IMPORTANT NOTES for all questions on the exam: For coding questions, the majority of the points are typically focused on the correctness of the code. However, there may be deductions for code that is roundabout/awkward/inefficient when more appropriate alternatives exist. For any coding questions, your answers should compile cleanly and not have any memory leaks or errors. You may need to scroll vertically or horizontally to fully view blocks of code. Solutions that violate any specified restrictions may get partial credit. For these problems, you do not need to worry about calling assert to check for heap errors. Style is secondary to correctness (e.g. there are no style deductions for using magic numbers).

1) Reconstructing Unicode Code Points

10 Points/45 Total

In Assignment 1, you implemented a function to transform Unicode codepoints to UTF8. In particular, all Unicode code points between the values of 0x080 and 0x07FF were encoded as two-byte figures with the following bit pattern:

```
110m npqr 10st wxyz
```

Assuming that the lowercase letters are some collections of zeroes and ones, this two-byte sequence might encode the binary codepoint 0000 0mnp qrst wxyz.

For this problem, you’ll implement a function called decode, which accepts the upper bytes (the one beginning with 110) and the lower byte of a two-byte UTF8 encoding and reconstructs the codepoint as a two-byte unsigned short.

Here is a template for the implementation:

```c
unsigned short decode(unsigned char upper, unsigned char lower) {
    if (___A___) error(1, 0, "Upper byte of encoding is malformed.
    if (___B___) error(1, 0, "Lower byte of encoding is malformed.
    unsigned short codepoint = 0;

    ___C___

    return codepoint;
}
```

Your job is to provide code to fill in the placeholders according to specification.
a. [3 points] _____A____ is a placeholder for the test that evaluates to true unless the three leading bits of the upper byte are 110. Construct a bitwise C expression that evaluates to true unless the three leading bits of upper are 110. Your test may use &, |, ==, !=, and hexadecimal, decimal, and/or binary constants. No other bitwise or relational operators are permitted for this part.
b. [2 points] _____B____ is a placeholder for the test that evaluates to true unless the two leading bits of the lower byte are 10. Construct a bitwise C expression that evaluates to true unless the two leading bits of lower are 10. Your expression may use <<, >>, ==, !=, and any constants you deem necessary. No other bitwise or relational operators are permitted for this part.
C) [5 points] c. ____C____ stands in for 1 – 5 lines of code to reconstruct the two-byte Unicode code point from UTF8 encodings captured by upper and lower. You may use any bitwise operators you like. Present the 1 - 5 lines of code needed to reconstruct codepoint before it's returned.
2) Tokenizing in Place  

Your second problem has you implement the following function:

```c
char *strseparate(char **p_string, const char *delimiters);
```

Provided `strseparate` is properly implemented, this program

```c
void print_tokens(const char *sentence) {
    char copy[strlen(sentence) + 1];
    strcpy(copy, sentence);
    char *curr = copy;
    while (*curr != '\0') {
        char *word = strseparate(&curr, ". !");
        printf("\"%s\"\n", word);
    }
}

int main(int argc, const char *argv[]) {
    print_tokens("This is a sentence!!");
    return 0;
}
```

outputs the following:

```
"This"
"is"
"a"
"sentence"
""
""
```

`strseparate` examines the C string reachable from `p_string`, locates the first occurrence of any character in `delimiters`, and overwrites it with '\0', thereby truncating the string. The address of the character following the freshly written '\0' is stored in the space addressed by `p_string`, and the original `char *` addressed by `p_string` is returned.

Presented below is a template implementation of `strseparate`.

---

**Solution:**

```c
#include <stdio.h>
#include <string.h>

char *strseparate(char **p_string, const char *delimiters) {
    for (char *p = *p_string; *p; p++) {
        while (*p && strchr(delimiters, *p)) p++;
        *p = '\0';
    }
    *p_string = (char *)((char *)p_string + strlen(*p_string));
    return *p_string;
}
```
char *strseparate(char **p_string, const char *delimiters) {
    char *front = ____A____;
    char *end = front + ____B____;
    ____C____
    return front;
}

