

CS107 Midterm Prep Exercises (SOLUTIONS ✓)

Ben Yan, Winter 2026
bbyan@stanford.edu

Curated from extra problems on our lab handouts, my tormented imagination, etc.

🚩 Problem 1: Zero to Hero (Bits and Bytes)

(a) Implement the function **zero_upper**, which accepts, as input, unsigned (32-bit) integers **value** and **N**. The function should use a bitwise expression to zero out the upper **N** bits of **value**, and return that result. You can assume **N** to be an integer between 1 and 31, inclusive.

For instance, if **value** had binary 11110000111100001111000011110000, and **N** were 8, then **zero_upper** should return a value with binary representation 00000000111100001111000011110000, with the upper or most significant 8 bits set to zero.

C/C++

```
// one possible solution here!
unsigned int zero_upper(unsigned int value, unsigned int N){
    return value & (-1U >> N);
}
```

(b) What does the bitmask expression you wrote above compute numerically about **value**?

None

The remainder when value is divided by $2^{(32 - N)}$.

The reason: polynomial division! Observe that bits are coefficients of a binary polynomial (e.g., 0101 is $0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$).

Note that **zero_upper()** will obliterate the higher bits corresponding to higher powers of 2, e.g., 2^{31} , 2^{30} , ..., $2^{(32 - N)}$. The remaining lower bits, i.e. $2^{(32 - N - 1)}$, ..., 2^0 , thus represent a remainder, and specifically the remainder at the cutoff bit with power $2^{(32 - N)}$.

As a concrete example, take $N = 31$. Zeroing out the upper 31 bits leaves only the LSB: 0 if value is even, and 1 if value is odd. This is identical to the remainder when value is divided by 2, or $2^{(32 - 31)}$.

📦 Problem 2: Mystery Box Function (Bits and Bytes)

Consider the function `mysteryBox` and its (mysterious, one could say!) implementation below.

```
int mysteryBox(int x, int y){  
    return (x & y) + ((x ^ y) >> 1);  
}
```

(a) What does the following code print?

```
printf("%d\n", mysteryBox(2, 10));    // %d is placeholder  
printf("%d\n", mysteryBox(10, 20));   // to print out an integer  
printf("%d\n", mysteryBox(20, 60));   // value, not its binary!
```

None

The code will print:

6

15

40

(b) (★ **Optional Fun Challenge**) In general, what mathematical relationship or property between any integers `x` and `y` does `mysteryBox(x,y)` compute? Why does it work?

None

mysteryBox computes the midpoint between the two integers x and y. To see how its bitwise expression gathers the needed powers of 2 of the midpoint, first recognize that the bits are coefficients of a binary polynomial (e.g., 0101 is $0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$).

Now consider that `x & y` picks out the non-zero coefficients (1-bits) which are shared, and `x ^ y` picks out the non-zero coefficients which are not shared between the two operands. The non-shared bits need to be divided by 2 (i.e., right-shifted), whereas the shared coefficients do not need to be divided by 2 because there are two copies of them in the sum. We add the bit groups together to produce the midpoint, allowing powers to carry over if needed. (from the Lab 1 extra problem solutions)

Trivia: This expression is immune to both positive/negative overflow!

Problem 3: Achilles Heel (Bits and Bytes)

Implement a function `achilles_heel`, which should accept an integer `position` as input, and return a (32-bit) integer whose binary representation is all 1 bits except for a solitary 0 bit at the given `position`, which is the offset from the leftmost bit / MSB.

For instance, if `position` is 4, then `achilles_heel` should return a value with binary `11110111111111111111111111111111`, with the solitary 0 bit (or the Achilles heel) located 4 positions right from the MSB. You can assume that `0 <= position <= 31`.

C/C++

```
int achilles_heel(int position){
    return ~(1 << (31 - position));
}
/* many alternatives! e.g., return -1 ^ (1 << (31 - position)); */
```

Problem 4: Power of Friendship (Bits and Bytes)

Implement a function `hasBitFriends`, which accepts an integer `num` as input, and returns `true` if the binary representation of `num` has a pair of consecutive 1 bits, and `false` otherwise.

C/C++

```
bool hasBitFriends(int num){
    return num & (num << 1) != 0;
}
/* alternatively, return num & ((unsigned int) num >> 1) != 0; */
```

/* Explanation:

If `num` has consecutive 1 bits in its binary, when we shift `num`'s bits to the left by one position, there should be overlapping 1 bits with the original. If there were no consecutive 1 bits, no overlap would exist.

