Question 1)
In all instances of the offset code, an array with an effective size of 8000 bytes is made. The difference with each offset value is that the program pushes the start of the array forward by “offset” amount of bytes. When the value is 16, the array pointer is pointing to an aligned location in memory. When this value is 17, the array ptr needs to realign itself in memory by pushing another 7 bytes along (potentially 15), every time the variable is called and storing/ accessing pointer data.

Question 2)
All array accesses are (s % SIZE) * 512 or a multiple of 512 appart. In a directly mapped cache, with 8 bytes for indexing (4K page table), the cache uses the last bits of the address as an index into the table. In this case, it would be very inefficient to have a cache with that can store 512 addresses/ data pairings but only use the first one as k*512 mod 512 is 0 (the first index of the page table).

Question 3)
If s==0, then the xor operation is 0 xor 0 = 0 or 0+0=0 on repeat. As a result, the array index of s % SIZE is also zero, meaning there’s no point in doing the test, as the program isn't testing memory access speed at different locations. Due to this, there’s no need to fetch the execution time of end-start.

Question 4)
Using the tail of my linked list (which represents the heap), I go back to the start of my allocated memory, in search for the “freed” address.

Double free: if the address is not found in there,, I check my unallocated addresses using my “free pointers” linked list. If the address is in there, I know that this is a “double free”.

Invalid free: if the address is in neither the allocated nor unallocated linked lists, I know that the address has never been called from malloc, hence an invalide free was called.

Use after free: On freeing an address, I write all the memory over with a placeholder value, 0xb16b00b5. As this data should not be used unless it is called by malloc again, the data does not need to store anything. This can then be double checked for write overs by comparing the data section and the canaries with the placeholder value. This does not work for memory that is freed using munmap, as the operating system has that responsibility, however the program should abort either by a segmentation fault from the munmap usage, or my program’s checker.

Question 5)
For 8 byte underflow, i saved the last 8 bytes of my 32 byte header as a canary. For overflow, I made another 8 byte canary at the end of the requested data segment. These are checked at the requested times to see whether they’ve been written over.

These are limited to only 8 bytes on either side, and if the attacker guesses the canary, then the canaries are worthless. Additionally, if the attacker changes the size of the allocated data and changes the tail pointer of the malloc linked list, the attacker could have control of all of the heap, like the dynamic memory is one big malloc chunk.