This document contains the questions and solutions to the CS107 Final given in Winter 2018 by instructor Chris Gregg. This was a 3-hour exam.

Problem 1 (Void * generics) (24 points) (suggested time: 30 minutes)

(a)

```
int remove_duplicates(void *arr, size_t nelems,
                      int width, int (*cmp)(void *, void *)) {
    int i = 0;
    while (i < nelems - 1) {
        void *ith = (char *)arr + i * width;
        void *ithplus1 = (char *)arr + (i+1) * width;
        if (cmp(ith,ithplus1) == 0) {
            // remove
            memmove(ithplus1,(char *)ithplus1+width,
                    (nelems-i-2)*width);
            nelems--;
        } else {
            i++;
        }
    }
    return nelems;
}
```

(b)

```
int cmp_long(void *p, void *q) {
    if (*(long *)p > *(long *)q) return 1;
    else if (*(long *)p < *(long *)q) return -1;
    return 0;
}
```

(c)

```
int newsz = remove_duplicates(arr, nelems,
                              sizeof(long), cmp_long);
```

Problem 2 (Floats) (17 points) (suggested time: 20 minutes)

(a) Instead of storing one bit for each binary digits place, as two's complement does, IEEE floating point stores numbers in the form x * 2^y, and stores x and y in binary representations. The benefit of this is that floats can store a much wider range of numbers because two's complement can only store 64 binary digits. The drawback of this is that not all numbers are representable, because some numbers cannot be represented in this scientific notation form.

Other possible mentions: float arithmetic more difficult, dedicated float sign bit means 2 zeros, float encodes infinity and other exceptional values, but increases complexity in comparison.

(b) 0.75

(c) A float is normalized if the exponent is not all 1s and not all 0s. A float is denormalized if the exponent is all 0s. A float is exceptional if the exponent is all 1s.

(d)

```
true
true
false
true
false
```

Problem 3 (Assembly) (24 points) (suggested time: 35 minutes)

(a)

```
void mystery(long *arr, size_t count) {

    if (count > 0) {  // line 1

        mystery(arr, count / 2); // line 2

        printf("%lu\n", arr[count-1]); // line 3

    }
}
```

(b)

```
void *mystery(void *arr, size_t nelems, int width,
              int(*cmp)(void *, void *)) {

    void *x = arr; // line 1

    for (size_t i = 1; i < nelems; i++) { // line 2

        void *y = (char *)arr + i * width; // line 3

        if (cmp(x, y) < 0) { // line 4

            x = y; // line 5

        }
    }

    return x; // line 6
}
```

Problem 4 (Runtime Stack) (20 points) (suggested time: 30 minutes)

(a) Run the program and type a password that is at least 16 characters long. You will see a message that says, “your password, xxxxxxxxxxxxxxxxxxx, is incorrect”, however it will list your password AND the real password. Run the program again and type the real password, and you will break in!

(b) The

```
strncpy(userpwcopy,userpw,16);
```

line will not null-terminate the copy if it is 16 or more characters long. Therefore, the `userpwcopy` variable will not be null-terminated, and the

```
printf("Your password, %s, is incorrect.\n",userpwcopy);
```

line will continue printing the `realpw` variable, in effect concatenating the two variables. This will print out the real password to the screen, which can be used to run the program again and gain access.

(c) There are a number of different answers. One would be:

After the `strncpy(userpwcopy,userpw,16);` line, add: `userpwcopy[15] = 0;`

There are other alternatives — one is to not bother using a copy of the user’s password and simply to print out the original:

```
printf("Your password, %s, is incorrect.\n",userpw);
```

Problem 5 (Heap Allocator) (36 points) (suggested time: 50 minutes)

(a)

```
int get_size(void *curr) {
    int mask = -1 << 2;
    return (*((int*)curr)) & mask;
}
```

(b)

```
bool is_allocated(void *curr) {
    int mask = 0x1;
    return (*((int*)curr)) & mask;
}
```

(c)

```
bool is_reallocated(void *curr) {
    int mask = 0x2;
    return (*((int*)curr)) & mask;
}
```

(d)

```
headerT *right_block(headerT *curr) {
    headerT *next = (headerT *)(((char*)curr + get_size(curr))
                    + sizeof(headerT) + sizeof(int));
    if ((char *)next >= (char*)heapStart + heapSize) {
        return NULL;
    }
    return next;
}
```

(e)

```
void myfree(void *ptr) {
    if (ptr == NULL) return;

    headerT *header = (headerT *)ptr - 1;

    // past end or before beginning — include offset padding
    if ((char *)header >= ((char*)heapStart + heapSize)) return;
    if ((char *)header < ((char*)heapStart + 4)) return;

    if (!is_allocated(header)) return;
    int mask = -1 << 2;
    header->payloadsz &= mask; // clear out alloc/realloc bits

    // clear footer
    int *footer = (int *)((char *)header + get_size(header) +
                  sizeof(headerT));
    *footer = header->payloadsz;

    // update free list
    header->next = free_list;
    header->prev = NULL;
    if (free_list != NULL) free_list->prev = header;
    free_list = header;
}
```

(f)

```
void *myrealloc(void *ptr, size_t size) {
    headerT *header = (headerT *)ptr - 1;

    int cursz = get_size(header);
    if (size <= cursz) return ptr;
    if (is_reallocated(header)) size *= 2;

    void *block = mymalloc(size);
    if (!block) return NULL;
    memcpy(block, ptr, cursz);
    myfree(ptr);

    int mask = 0x2;

    // rewind to header, mark realloc
    header = (headerT *)block - 1;
    header->payloadsz |= mask;

    // mark footer realloc and make new header
    int *footer = (int *)((char *)header + sizeof(headerT)
                  + get_size(header));
    *footer |= mask;

    return block;
}
```

Problem 6 (gcc / make) (9 points) (suggested time: 10 minutes)

(a) Constant folding is one optimization where GCC will embed constant values directly into the assembly instructions, instead of outputting assembly instructions to calculate it when the program is run.

Other possibilities: constant sub-expression elimination (calculates repeated expression once in assembly instructions and saves the result to avoid recomputation), strength reduction (changes more expensive instructions such as multiply and divide to less expensive operations such as shifts, mod, etc.), more GCC optimizations besides this!

(b) Optimizations can cause the assembly to not map well to the C code anymore, because many C lines may be converted to few instructions, variables may be optimized out, etc.

(c) This is an example of constant folding – GCC can figure out the value of the strlen and sizeof expressions at compile time in this code. Sizeof is always a compile-time operation, so it will never appear in assembly instructions. The strlen expression here is on a string constant, which means its length is known at compile-time. For this reason embeds the values directly in the assembly instructions instead of making a call to strlen.