Solutions to practice final problems

**Floats**
Calls a and b compute the correct result. Call c returns 0 (not 2), loss of precision in first addition. Call d returns infinity due to overflow.

All powers of two have all 0 bits for the entire significand.

\[
\begin{array}{c}
0 \\
01111101 00000000000000000000000000000000
\end{array}
\]

Floating point allows greater range and precision is relative to magnitude (instead of absolute for all values).

**Assembly**

```assembly
int binky(char *param1, long param2) {
    int arr[4] = {16, 0, param2, *param1};
    void *ptr = &arr[2];
    while (ptr != 0) {
        arr[0] *= strlen("CS107"); // or any 5-character string
        ptr = binky(arr[0], arr + *(long *)ptr);
    }
    return 0;
}
```

Given the initialization of ptr, the compiler can determine the loop test will be true on first iteration and loop always entered, thus the first jump to the loop test could be removed.

The call to `strlen` on a constant string was replaced by its known length. The optimized multiply by 5 was expressed as

```assembly
lea (%rdx,%rdx,4),%edi
mov $0x0,%eax
```

**Runtime stack**

As if by magic, the function returns `amount`. This is the value in `%rax` when the function returns.

When asked how much to withdraw, enter -1000000 and voila, you're a millionaire!

`(unsigned int)amount >= max` forces an unsigned comparison. Negative numbers are no longer accepted.

When asked how much to withdraw, enter 0, followed by 15 pad chars (any non-digit non-newline char will do), then a 4-byte integer written in little-endian order of the value you would like your balance to be. When `gets` reads this string into `buf`, the last 4 bytes overwrite the saved `%rbx` register, effectively changing the value of the `balance` local variable in `main`. Very sneaky!
Heap allocator
Constant folding. If an expression only involves constants, the compiler evaluates it at compile-time and will use the immediate value in place of generating the instructions to compute it again.

These three work:

\[
(*\text{hdr} \& \text{MSB\_MASK}) == \text{MSB\_MASK} \\
(*\text{hdr} \& \text{MSB\_MASK}) \\
*(\text{signed char} *)\text{hdr} < 0
\]

```c
size_t get_blocksize(Header *hdr) {
    return 1L << (*hdr & SIZE_MASk);
}

Header *get_neighbor(Header *hdr) {
    Header *next = hdr + get_blocksize(hdr);
    if (next >= heap_start + heap_size) return NULL;
    return next;
}
```

memcpy(first + 1, &second, sizeof(void *));
*(Header **)(first + 1) = second;

```c
Header *next = get_next(cur);
if (prev == NULL) 
    free_list = next;
else  
    set_next(prev, next);
```

Block sizes have to be exact power of 2.

```c
void my_free(void *ptr) {
    Header *hdr = (Header *)ptr - 1;
    *hdr |= MSB_Mask;
    set_next(hdr, free_list);
    free_list = hdr;

    Header *next = get_neighbor(hdr);
    if (next && *hdr == *next) { // compares size and freed
        remove_from_freelist(next);
        *hdr += 1;
    }
}
```

Double-link:
Code changes: Must splice backward link in addition to forward when adding/removing from list
Throughput: mymalloc unchanged, myfree was O(N) now O(1)
Utilization: min block size increases to 32 (min payload >= 16 bytes for two pointers)

Segregate:
Code changes: segregate by powers of two, use log size from header bits as index into array of linked lists, same code to add/remove from list
Throughput: mymalloc was O(N) now O(1) (takes first block), myfree was O(N) now O(N/B) (number of buckets ~32)
Utilization: mymalloc now operating as best-fit, may result in less fragmentation