CSE 351 Winter 2018 Welcome back to section ☺
Section 10 Worksheet: Memory Allocation
To understand the computer, you must become the computer.

For this problem, you will use a heap simulator to answer questions about heap memory allocation in C using `malloc` and `free`. First, access the heap simulator, either by going to the Lab 5 page on the course website, or by following the link:

`https://sarangjo.github.io/cse351-heap/`

Read through the information on the page so you can understand how the simulator works. If you check "simulation mode" it will walk you through each step in the process of allocating a block of memory on the heap. Notice that each header (and footer) contains three fields. They correspond to:

```
block size :  previous block allocated?  :  block allocated?
```

where `1` means "allocated" and `0` means "unallocated." Remember that block size **includes** the size of the headers and footers. In Lab 5, all of this will be stored within 8 bytes using by reserving the last few bits of the size (see the spec for more information).

Now, answer the following questions:

1. Try allocating a 23 byte block with "simulation mode" enabled.
   (a) How much total space is allocated for the block, including for metadata?
   (b) What is the header for the newly allocated block?
   (c) What is the header for the block immediately following the allocated block?
   (d) Does the newly allocated block have a footer tag? If not, why doesn't it need one?
   (e) What pointer is returned to the client?

2. Now, armed with your newfound knowledge about this mysterious memory allocator, figure out some of its properties. You may add and/or remove whichever blocks you want to help answer these questions.
   (a) What is the **alignment** for this memory allocator?
   (b) What is the minimum block size? Why? What does the block size include?

3. Starting with an empty heap (you can empty the heap by refreshing the page), "execute" the following code:

```
malloc(30)
malloc(40)
malloc(70)
```

   (a) What pointer is returned if we execute another `malloc` now?

   (b) Which block(s) could you free that would cause fragmentation in the heap?

   (c) Which block(s) could you free that would cause coalescing to occur?

   (d) Suppose `free(48)` is run immediately after `malloc(70)`. Draw a diagram of what the free list looks like afterwards.

   (e) What is the maximum sized **payload** that we can allocate (i.e. the argument to `malloc`) such that we are returned a pointer to the address `48`?

4. What are the minimum and maximum arguments to `malloc` such that we must allocate a block of size N bytes? The first one has been filled out for you.

N (bytes)	Minimum Argument	Maximum Argument
32	1	24
48		
72		
88		
128		

5. In Lab 5, we will implement a memory management system that uses an explicit free list. Each block has pointers to the next and previous blocks. This is the block struct we will use:

```
        struct BlockInfo {
            // Size of the block (in the high bits) and tags for whether the
        // block and its predecessor in memory are in use.
        size_t sizeAndTags;

        struct BlockInfo* next;
        struct BlockInfo* prev;
    };
        typedef struct BlockInfo BlockInfo;
```

(a) Implement the following method. Try using bitwise operators to access the tags in sizeAndTags.

```
        // Bit masks used to retrieve tags from sizeAndTags.
        #define TAG_USED 1
        #define TAG_PRECEDING_USED 2
        // SIZE(blockInfo->sizeAndTags) extracts the size of a 'sizeAn

        // Copies the tags (TAG_PRECEDING_USED and TAG_USED) from bloc
        // to originalBlock. Leaves the size of originalBlock unchange
        void copyTags(BlockInfo* originalBlock, BlockInfo* blockToCopy

            size_t copyUsed = ____________________________;

        size_t copyPrecedingUsed = ____________________________;

        originalBlock->sizeAndTags = ______________________________
    }
```

(b) Now, some practice with doubly-linked lists. Implement the following method.

```
        BlockInfo *FREE_LIST_HEAD;

        // Removes a block from the free list.
        void removeFreeBlock(BlockInfo* freeBlock) {
            BlockInfo *nextFree, *prevFree;

        nextFree = freeBlock->next;
        prevFree = freeBlock->prev;

        // Your implementation goes here...










    }
```

Question F9: Memory Allocation [9 pts]

(A) In a free list, what is a **footer** used for? Be specific. Why did we not need to use one in allocated blocks in Lab 5? [2 pt]

Footer:
Lab 5:

(B) We are designing a dynamic memory allocator for a **64-bit computer** with **4-byte boundary tags** and **alignment size of 4 bytes**. Assume a footer is always used. Answer the following questions: [4 pt]

Maximum tags we can fit into the header (ignoring size): ________ tags

Minimum block size if we implement an *explicit* list: ________ bytes

Maximum block size (leave as expression in powers of 2): ____________ bytes

(C) Consider the C code shown here. Assume that the `malloc` call succeeds and `foo` is stored in memory (not just in a register). Fill in the following blanks with ">" or "<" to compare the *values* returned by the following expressions just before `return 0`. [3 pt]

```
#include <stdlib.h>
int ZERO = 0;
char* str = "cse351";

int main(int argc, char *argv[]) {
    int *foo = malloc(8);
    free(foo);
    return 0;
}
```

```
&foo   _____   &ZERO

&str   _____   ZERO

&main  _____   str
```