The expression `num & (num << 1)` then evaluates to 0 when no 1 bits overlap between `num` and `num << 1`, and non-zero if we have some overlap.
*/

Problem 5: Life Comes Full Circle (Strings)

Implement the function `comesFullCircle`, which accepts a `char* str` and an integer `N` as input, and returns `true` if the string `str` has a matching prefix and suffix that is precisely `N` characters long, and `false` otherwise. That is, we return `true` only if the first `N` characters matches the last `N` characters, which are allowed to overlap. Some examples are shown below.

- `comesfullCircle("hunterXhunter", 3)` should return `false`.
- `comesfullCircle("hunterXhunter", 6)` should return `true`.
- `comesfullCircle("lalala", 2)` should return `true`.
- `comesfullCircle("lalala", 4)` should return `true`.
- `comesfullCircle("lalala", 7)` should return `false`.

Be sure to take advantage of the `string.h` library functions. You can assume `N >= 1`, but in the possible event that `N` exceeds the string's length, you should simply return `false`.

C/C++

```
bool comesFullCircle(char* str, int N){
    if (N > strlen(str)) return false;
    char* suffix = str + strlen(str) - N;
    // compares the first N chars (prefix) with last N chars (suffix)
    return strncmp(str, suffix, N) == 0;
}
```

🍷 Problem 6: String Theory, From All Matter to Strings (Strings, ★Challenge)

The `atoi` function converts a string of digits to a decimal, but no standard function exists that converts in reverse. Here, we'll implement the function `iota`, which will accept as input an unsigned long `num` and a `char[]` buffer. The function should convert `num` into a string of digits, and store the result in the passed-in buffer. You can assume that `buf` holds enough space for each digit character and null-terminator `\0`, and that `num` is a positive value.

For instance, calling `iota(107, buf)` should populate `buf` as the string of digits `"107"`. Note also that `'0' + digit` will get the character that corresponds to an integer digit.

C/C++

```
char* iota(unsigned long num, char[] buffer){
    // get how many digits in num
    int digit_count = 1;
    while ((num /= 10) > 0){ // note num gets divided by 10 here!
        digit_count++;
    }

    // iterates through digits from right to left, adding to buffer
    buffer[digit_count] = '\0';
    for (int i = digit_count - 1; i >= 0; i++){
        int curr_digit = num % 10; // get digit on right
        buffer[i] = '0' + curr_digit;
        num /= 10; // shave off current rightmost digit
    }
}

/* alternative solution */
char* iota(unsigned long num, char[] buffer){
    // first, add the digits to the buffer in reverse order
    int len = 0;
    while (num > 0){
        buffer[len++] = '0' + (num % 10); // get digit on right
        num /= 10; // shave off current rightmost digit
    }

    // then perform swaps to reverse the elements in the buffer
    for (int i = 0; i < (len - 1) / 2; i++){
        char temp = buffer[i];
        buffer[i] = buffer[len - 1 - i];
        buffer[len - 1 - i] = temp;
    }
    buffer[len] = '\0';
}
```

🔥 Problem 7: Domain Expansion, Malevolent Heap (Dynamic Memory, ★ Challenge)

Implement a function `expand`, which receives an array of `char* strings`, and an integer `strings_count` as input. The function should return a string which is the concatenation of all `strings_count` strings in `strings`. This returned string should be heap-allocated, and it's the client's responsibility, not that of `expand`'s implementation, to free this memory when done.

For instance, if `strings` consists of `"cleave"`, `"dismantle"`, `"divine"`, and `"flame"`, with `strings_count` equal to 4, then `expand` should return a heap-allocated string `"cleavedismantledivineflame"`. Don't forget space for the null-terminator!

C/C++

```
char* expand(char* strings[], int strings_count){
    char* result = strdup(strings[0]); // start with first string

    // loop through other strings, concatenating each with result
    for (int i = 1; i < strings_count; i++){
        size_t new_len = strlen(result) + strlen(strings[i]) + 1;
        result = realloc(result, new_len); // enlarge heap buffer
        strcat(result, strings[i]);
    }
    return result;
}

/* alternative that's less efficient, but doesn't need realloc() */
char* expand(char* strings[], int strings_count){
    // figure out length of eventual concatenated result
    size_t final_len = 0;
    for (int i = 0; i < strings_count; i++){
        final_len += strlen(strings[i]);
    }

    // create large enough heap buffer, and join together all strings
    char* result = malloc(final_len + 1); // include \0 char
    strings[0] = '\0'; // allows initial concatenation with strings[0]
    for (int i = 0; i < strings_count; i++){
        strcat(result, strings[i]);
    }
    return result;
}
```

Problem 8: Domain Expansion, Unlimited Void* (Dynamic Memory)

We implement a simplified `calloc`, which dynamically allocates an array of `nelems` elements each of size `elemsz` bytes, and returns a pointer to the base address of that array. The memory contents should be all initialized to 0, which is the shining feature that distinguishes `calloc` from ordinary `malloc`, though you are permitted to use `malloc` in your implementation. You don't have to perform error-checking on the inputs, and you may assume that the total size of the memory region, `nelems * elemsz`, is positive and a multiple of `sizeof(long)` or 8.

