Problem 1: Linked Lists of Wedged Strings 10 points

Write a function `array_to_list` to convert an array of '\0'-terminated strings like this:

```
"Red"  "Pink"  "Purple"  "Blue"
```

into a linked list of perfectly sized character nodes, where each node wedges all of the characters of a string and the address of the next node into one contiguous block:

```
Red   Pink   Purple   Blue

Yellow Green Orange and so forth
```

Notice that each node of the list stores the individual characters of the string, followed by the '\0' (represented by a shaded box), followed by an eight-byte address identifying the location of the next node of the list (and if these eight bytes are all zeroes, they store a NULL and you’ve reached the end of the list.)

`array_to_list` takes a standard array of `char *`s along with its length and constructs the corresponding list of perfectly sized nodes. Your implementation:

- shouldn’t worry about alignment restrictions. In practice, padding would sometimes be inserted between the '\0' s and the pointers, but don’t worry about that here.
- should return the address of the first character in the first node of the list, or NULL if the array is empty
- should make use of malloc, strlen, and strcpy as needed.
- should be implemented iteratively, using a for loop that iterates over the array in reverse. In fact, the starter code we provide you includes such a loop.

Use the space provided to present your implementation.
Problem 2: Assembly Code Analysis

20 points

The assembly code presented below was generated by compiling a function called ella without optimization—i.e., using -Og, which is the standard flag used during development.

```
0x116d <+4>:     push   %r12
0x116f <+6>:     push   %rbp
0x1170 <+7>:     push   %rbx
0x1171 <+8>:     mov    %rdi,%r12
0x1174 <+11>:    mov    %rsi,%rbx
0x1177 <+14>:    lea    0x4(%rsi),%rbp
0x117b <+18>:    mov    (%rdi),%rdi
0x117e <+21>:    callq  0x1060 <strspn@plt>
0x1183 <+26>:    test   %rax,%rax
0x1186 <+29>:    je     0x1195 <ella+44>
0x1188 <+31>:    cmpb   $0x0,(%rbx)
0x118b <+34>:    jne    0x11a5 <ella+60>
0x118d <+36>:    mov    %rbp,%rax
0x1190 <+39>:    pop    %rbx
0x1191 <+40>:    pop    %rbp
0x1192 <+41>:    pop    %r12
0x1194 <+43>:    retq
0x1195 <+44>:    mov    %rbp,%rsi
0x1198 <+47>:    mov    %rbp,%rdi
0x119b <+50>:    callq  0x1060 <strspn@plt>
0x11a0 <+55>:    mov    %rax,%rbx
0x11a3 <+58>:    jmp    0x118d <ella+36>
0x11a5 <+60>:    mov    %rbp,%rsi
0x11a8 <+63>:    mov    %r12,%rdi
0x11ab <+66>:    callq  0x1169 <ella>
0x11b0 <+71>:    mov    %rax,%rbp
0x11b3 <+74>:    jmp    0x118d <ella+36>
```

a. [14 points, 2 - 3 points each] First, fill in the blanks below so that ella is programmatically consistent with the unoptimized assembly you see above. Note that the C code is nonsense and should just be a faithful reverse engineering of the assembly. You may not typecast anything. The same template you see below is also on the right, and you
should replace each of _____A_____, _____B_____, etc. with the C expressions or function calls they represent.

```c
char *ella(char *aretha[], char *diana) {
    char *vocalist = _____A____;
    if (_____B____)
        return _____C____;
    if (_____D____)
        return _____E____;
    return _____F____;
}
```

Now, study the more aggressively optimized version of `ella` presented below before answering the remaining questions. These short answer questions don’t really require an understanding of the `gcc` optimizations discussed in lecture, but rather a clear understanding of what the compiler can get away with to make code run more quickly.
b. [2 points] The unoptimized version pushes three caller-owned registers to the stack, and the optimized version only pushes two. Why doesn’t the optimized version need to push %r12?

c. [2 points] The unoptimized version clearly makes a recursive call to ella, whereas the second version doesn’t. What is the second version doing instead, and why can it do it?

d. [2 points] The unoptimized version uses callq to invoke strstr, whereas the optimized version uses jmpq instead. What does callq do that jmpq doesn’t, and why can the optimized version use jmpq instead of callq?
Problem 3: Ellipses, `printf`, and Crawling The Stack 10 points

The C standard allows for a variable number of arguments to be passed to a function, provided the prototype makes use of the **ellipses** to clarify where the required arguments end and the optional arguments begin. The prototype for `printf`, for instance, is:

```c
int printf(const char *control, ...); // we'll ignore the return value
```

The prototype mandates that a C string be passed in as the first argument. But after that, the caller can provide zero, one, two, three, or even more arguments beyond that.

