The CS107 midterms are graded, and you can check out how you did by visiting Gradescope. I thought the exam was challenging and focused on teasing out CS107-specific knowledge, and the majority of you pretty much nailed it. The median grade was a 36 out of 45, and the standard deviation was just a smidge below 8.0. Here’s a graph of how everyone did:

![Graph of CS107 Midterm Statistics and Solution](image)

Each dot represents a single exam score, and scores ranged from 2 to 45. If you didn't do as well as you hoped to, you’re more than welcome to come chat during my office hours or set up a time to speak with me outside my scheduled office hours set. And even though we expect the final exam to be equally challenging, understand that the two exams test very different things, so it’s quite possible to salvage a poor midterm score and do well on the final.

The rest of this handout includes my own solutions. We understand the exams count for a large portion of your grade, so we try to be as transparent as possible about the criteria, which is openly visible on Gradescope. If you have a legitimate concern about how your midterm was graded because something was clearly correct when it was marked incorrect, you may submit a regrade request by next Friday, just before we depart for Thanksgiving break. Please understand, however, that we can’t grade your midterm differently than others simply because you wrote one thing when you meant something else. We have to grade what was written down.
Solution 1: Reconstructing Codepoints from UTF8 Encodings

unsigned short decode(unsigned char upper, unsigned char lower) {
    if ((0xe0 & upper) != 0xc0) error(1, 0, "Upper byte of encoding is malformed.");
    if ((lower >> 6) != 0x2) error(1, 0, "Lower byte of encoding is malformed.");
    unsigned short codepoint = 0;
    codepoint |= 0x3F & lower;
    codepoint |= (0x1F & upper) << 6;
    return codepoint;
}

We were equally happy with binary or even decimal masks (though in practice you should use hexadecimal). There were also other ways to reassemble the codepoint from the relevant bits of upper and lower, but the approach above is more or less what’s really done with UTF8-to-codepoint conversion.

Solution 2: strseparate

char *strseparate(char **p_string, const char *delimiters) {
    char *front = *p_string;
    char *end = front + strcspn(front, delimiters);
    *p_string = end + (*end == '\0' ? 0 : 1);
    *end = '\0';
    return front;
}

There was a little bit of wiggle room on how this could be implemented. The real challenge was knowing when it was okay to rely on front and end and when it was necessary to trampoline through p_string.

Problem 3: Arrays of Suffixes

typedef struct entry {
    char **suffixes;
    size_t count;
} entry;

entry *broadcast(char *strings[], size_t length) {
    entry *entries = malloc(26 * sizeof(entry));
    for (size_t i = 0; i < 26; i++) {
        entries[i].suffixes = NULL;
        entries[i].count = 0;
    }
    for (size_t i = 0; i < length; i++) {
        size_t bucket = strings[i][0] - 'a';
        size_t count = ++entries[bucket].count; // count is incremented value
        entries[bucket].suffixes = realloc(entries[bucket].suffixes, count * sizeof(char *));
        entries[bucket].suffixes[count - 1] = strdup(strings[i] + 1);
Two fun facts about the implementation:

- The first for loop—the one that zeroes everything out—could have been managed by a call to `memset(entries, 0, 26 * sizeof(entry))`, or equivalently `bzero(entries, 26 * sizeof(entry))`. Zeroing out large regions of memory is common enough in systems programming that there are dedicated library functions to do precisely that.

- `realloc(NULL, size)` is implemented to simply call and return `malloc(size)`.

Solution 4: Finding adjacent matches

```c
size_t find_adjacent(void *base, size_t nelems, size_t elem_size, void *result, int (*cmp_fn)(void *, void *)) {
    for (size_t i = 0; i < nelems - 1; i++) {
        void *first = (char *) base + i * elem_size;
        void *second = (char *) first + elem_size;
        if (cmp_fn(first, second) == 0) {
            memcpy(result, first, 2 * elem_size);
            return i;
        }
    }
    return nelems; // adjacent pair wasn’t found, return nelems
}

int last_char_cmp(void *ptr1, void *ptr2) {
    char *s1 = *(char **) ptr1;
    char *s2 = *(char **) ptr2;
    return strlen(s1) - 1 - s2[strlen(s2) - 1];
}
```