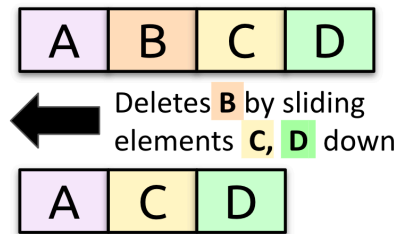
C/C++

```
void *calloc(size_t nelems, size_t elemsz){
    // first, allocate heap memory for the entire array
    size_t total_bytes = nelems * elemsz;
    void* array_ptr = malloc(total_bytes);

    // zero out the memory in 8-byte chunks, i.e. size of a long
    // could also do smaller byte chunks, but this is more efficient
    long* chunk_ptr = array_ptr;
    size_t num_chunks = 1 + (total_bytes - 1) / sizeof(long);
    for (int i = 0; i < num_chunks; i++){
        *chunk_ptr = 0;
        chunk_ptr++;
    }
    return array_ptr;
}
```

👉 Problem 9: Yeet That Element (Generics)

Implement a **generic function yeet** that, given an array with starting address **base** and a given integer **index**, deletes (or in the colorful language of my generation, “**yeets**”) the element at that **index**. Specifically, we should slide all the elements to the right of the **index** down, ensuring that the array base should stay in-place. The function is also supplied with the number of array elements passed in as an **int*** pointer, and the **width** / size in bytes of each element.



We can assume $0 \leq \text{index} < \text{nelems}$, and that the array is non-empty. Fill in the blanks.

C/C++

```
void yeet(void *base, int index, int *nelems_ptr, int width){
    void* elemToDelete = (char*) base + index * width;
    void* elemToRight = (char*) elemToDelete + width;

    // slide all the elements to the right of index down
    memmove(elemToDelete, elemToRight,
            ((*nelems_ptr) - 1 - index) * width);

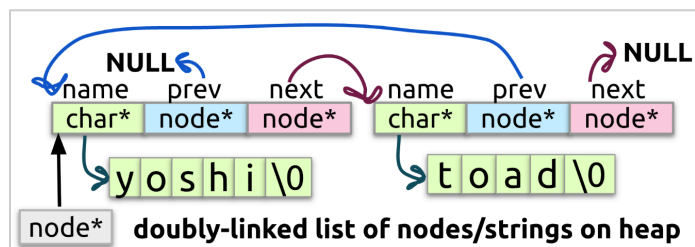
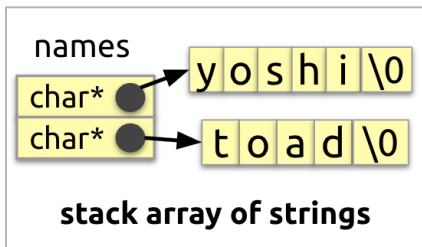
    // finally, update the number of elements
    (*nelem_ptr)--;
}
```

🧑‍🎓 Problem 10: Switch It Up Like Nintendo (Dynamic Memory, Structs, ★ Challenge)

Given a stack-allocated array **names** of valid C strings which represent character names (e.g., Yoshi, Kirby, Zelda, etc.), implement a function **switch_it_up_like_nintendo** that converts it into a dynamically-allocated, doubly-linked list of nodes of the following type:

```
typedef struct node {
    char* name; // pointer for heap-allocated name string
    node* prev; // pointer to previous node, NULL if none
    node* next; // pointer to next node, NULL if none
} node;
```

The list should preserve the order of the original array, and each node should be heap-allocated, along with each node's **name** string. The **prev** pointer of the starting node, and the **next** pointer of the last node, should be set to **NULL**. The function should return a pointer to the starting node. You can guarantee that the input array's length **num_names** ≥ 1 .



C/C++

```
void switch_it_up_like_nintendo(char* names[], int num_names){
    node* headNode = NULL; // pointer to starting node
    node* prevNode = NULL; // prev of first node is NULL
    for (int i = 0; i < num_names; i++){
        // heap-allocate each node, and its name string
        node* curNode = malloc(sizeof(node));
        curNode->name = strdup(names[i]);

        // wire up the current node to linked list
        if (headNode == NULL) headNode = curNode;
        if (prevNode != NULL) prevNode->next = curNode;
        curNode->prev = prevNode;
        prevNode = curNode;
    }
    prevNode->next = NULL; // next of last node is NULL
    return headNode;
}
```

Note: Arrow→ is for struct pointers. Use dot (.) if struct type itself, e.g., (***node_ptr**).name