# C++ Programming: Judge Assignment 1 (JA1)

The following tasks should be submitted to the SoftUni Judge system, which will be open starting Saturday, 18 March 2017, 10:00 (in the morning) and will close on Sunday, 26 March 2017, 23:59. You will be provided with a link to the “contest” (where you will submit the assignment) later.

For this assignment, the code for each task should be a single C++ file, the contents of which you copy-paste into the Judge system.

Please be mindful of the strict input and output requirements for each task, as well as any additional requirements on running time, used memory, etc., as the tasks are evaluated automatically and not following the requirements strictly may result in your program’s output being evaluated as incorrect, even if the program’s logic is mostly correct.

You can use C++03 and C++11 features in your code.

Unless explicitly stated, any integer input fits into int and any floating-point input can be stored in double

The tasks here do have memory and time restrictions, but that is just to introduce the exam format. In this first assignment, the input data is small enough and the tasks simple enough that you don’t need to think about optimizing your code too much. Focus on code that works and is easy to read and modify vs. super-optimized code.

## Task 1 – Average Color (JA1-Task-2-Average-Color)

Colors in computers are often represented in the RGB format, with an 8-bit (0-255) number per each of the R (red), the G (green) and the B (blue) color channels. An example of an RGB color would be the triplet 255, 0, 0, which is the color red (red channel has the highest value, the other channels are “dark” as they have 0s as values) and 128, 128, 0 is the olive color (yeah, I know, olives are vegetables, not colors, but that’s how color experts seem to define it. It’s a very dark, greenish yellow. Anyway, that’s beside the point)

A common way of writing this format (e.g. in CSS) is to use the hexadecimal triplet notation, or Hex Code – we write a “#” in front of a string of 6 alphanumeric characters (letters and/or numbers), each 2 of which represent a hexadecimal number from 0 to 255, with 00 being equal to 0 and FF being equal to 255. For the above examples, red would be written as #ff0000, and olive would be written as #808000

Write a program which, given two colors written in the Hex Code format (with a “#” in front of exactly 6 hexadecimal digits), computes the “average” between the colors – by calculating the average of each channel separately – and prints the resulting color in the same Hex Code format to the console. For computing the average of the channels of two colors, just sum the numbers of the channels and divide them by 2 (integer division, i.e. round down to the nearest integer, i.e. take the floor value).

That is, if the first color has the components red1, green1, blue1,   
and the second color has the components red2, green2, blue2,   
then the “average” of those two colors is calculated as   
(red1 + red2) / 2, (green1 + green2) / 2, (blue1 + blue2) / 2.   
For our 2 example colors #ff0000 and #808000 above, the average would be #bf4000   
(because ff = 255, 80 = 128, (255 + 128) / 2 = 383 / 2 = 191, which is bf in hexadecimal, for the red channel, and 00 = 0, 80 = 128, (128 + 0) / 2 = 64, which is 40 in hexadecimal, for the green channel)  
Side note: If you’re interested, that’s not exactly how color “averaging” is done in real systems – the correct approach would be raising each channel’s value to the power of 2 before doing the average and then taking the square root of the result, but we don’t want to do that in this task for the sake of simplicity, if you want to learn more, see this video: <https://youtu.be/LKnqECcg6Gw>

### Input

Two Hex Code color values on the same line, separated by a single space. Any letters in the input will be lowercase

### Output

A single Hex Code color value representing the “average” of the two colors. Any letters in the output must be lowercase.

### Restrictions

The total running time of your program should be no more than 0.1s

The total memory allowed for use by your program is 5MB

### Example I/O

|  |  |
| --- | --- |
| Example Input | Expected Output |
| #ff0000 #808000 | #bf4000 |
| #2b00b5 #0ff1ce | #1d78c1 |

## Task 2 – Similar Words (JA1-Task-2-Similar-Words)

We will consider two words – W1 and W2 – “similar” if they have the same length, start with the same letter, and a minimum percentage – P – of their letters match (a letter in W1 matches a letter in W2 if the two letters are the same symbol and are at the same position in both words).

For example, if W1 = “kittens” and W2 = “kidding”, the matching letters would be k, i, and n. That gives us 3 matching letters out of 7 letters, which is about 42.8% of the letters. If P = 40, then we would say the words match. If P = 43, we would say the words don’t match.

A word is any uninterrupted sequence of lowercase English letters (a-z). So, punctuation or spaces surround a word from both “sides” (unless the word is at the start and/or end of the text).

Write a program, which reads a line of lowercase text T (letters and punctuation, but no numbers), a lowercase word W (letters only) and an integer number P and prints out how many words similar to W there are in the text.

### Input

The text T, containing lowercase English letters (a-z) and punctuation (.,;!? and space) will be entered on the first line of input

The second line of input will contain a single word W, containing only lowercase English letters (a-z), followed by a single space and the integer P.

### Output

A single line containing an integer number – the number of words similar to W in the text T, considering the percentage P

### Restrictions

The text T will be no longer than 500 symbols and no shorter than 1 symbol.

The word W will be no longer than 30 symbols and no shorter than 1 symbol.

P will be between 1 and 100, inclusive.