Your job is to fill in the blanks with C expressions that collectively realize the desired functionality.
a. [3 points] ___A___ stands in for a single expression that initializes \texttt{front} to store a copy of the \texttt{char *} address pointed to by \texttt{p_string}. Present a simple expression involving \texttt{p_string} that should replace ___A___ if \texttt{front} is to be properly initialized.
b. [3 points] B stands in for a single call to a C string library function that computes the number of non-delimiters addressed by front. By computing this number and adding it to front, we initialize end to address the first delimiter beyond the zero or more non-delimiter characters that front addresses. Present a simple expression involving a C string library function that computes the number of non-delimiter characters currently addressed by front.
c. [4 points] _____C____ is a placeholder for 1-3 lines of code needed to complete the implementation of `strseparate`. These lines shouldn't need to call any library functions. They should stamp down the '\0' needed to separate the token being returned from the rest of the string (presumably to be parsed by future calls to `strseparate`). These lines must also take care to update the space addressed by `p_string`. Present up to three short lines of C code to complete the implementation.
3) Arrays of Suffixes 10 Points/45 Total

Given an array of C strings (all of which are guaranteed to be purely alphabetic, lower case, and nonempty), implement a function called broadcast that dynamically allocates, populates, and ultimately returns an array of 26 records of the following type:

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

All suffixes fields are initially set to NULL, and all count fields are initially set to 0. Here is broadcast’s prototype:

```c
entry *broadcast(char *strings[], size_t length);
```

Your broadcast function should traverse the supplied string array exactly once and populate the 0th slot of an array with copies of all strings beginning with the letter a, the 1st slot with all of the strings beginning with the letter b, and so forth, until all strings have been processed. Your algorithm should ensure the relevant suffixes array within each entry is always just as large as needed—that is, suffixes is initially NULL to reflect the absence of strings but needs to be repeatedly reallocated to be just large enough—and no larger—to store the strings that have been copied there so far. If, for instance, the supplied string array contains 54 strings beginning with the letter z, then your implementation will reallocate the suffixes field that exists on behalf of the letter z a total of 54 times.

One caveat: Since all of the words placed in the 0th array begin with the letter a, there’s no need to include the leading a of the word in the copy. Thus, if "abacus", "apopletic", and "azalea" are the only words in the supplied array that begin with the letter a, then the 0th slot of the array would ultimately be populated with the dynamically allocated strings "bacus", "popletic", and "zalea". Of course, each of these suffixes is a heap-based string that’s memory independent of the original, and everything illustrated here for words beginning with the letter a applies to all other letters too.

The code template below traverses the supplied array precisely one time, allocates space for the 26 entries, distributes each of the strings across the 26 entries, extending their suffixes arrays on an as-needed basis (never over-allocating), and then returning the array of entries.
entry *broadcast(char *strings[], size_t length) {
    entry *entries = malloc(___A____);
    for (size_t i = 0; i < 26; i++) {
        ___B___
    }
    for (size_t i = 0; i < length; i++) {
        size_t slot = strings[i][0] - 'a';
        ___C___
    }
    return entries;
}

Your task at hand is to tell us how the three placeholders should be filled in to provide a fully functional implementation.
a. [2 points] ____A____ stands in for the expression needed to dynamically allocate the space for just the array of entries. What should ____A____ be replaced with to always allocate the correct number of bytes? (You shouldn't make assumptions about how many bytes an entry is. The compiler decides that.)
b. [3 points] ____B____ should be the 2 – 4 lines of code needed to zero out all fields within all 26 entries. Supply the code snippet that should be used in place of ____B____.
c. [5 points] ____C____ represents the sequence of lines necessary to copy a particular suffix to the correct entry. Your implementation should be careful to extend some suffixes array by precisely one C string with each iteration and then ensure that a copy of the relevant suffix gets appended as the last item. Supply the code snippet that should be used to replace ____C____.
4) Finding Adjacent Matches

15 Points/45 Total

Implement the generic function `find_adjacent` to search the provided array for the first occurrence of two neighboring elements that match, copy the two neighboring elements into the space addressed by `result`, and return the index of the first of the two matching elements. (If there is no such pair, then the function should return the value of `nelems` as a sentinel that no match was found, and `result` can be ignored.)

