Above is the histogram and class statistics for the midterm. We saw lots of high-quality answers all around, even some perfect and near-perfect total scores, a rarity on an exam with so many details to manage. ¡Felicidades!

CS107 is packed with challenging material and we ask a lot of you. No one escapes unscathed— the autograder runs roughshod over your code and we mark up your work with every issue we can find. Our red ink reflects the many subtleties of systems and that there is always room for improvement, but we also want to assure you that your achievements in mastering the core content are being recognized and we will have that in mind when setting course grades. The median course grade for CS107 generally lands in the neighborhood of B+/B.

The midterm serves as a mid-quarter assessment. Bravo to those who achieved the result you worked toward! If your score shows you don’t yet have a solid grip on the material or were not able to successfully demonstrate your ability on the exam, now is the time to figure out what changes to make going forward. I offer advice that might be of help in that regard at [http://cs107.stanford.edu/advice/midquarter.html](http://cs107.stanford.edu/advice/midquarter.html). Please reach out to us in office hours if you need further consultation on plotting your strategy.

Your graded exam is available online at Gradescope. You should receive an email at your sunet@stanford.edu address with information on how to access it.

**Grading rubrics**

Our rubrics are designed to apply constructively. Your score starts at zero and we identify in your answer those tasks for which you make a valid attempt and/or accomplish correctly and attach the appropriate rubric items, each of which makes a positive contribution to the score. For tasks that were not attempted or not accomplished correctly, there will be no rubric item applied and there is no change in score for those items.

In the coding questions, care in handling pointers/memory, applying the correct typecast, computing the right offset, and accessing at the appropriate level of indirection were the key skills we were assessing. The bulk of the points are apportioned for those issues, with correspondingly lighter weighting on the mundane tasks (for loops, return values, integer counters, etc.).
Regrades: If you believe there was an error in scoring your exam, please send email to cs107@cs with an explanation of the concern and ask for our review. The entire exam will be re-evaluated. Regrade requests must be initiated by Mon Nov 12th.

Problem 1: Bits, bytes, and numbers
Consider the mystery function. The marked line does most of the work of the function.

```c
int mystery(unsigned int v)
{
    int c;
    for (c = 0; v; c++)
    {
        v &= v - 1; // Line 5
    }
    return c;
}
```

1a) Identify the change in bit pattern between a non-zero unsigned value number and its numeric predecessor (number - 1).

*The least significant 1 bit is now a 0 and any bits further to right are all 1s.*

1b) How does the bit pattern of v change after executing line 5?

*The least significant 1 bit is changed to a 0.*

1c) In terms of the bit pattern for v, what value is returned by the call mystery(v)?

*The count of 1 bits in v.*

1d) The following statements appear in a C program running on our myth computers.

```c
int x = /* initialization here */
bool result = (x > 0) || (x-1 < 0);
```

Either argue that result is true for all values of x or give a value of x for which result is false. Is result always true?

*No. If x is INT_MIN, result is false.*

Commentary from grading problem 1
We found that some students went into too much detail in their answers to parts (a) and (b) – it was not necessary to separate evens from odds, for instance. This led to a number of students either forgetting a “case,” or not sufficiently explaining what was actually happening. A number of students lost points on parts (a) and/or (b) but did correctly determine the answer to part (c). Students who incorrectly answered part (c) would probably have had success if they walked through the algorithm on a few more small examples. Students were largely successful on part (d).
Problem 2: C-strings

2a) The function strip_leading(char *input, const char *discard) removes the leading characters from the string input that occur in the string discard. Here are some examples:

```c
char *word = strdup("details");
strip_leading(word, "s") no change to word
strip_leading(word, "due") changes word to "tails"
strip_leading(word, word) changes word to ""
```

Requirements:

- Your function should not allocate, deallocate, or resize any memory. Instead it should destructively modify the input string. If the input string loses some characters, its memory will be over-allocated at the end; your implementation should leave the memory as-is.
- Re-implementing functionality from the standard library will result in loss of credit. *Hint:* your code should not have any explicit loops, instead call the library functions!
- The string-copying routines (e.g. strcpy/strcat) cannot be used when the source and destination overlap. *Hint:* remember what alternatives are in the standard library!

```c
void strip_leading(char *input, const char *discard)
{
    size_t n = strspn(input, discard);
    memmove(input, input+n, strlen(input) - n + 1);
}
```

2b) Your colleague adds this final line to strip_leading to fix the over-allocation issue:

```c
input = realloc(input, strlen(input)+1);
```

Not only is this call not likely to shrink the memory, it causes two distinct memory errors. What are they?

