# Operating Systems - A3

## A3. Simulate a Cache (20%)

Your task is to simulate a hardware cache with **k-way associativity** and **LRU replacement strategy**. To understand the task description given below, you must be familiar with the lecture on hardware-caches.

## Input

The first line describes the cache. Each of the following lines describes a memory access.

The cache is described by four integers:

- **W** is the number of bits in one word. This number will be a multiple of 8.
- **C** is the number of data bytes in the cache. This number will be a power of 2. (In **C**, we do not count the space needed to store tags, just the space needed to store data.)
- **B** is the number of bytes in one cache block. This number will be a divisor of **C**.
- **k** is the number of lines in a block. This number will be a divisor of **B**.

All **W**, **C**, **B** and **k** will be positive integers. The number **W** of bits in a word will be ≤1024≤1024. The size **C** of the cache will not exceed the number of available addresses: C≤2W≤2. The number **kC**/**B** of lines in the cache will be ≤220≤220.

Each memory access is described by one address, which is a **W**-bit integer, in the interval [0,2W)[0,2). Such an address is the index in memory of the *byte* being accessed. The number of memory accesses will be ≤220≤220.

## Output

For each memory access in the input, print 'C' if it is served by the cache, and print 'M' if it is served by the memory. Whitespace (space and newlines) will be ignored, so use it as you see fit.

**Assume that the cache is initially empty.**

## Example 1

**Input**

```
32 1024 64 4
409
658
915
1172
661
1429
1168
403
925
1172
```

**Output**

```
MMMM
CMCM
MC
```

**Explanation.** On this machine, each word and each address has **W**=32 bits. The cache contains **C**=1024 bytes. Since each block contains **B**=64 bytes, there are **C**/**B**=1024/64=16 blocks. Since 16=2416=24, we can identify blocks using a 4-bit index. Since each block contains **k**=4 lines, there are **B**/**k**=64/4=16 bytes in a line; thus, to identify a byte in a line we need a 4-bit offset. Now let us write the addresses in binary, coloring the tag, the index, and the offset by blue, yellow and red, respectively. (Groups with different colors are also separated by underscores.)

```
00000000000000000000001_1001_10012 = 40910000000000000000000000010_1001_00102 = 65810000000000000000000000011_1001_00112 = 9151000000000
0000000000000100_1001_01002 = 11721000000000000000000000010_1001_01012 = 66110000000000000000000000101_1001_01012 = 14291000000000000000
0000000100_1001_00002 = 11681000000000000000000000001_1001_00112 = 40310000000000000000000000011_1001_11012 = 92510000000000000000000000
100_1001_01002 = 117210
```

In general, replacement in is handled independently for distinct blocks, but it so happens that in this example all addresses correspond to block 1001. The offset is irrelevant for our problem, because cache lines are loaded/evicted in their entirety; so, we ignore the offsets. Using the LRU strategy for tags, we get the following operation sequence, in the notation from the slides:

[]→1F[1]→2F[1,2]→3F[1,2,3]→4F[1,2,3,4]→2[1,3,4,2]→5F[3,4,2,5]→4[3,2,5,4]→1F[2,5,4,1]→3F[5,4,1,3]→4[5,1,3,4]
[]→1�[1]→2�[1,2]→3�[1,2,3]→4�[1,2,3,4]→2[1,3,4,2]→5�[3,4,2,5]→4[3,2,5,4]→1�[2,5,4,1]→3�[5,4,1,3]→4[5,1,3,4]

Note that each block contains 4 lines, which is why in the simulation above we never go above size 4. Each fault means that we access memory (M), while the absence of a fault means that the cache (C) is sufficient. We get the string MMMMCMCMMC. In the output you submit, you may split this string on different lines.

## Example 2

This example is for a directly mapped cache.

**Input**

```
8 2 1 1
0
1
0
1
0
1
2
1
2
1
```

**Output**

```
MM
CCCC
M
CCC
```

**Explanation.** Each address has 8 bits, so there are 28=25628=256 possible addresses. The cache contains 2 data bytes, each belonging to a different block. Since there is one line per block, that means that each line contains one byte, so the offset in the line has 0 bits. Since there are two blocks, the index of the block is 1 bit long. In other words, even addresses are handled by one byte of the cache, and odd addresses are handled by the other byte of the cache. We handle the two blocks independently.

```
0M
   1M
0C
   1C
0C
   1C
2M
   1C
2C
   1C
```

Interleaving the indicators M/C, we get the string MMCCCCMCCC, which is the desired output.

## Example 3

This example is for a fully associative cache.

**Input**

```
8 4 4 4
1
2
3
4
1
2
5
1
2
3
4
5
```

**Output**

```
MMMMCCMCCMMM
```

**Explanation.** Addresses have 8 bits. The cache contains 4 bytes, all in one block made of 4 lines; thus, each line contains 1 byte. Because there is 1 byte per line, the offset has log21=0log21=0 bits. Because there is 1 block per cache, the index has log21=0log21=0 bits. Thus, addresses are tags themselves, and we simply simulate LRU as in the slides:

[]→1F[1]→2F[1,2]→3F[1,2,3]→4F[1,2,3,4]→1[2,3,4,1]→2[3,4,1,2]→5F[4,1,2,5]→1[4,2,5,1]→2[4,5,1,2]→3F[5,1,2,3]→4F[1,2,3,4]→5F

[]→1�[1]→2�[1,2]→3�[1,2,3]→4�[1,2,3,4]→1[2,3,4,1]→2[3,4,1,2]→5�[4,1,2,5]→1[4,2,5,1]→2[4,5,1,2]→3�[5,1,2,3]→4�[1,2,3,

So, the answer is MMMMCCMCCMMM.

## Submission

You will receive 20 files, each of them with one input as described above. You have to produce the 20 corresponding outputs, and upload them to a webserver, alongside an explanation of how you produced those outputs. The explanation may be a short English paragraph, or it may be a program *you* wrote. You will find out immediately if your output is correct or not. You can resubmit as many times as you want before the deadline.

**Note that the 20 inputs you receive are randomly generated from your Kent username. So, different students will have different memory accesses to simulate.**