1. Cache address layout
    a. Going to use slightly different terminology than the book
        i. Same idea, but I find the book's variable names less than intuitive
    b. Terminology
        i. *C* = cache size
            1. Almost always given in terms of KB or MB
            2. KB = 2^{10}, MB = 2^{20}, GB = 2^{30}
                a. Will probably want to memorize these powers of 2
                b. Will be using powers of 2 a lot for these problems
        ii. *LS* = line size
            1. Size of the lines of cache we discussed earlier
            2. Typical line size ranges from 4 to 64 bytes, but could be larger
        iii. *LIC* = lines in cache
            1. The total number of lines we have
            2. LIC = C / LS
        iv. *S* = sets
            1. Cache lines divided into groups called sets
            2. Exactly how many sets depends on the cache mapping
                a. Will discuss this further in a bit
        v. *W* = ways
            1. The number of places that a line of cache could potentially go in each set
            2. Will discuss this further in a bit
            3. S * W = LIC
    c. Address layout (in bits)

Tag	**Set**	**Offset**		**Address Bits**
Address bits – set bits – offset bits	$\log_2 S$	$\log_2 LS$	=	Usually given

        i. Split up an entire physical address according to the above
        ii. Set and offset bits determine where exactly the cache line goes
        iii. Tag uniquely identifies memory addresses that map to same set
            1. Tag bits are always "whatever's left"
    d. Problems of this type
        i. Usually given some of the variables above
        ii. Asked to calculate the rest

2. Cache mapping types
    a. How does an address from main memory map to the cache?
    b. Direct mapped (DM)
        i. Each block of main memory gets directly mapped to a single cache line
        ii. W = 1, S = LIC
            1. Only one line in each set
            2. Number of sets is equal to number of lines
                a. S = LIC / 1 = LIC
        iii. Easy and cheap to implement
        iv. What happens if different memory accesses map to same line of cache?
            1. Line of cache keeps getting evicted for the other one
            2. Performance penalty since we must keep going to main memory for line we just evicted

v. Example
   1. 8-byte DM cache with line size of 2, and 4-bit address
   2. LIC = C / LS = 8 / 2 = 4
   3. For a DM cache, S = LIC = 4
   4. Address format below

Tag	Set	Offset		Address Bits
4 – 2 – 1 = 1 bit	$\log_2 S = \log_2 4 = 2$ bits	$\log_2 LS = \log_2 2 = 1$ bit	=	4 bits

3. Cache mapping example
   a. Use the same address format we calculated in the previous problem
   b. Memory values for addresses below

Address	Data
0110	0x1B
0111	0x59
1000	0xFE
1001	0x3D
1110	0x0C
1111	0x3A
1010	0x25
1011	0x98

   c. How do we map the addresses above to the cache?
      i. Use the address layout above to split the addresses into each portion
      ii. Use those portions below to fill in the cache in the next example

Memory Access	Tag	Set	Offset	Access Type	Hit or Miss
0110	0	11	0	Read	Miss
0111	0	11	1	Read	Hit
1000	1	00	0	Read	Miss
1001	1	00	1	Read	Hit
1110	1	11	0	Read	Miss
1111	1	11	1	Read	Hit
1010	1	01	0	Read	Miss
1011	1	01	1	Read	Hit

4. Fill in the cache example
    a. Use the memory access table from the previous problem to fill in the cache below
        i. Make accesses in same order as table above
    b. First two accesses
        i. 0110 reads from the cache
            1. Line isn't in cache, so we have a miss
            2. We grab the line from memory
                a. When making accesses to the cache, need to pull entire line
                b. We also grab 0111 from memory, because line size is two
            3. Read from cache, CPU pulls value 0x1B from the cache
        ii. 0111 reads from the cache
            1. Compare tags, tag for 0110 matches tag currently in set 11
            2. Since tags match, CPU pulls value 0x59 from the cache

Set Number	Line Number	Tag	Byte 0	Byte 1
00	0			
01	1			
10	2			
11	3	0	1B	59

    c. Next two accesses
        i. Same idea as previous two
        ii. 1000 reads from the cache
            1. Line isn't in cache, so we have a miss
            2. We grab the line from memory
                a. We also grab 1001 from memory
            3. Read from cache, CPU pulls value 0xFE from the cache
        iii. 1001 reads from the cache
            1. Compare tags, tag for 1001 matches tag currently in set 00
            2. Since tags match, CPU pulls value 0x3D from the cache

Set Number	Line Number	Tag	Byte 0	Byte 1
00	0	1	FE	3D
01	1			
10	2			
11	3	0	1B	59

d. Next two accesses
    i. 1110 reads from the cache
        1. There is a line in the cache, but the tags don't match!
        2. Need to evict the line currently in cache and place new line inside
        3. We grab the line from memory
            a. We also grab 1111 from memory
            b. Evict line currently in cache, replace with new line and new tag
        4. Read from cache, CPU pulls value 0x0C from the cache
    ii. 1111 reads from the cache
        1. Compare tags, tag for 1111 matches tag currently in set 11
        2. Since tags match, CPU pulls value 0x3A from the cache

Set Number	Line Number	Tag	Byte 0	Byte 1
00	0	1	FE	3D
01	1			
10	2			
11	3	~~0~~ 1	~~1B~~ 0C	~~59~~ 3A

e. Final two accesses
    i. 1010 reads from the cache
        1. Line isn't in cache, so we have a miss
        2. We grab the line from memory
            a. We also grab 1011 from memory
        3. Read from cache, CPU pulls value 0x25 from the cache
    ii. 1011 reads from the cache
        1. Compare tags, tag for 1011 matches tag currently in set 01
        2. Since tags match, CPU pulls value 0x98 from the cache

Set Number	Line Number	Tag	Byte 0	Byte 1
00	0	1	FE	3D
01	1	1	25	98
10	2			
11	3	~~0~~ 1	~~1B~~ 0C	~~59~~ 3A