*There is no guarantee that the input string is heap-allocated, attempting to realloc non-heap memory has unpredictable results.*

*The input pointer is not passed by reference, so re-assigning does not have a persistent effect. The caller's original pointer is unchanged.*

Commentary from grading problem 2

We were happy to see how many of you turned to strspn to scan for the discard characters! From there, modifying the string must be done with memmove due to overlapping regions. Moving the pointer fails to have any effect at all as the pointer is passed by value. For 2b, we also accepted answers justifying that the sample realloc we studied will not move a pointer when shrinking size but it's worth noting this behavior is implementation-specific, not a guarantee of standard realloc.
Problem 3: Pointers and generics

3a) The generic find_min searches an array for its smallest element according to a client-supplied callback function. The function arguments are the array base address, the count of elements, the size of each element in bytes and a comparison function. The function returns a pointer to the minimum array element. As an example, find_min on the array \{3.7, 9.4, 1.1, -6.2\} with ordinary float comparison returns a pointer to the last element in the array. Fill in each of the three blank lines with the necessary expression so that the function works correctly.

```c
void *find_min(void *base, size_t nelems, size_t width,
               int (*cmp)(const void *, const void *))
{
    assert(nelems > 0);  // error if called on empty array
    void *min = base;    // Line 1
    for (size_t i = 1; i < nelems; i++) {
        void *ith = (char *)base + i*width;  // Line 2
        if ( cmp(ith, min) < 0 )            // Line 3
            min = ith;
    }
    return min;
}
```

3b) Complete the program started below to use find_min to find the command-line argument with the minimum first character ("minimum" means smallest ASCII value) and print that character. For example, if invoked as .\program red green blue, the program prints 'b'. You must fill in the blank line in main with a call to find_min and can assume it works correctly. You will also need to implement the comparison callback function. Hint: remember that the command-line arguments start at index 1 in the argv array.

```c
int cmp_first(const void *p, const void *q)
{
    return **(const char **)p - **(const char **)q;
}
```

```c
int main(int argc, char *argv[])
{
    char ch = **(char **)find_min(argv+1, argc-1, sizeof(*argv), cmp_first);
    printf("Min first char of my arguments is \%c\n", ch);
    return 0;
}
```

3c) The selection sort algorithm works by repeatedly selecting a minimum element and swapping it into position. On the first iteration, it finds the minimum array element and swaps it with the first element. The second iteration finds the minimum element of the subarray starting at the second position and swaps it into the second position. This process repeats on shorter and shorter subarrays until the entire array is sorted.
Implement the `selection_sort` function below to perform the selection sort algorithm on a generic array. You will need to call `find_min` and can assume it works correctly.

```c
void selection_sort(void *base, size_t nelems, size_t width,
                     int (*cmp)(const void *, const void *))
{
    for (size_t i = 0; i < nelems - 1; i++) {
        void *ith = (char *)base + i*width;
        void *min = find_min(ith, nelems - i, width, cmp);
        char tmp[width];
        memcpy(tmp, ith, width);
        memcpy(ith, min, width);
        memcpy(min, tmp, width);
    }
}
```

**Commentary from grading problem 3**

This was an homage to the time invested in lab/assign 3 and 4. Much of the grading for this question hinged on handling `void*` pointers at the appropriate level of indirection. For example, the pointers exchanged with `find_min` (args to callback, return value) are all expressed as pointers to elements. Right versus wrong may come down to the presence or absence of a single `*` or `&`, but don't dismiss these issues as minor. This was a crucial insight being tested and it was heavily weighted in assessment. Keeping your pointers straight is challenging; those of you were vigilant should feel well-rewarded for your efforts!