The parameters are specified as follows:

- `base`: a pointer to the first element of an array.
- `nelems`: the number of elements in the provided array.
- `elem_size`: the size of a single array element, in bytes.
- `result`: the address of previously allocated space where the two matching elements should be copied. You may assume the space has been properly allocated and can be safely written to.
- `cmp_fn`: a function pointer that accepts two parameters, both pointers to elements of the relevant type, and returns a negative number if the first parameter is considered less than the second, 0 if the first parameter and the second parameter are considered equal, or a positive number if the first parameter is considered larger than the second.

Assume, for example, we call `find_adjacent` on the following int array of length 8:

```
[8, 4, 7, 4, 4, 1, 1, 9, 3]
```

Provided we supply a comparison function that orders numbers in ascending order, the call would return 3, which is the index of the first of the two matching 4's, and would copy the two 4's into the space addressed by `result`.

Here's the implementation of the function, minus some key parts you'll fill in:
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 = ____A____;
        void *second = ____B____ + elem_size;
        if (____C____) {
            ____D____
            return i;
        }
    }
    return nelems; // adjacent pair wasn’t found, return nelems
}

Your job here is to tell us how these four placeholders should be filled in to provide a fully functional implementation. A fifth part has you take on the role of client and write a comparison function that can be passed to find_adjacent.
a. [3 points] _____A_____ should be a valid expression that computes the address of the \(i^{th}\) array element. Your expression not only needs to compute the correct address, but it needs to always compile as well.
b. [2 points] ____B____ should be a valid expression that computes the address of the i + 1\(^{th}\) array element using first. Present that short code snippet needed to properly initialize second.
c. [3 points] _____C____ should be the full test involving the supplied comparison function. Present the expression involving `cmp_fn` that evaluates to `true` if and only if the neighboring elements match.
d. [3 points] _____D____ should be a single line of code that knows how to copy both of the two matching elements into the space addressed by result. You can assume that result addresses space large enough to accommodate the copy.
Suppose now that you have an array of C strings, and you’d like to find the first adjacent pair of strings that end in the same exact letter. For example, if you had the following declarations:

```c
char *neighbors[2];
```

The following code snippet should bind the variable `found` to 3 and copy the `char *`s leading to "Arushi" and "Indri" into the `neighbors` array.

```c
size_t found = find_adjacent(names, 7, sizeof(char *), neighbors, last_char_cmp);
```

e. [4 points] This, of course, depends on a working implementation of `last_char_cmp`. Your job for this final part is to implement the function to compare last characters of the strings. You may assume that all strings are at least one character long. Return negative numbers when the last character of the first string is alphabetically smaller than that of the second, 0 if those last characters match, and a positive number otherwise. You’re responsible for implementing the entire function. Recognize, of course, that most of the points are dedicated to how you interpret the two parameters.
C Strings

```c
size_t strlen(const char *str);
int strcmp(const char *s, const char *t);
int strncmp(const char *s, const char *t, size_t n);
char *strchr(const char *s, int ch);
char *strstr(const char *haystack, const char *needle);
char *strcpy(char *dst, const char *src);
char *strncpy(char *dst, const char *src, size_t n);
char *strcat(char *dst, const char *src);
char *strncat(char *dst, const char *src, size_t n);
size_t strspn(const char *s, const char *accept);
size_t strcspn(const char *s, const char *reject);
char *strdup(const char *s);
int atoi(const char *s);
long strtol(const char *s, char **endptr, int base);
```

Memory

```c
void *malloc(size_t sz);
void *calloc(size_t nmemb, size_t sz);
void *realloc(void *ptr, size_t sz);
void free(void *ptr);

void *memcpy(void *dst, const void *src, size_t n);
void *memmove(void *dst, const void *src, size_t n);
void *memset(void *base, int byte, size_t n);
```

Search and Sort
void qsort(void *base, size_t nelems, size_t width,
          int (*compar)(const void *, const void *));
void *bsearch(const void *key, const void *base, size_t nelems, size_t width,
              int (*compar)(const void *, const void *));
void *lfind(const void *key, const void *base, size_t *p_nelems, size_t width,
            int (*compar)(const void *, const void *));
void *lsearch(const void *key, void *base, size_t *p_nelems, size_t width,
              int (*compar)(const void *, const void *));

I/O

char *fgets(char buf[], int buflen, FILE *fp);