The total running time of your program should be no more than 0.1s

The total memory allowed for use by your program is 5MB

### Example I/O

|  |  |
| --- | --- |
| Example Input | Expected Output |
| kittens,kidding.  kittenz 40 | 2 |
| abcd  dcba 1 | 0 |
| aaaa aabb abbb baaa  aaaa 50 | 2 |
| aaaa  aa 1 | 0 |

## Task 3 – Fill Matrix (JA1-Task-3-Fill-Matrix)

You are given a matrix (2D array) of lowercase alphanumeric characters (a-z, 0-9), a starting position – defined by a start row startRow and a start column startCol – and a filling symbol fillChar. Let’s call the symbol originally at startRow and startCol the startChar. Write a program, which, starting from the symbol at startRow and startCol, changes to fillChar every symbol in the matrix which:

* is equal to startChar AND
* can be reached from startChar by going up (row – 1), down (row + 1), left (col – 1) and right (col + 1) and “stepping” ONLY on symbols equal startChar

So, you basically start from startRow and startCol and can move either by changing the row OR column (not both at once, i.e. you can’t go diagonally) by 1, and can only go to positions which have the startChar written on them. Once you find all those positions, you change them to fillChar.

In other words, you need to implement something like the Fill tool in MS Paint, but for a 2D char array instead of a bitmap.

### Input

On the first line, two integers will be entered – the number R of rows and number C of columns.

On each of the next R lines, C characters separated by single spaces will be entered – the symbols of the Rth row of the matrix, starting from the 0th column and ending at the C-1 column.

On the next line, a single character – the fillChar – will be entered.

On the last line, two integers – startRow and startCol – separated by a single space, will be entered.

### Output

The output should consist of R lines, each consisting of exactly C characters, NOT SEPARATED by spaces, representing the matrix after the fill operation has been finished.

### Restrictions

0 < R, C < 20   
0 <= startRow < R   
0 <= startCol < C

All symbols in the input matrix will be lowercase alphanumerics (a-z, 0-9). The fillChar will also be alphanumeric and lowercase.

The total running time of your program should be no more than 0.1s

The total memory allowed for use by your program is 5MB

### Example I/O

|  |  |
| --- | --- |
| Example Input | Expected Output |
| 5 3  a a a  a a a  a b a  a b a  a b a  x  0 0 | xxx  xxx  xbx  xbx  xbx |
| 5 3  a a a  a a a  a b a  a b a  a b a  x  2 1 | aaa  aaa  axa  axa  axa |
| 5 6  o o 1 1 o o  o 1 o o 1 o  1 o o o o 1  o 1 o o 1 o  o o 1 1 o o  3  2 1 | oo11oo  o1331o  133331  o1331o  oo11oo |
| 5 6  o o o o o o  o o o 1 o o  o o 1 o 1 1  o 1 1 w 1 o  1 o o o o o  z  4 1 | oooooo  ooo1oo  oo1o11  o11w1z  1zzzzz |
| 5 6  o 1 o o 1 o  o 1 o o 1 o  o 1 1 1 1 o  o 1 o w 1 o  o o o o o o  z  4 0 | z1oo1z  z1oo1z  z1111z  z1zw1z  zzzzzz |

## Task 4 – Named Operations (JA1-Task-4-Named-Operations)

We have an imaginary machine that has 4 operations, each of which works on an array of positive integers, more specifically on a range of elements of that array, defined by an inclusive start and an exclusive end index, and returns an integer. In C++, each the operations might look similar to this:

int operation(int array[], int startIndex, int endIndex);

Each of the operations has a number, the first being 0, the second 1, the third 2 and the fourth – 3. The operations are the following:

1. Calculates the sum of the elements from array[startIndex] to array[endIndex - 1]
2. Calculates the average of the elements from array[startIndex] to   
   array[endIndex - 1] as an integer, by ignoring the floating-point part.
   1. E.g. the average of the numbers **4** and **5** would be calculated as 4
3. Finds the minimum (smallest) of the elements from array[startIndex] to array[endIndex - 1]
4. Finds the maximum (largest) of the elements from array[startIndex] to array[endIndex - 1]

To use the operations, the user must first input an array of positive integers. Then, the user must define names for the operations. A user can define multiple names for the same operation. When names have been given for the operations, the user can start calling operations, by typing in the name of an operation followed by two numbers, defining the start and end index for the operation. When the user wants to stop doing operations, they input the word end and the machine prints out a summary of the operations and their results

Write a program which simulates our imaginary machine.

### Input

The first line will be the array of integer numbers, separated by single spaces.

The second line will contain a single number **P** – the number of operation names the user wants to input.

Each of the following **P** lines will contain an operation name, written with lowercase English letters (a-z) and the number of the operation which is being named, separated by a single space.

Each of the following lines will contain a single operation call. Each operation call will begin with a name of an operation (which was entered previously), a start index and an end index, separated by single spaces.

The last line of input will contain only the word **end**

### Output

A single line listing all the operations done, with the following format, where **C** is the number of all operation calls done and **operationString** is the info of an operation call (see below)

* [C]{operationString,operationString,…,operationString}
* i.e. [C]{ followed by an operationString for each call, in the order they were called from the console, comma-separated, followed by }

The info of an operation call, called an **operationString**, has the following format, where **name** is the name of the operation, **startIndex** is the start index number, **endIndex** is the end index number and **result** is the integer value of the result of the operation:

* name(startIndex,endIndex)=result

### Restrictions

**0 < C < 10**

**0 < P < 20**

The input array will be between **1** and **20** elements long

startIndex < endIndex

**startIndex** and **endIndex** will always be correct indices inside the input array

Operation names will be between **1** and **20** lowercase English letters (**a-z**)

### Example I/O

|  |  |
| --- | --- |
| Example Input | Expected Output |
| 5 4 3 2 1  4  avg 1  sum 0  min 2  max 3  max 0 5  sum 0 5  sum 0 1  min 0 5  avg 0 2  sum 0 5  end | [6]{max(0,5)=5,sum(0,5)=15,sum(0,1)=5,min(0,5)=1,avg(0,2)=4,sum(0,5)=15} |
| 1 2 3 4  3  sumo 0  sum 0  maximum 3  sum 0 4  sumo 0 4  end | [2]{sum(0,4)=10,sumo(0,4)=10} |