Index/12#3)**.**

Write a program that reads a sequence of **n integers** and calculates and prints the **frequencies of all couples** of two consecutive numbers. For example, for the input sequence **{ 3 4 2 3 4 2 1 12 2 3 4 }**, we have 10 couples (6 distinct), shown on the right with their occurrence counts and frequencies (in percentage).

|  |  |  |
| --- | --- | --- |
| **Couple** | **Occurrences** | **Percentage** |
| 3 4 | 3 times | 30.00% |
| 4 2 | 2 times | 20.00% |
| 2 3 | 2 times | 20.00% |
| 2 1 | 1 times | 10.00% |
| 1 12 | 1 times | 10.00% |
| 12 2 | 1 times | 10.00% |

### Input

The input data should be read from the console. At the first line, we have the **input sequence of integers**, separated by a space.

The input data will always be valid and in the format described. There is no need to check it explicitly.

### Output

Print all **distinct couples** of two consecutive numbers (without duplicates) found in the input sequence (from left to right) along with their **frequency of appearance** in the input sequence (in **percentages**, with two decimal digits, with traditional rounding). Use the format: "**couple -> percentage**" (see the examples below). Beware of **formatting**!

### Constraints

* All input numbers will be integers in the range [-100 000 … 100 000].
* The **count** of the numbers will be in the range [2..1000].
* Time limit: 0.5 sec. Memory limit: 16 MB.

### Examples

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** |  | **Input** |  | **Input** |
| 3 4 2 3 4 2 1 12 2 3 4 | 5 10 5 10 10 5 5 10 5 10 10 5 | 10 20 10 10 10 |
| **Output** | **Output** | **Output** |
| 3 4 -> 30.00%  4 2 -> 20.00%  2 3 -> 20.00%  2 1 -> 10.00%  1 12 -> 10.00%  12 2 -> 10.00% | 5 10 -> 36.36%  10 5 -> 36.36%  10 10 -> 18.18%  5 5 -> 9.09% | 10 20 -> 25.00%  20 10 -> 25.00%  10 10 -> 50.00% |

## \* Labyrinth Dash

**This problem is from the Java Basics (11 May 2015). You can test your solution** [**here**](https://judge.softuni.bg/Contests/Practice/Index/86#2)**.**

Enough hard problems. Let’s play a game! You will be given the layout of a labyrinth (a two-dimensional array) and a series of moves. Your task is to navigate the labyrinth and **print the outcome of each move**.

On the first line of input you will be given the **number N representing the count of rows** of the labyrinth. On each of the next N lines you will receive a **string containing the layout of the given row**. On the last line of input you will receive **a string containing the moves** you need to make. Each move will be one of the following symbols: **"v" (move down), "^" (move up), "<" (move left) or ">" (move right).** The string will not contain any other characters.

The **player starts with 3 lives and begins the journey at position (0, 0).** When you make a move, there can be several different outcomes: **1) Hit a wall** – a wall is represented by the symbols **"\_" (underscore) and "|" (pipe).** Hitting a wall means the player stays in place; in this case you should print on the console **“Bumped a wall.”** **2) Land on an obstacle** – obstacles are the following symbols: **"@", "#", "\*"**. If you move to a position containing one of these symbols the player loses a life point and you should print **"Ouch! That hurt! Lives left: X"** on the console. If the player is left with 0 lives, the game ends and you should print **"No lives left! Game Over!"** **3) Get a new life** – when you land on the symbol **"$"** the player receives an additional life point. Print **"Awesome! Lives left: X"** on the console. Additional lives ('$') are removed once the player passes through the cell (i.e. they are replaced with dots). **4) Drop out of the labyrinth** – if you land on an empty cell (one containing a space), or outside the boundaries of the array, the game ends and you should print **"Fell off a cliff! Game Over!"** **5) Land on the symbol "." (dot)** – move the player to the new position, nothing else happens; print on the console **"Made a move!"**

When the game ends (either the player has lost or all moves were made), print **"Total moves made: X"**.

### Input

* The input data should be read from the console.
* On the first line of input you will receive the number N – number of rows of the labyrinth.
* On the next N lines you will receive the layout of the labyrinth.
* On the last line you will receive the moves you need to make as a string.
* The input data will always be valid and in the format described. There is no need to check it explicitly.

### Output

* The output should be printed on the console.
* For each outcome print the required output as described above.

### Constraints

* The number N will be an integer in the range [1 … 15].
* The labyrinth will contain only the symbols – "\_", "|", "@", "#", "\*", "$", " " (single space), ".".
* The string containing the moves will be comprised of the following symbols only – "v", "^", "<", and ">".
* Allowed working time for your program: 0.5 seconds. Allowed memory: 16 MB.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  .|  ..|  \*.$ .  ###...  \_\_\_\_\_\_  >v>>vv>>>^^^<< | Bumped a wall.  Made a move!  Made a move!  Bumped a wall.  Made a move!  Ouch! That hurt! Lives left: 2  Ouch! That hurt! Lives left: 1  Made a move!  Made a move!  Fell off a cliff! Game Over!  Total moves made: 8 | Player starts at (0, 0). First move is ">" (right), which takes the player into a wall. Next, he moves down and right. The next move is right again and he hits another wall. He then moves down twice, on the second move he lands on an obstacle ("#") and loses a life point. He then moves right and loses another life. Two moves to the right are followed by a move upwards which takes him out of the labyrinth (empty cell), so the game is over. The total number of moves where the player actually changed position is 7. |