For our purposes, assume the implementation of `printf` allocates enough stack memory to store all of the additional arguments, side by side, as a packed array of bytes. For instance, a call to `printf("%d %s %s\n", 153, "frisbee", "sunshine")` would prompt `printf` to allocate stack space for exactly 20 bytes—or rather, `sizeof(int) + 2 * sizeof(char *)` bytes—and populate that space with the four-byte data representation of 153, followed by the eight-byte data representation of the first `char *` value, followed by the eight-byte data representation of the second `char *` value. Assuming the 'f' of "frisbee" and the 's' of "sunshine" reside at addresses 0x453200 and 0x453220, respectively, the memory for this stack array would be laid out like this:

| 153 | 0x453200 | 0x453220 |

Of course, the example above generalizes to all reasonable calls to `printf`. For example:

- A call to `printf("%s %s %s %d %d", "abc", "def", "wxyz", 45, 55)` would allocate space for 32 bytes and pack together, in order, the three addresses and the two integers. `3 * sizeof(char *) + 2 * sizeof(int)` equal 32.
- A call to `printf("%d %d %s %d %d", 1, 2, "3", 4, 5)` would allocate space for 24 bytes: `int, int, char *, int, int`.

In all cases, `printf` essentially constructs a miniature stack frame and populates it with copies of any additional arguments in the order they were supplied.

Once `printf` assembles and populates the array, it then calls a helper function—a function called `myprintf`—and passes one the original `control` parameter verbatim and the base address of the packed array of bytes as arguments one and two.
void myprintf(const char *control, const void *args);

By doing so, `myprintf` has access to all of the material—the control string as a template of what to print and the additional values needed to fill in any placeholders—to publish the string to the console.

For simplicity, we’ll assume that the only placeholders ever present in control are `%d` and `%s`. You can further assume the following core helper operations figure out how to print a C string and an integer to standard output:

```c
void print_string(const char *str);
void print_int(int num);
```

The space below includes a partial implementation of `myprintf`, and you’re to complete it. You’ll do so by manually crawling down the array of memory addressed by `args`, using the control string to decide whether the next figure to be consumed from that memory is an int or a `char *`. Once implemented, your `myprintf` would contribute to the following:

```c
int main(int argc, const char *argv[]) {
    printf("My favorite numbers are %d, %d, and %d.\n", 28, 496, 8128);
    printf("My name is %s %s %s.\n", "Barbara", "Ann", "Cain");
    printf("You remind me of %s%d%s%s.\n", "R", 2, "D", "2");
    return 0;
}
```

and generate the following:

```
My favorite numbers are 28, 496, and 8128.
My name is Barbara Ann Cain.
You remind me of R2D2.
```

a. [6 points] A partial implementation of `myprintf` is presented on the right. You’re to work through the code I provide and complete its implementation. You can assume that `args` addresses a properly assembled array of manually packed bytes as described above. If there were no additional arguments, you can assume that `args` is NULL. You can also assume that every `%` in the control string will be following by either a 'd' or an 's'.

b. [4 points, 1 point each] Describe what would be printed by each of the following calls to `printf` if it relies on the `myprintf` you’ve just implemented. If the call generates a segmentation fault, then say so.
   - `printf("%s", 0, 0);`
   - `printf("%d", "107");`
   - `printf("%d %d", 555);`
   - `printf("lots of smoke and mirrors", "lots", "of", "them");`
Problem 4: Implicit Allocators with Headers and Footers 20 points

You are implementing a custom allocator that relies on an eight-byte header and an eight-byte footer—a replica of the same node’s header—that overlays the last eight bytes of a free node’s payload. The footer is used to implement left coalescing, which can be implemented to help malloc, free, realloc to reduce fragmentation.

The most significant bit of the header (and footer) records whether the node is free (1 free, 0 in-use). The second most significant bit records whether the node to its immediate left is free (1 free, 0 otherwise). The remaining 62 bits encode the size of the entire node—header plus payload—in bytes. All request sizes are rounded up to the nearest multiple of eight bytes, so all payload addresses are aligned accordingly.

In the above diagram, the 32-byte payload with base address 0x2008 is free, the 24-byte payload with base address 0x2030 is in use, and the 40-byte payload at address 0x2050 is free. Note this allocator is technically an implicit one, as there’s no mention of linked lists anywhere.

Assume the following #define constants and global variables have already been set up:

```c
#define HEAD_SIZE sizeof(size_t)
#define FOOT_SIZE HEAD_SIZE

// flags used to isolate free, left-free bits, and payload size from
#define FREE (1 << 63)
#define LEFT (1 << 62)
#define SIZE ______________

static size_t *heap_start; // base address of entire heap segment
static size_t heap_size;   // number of bytes in the entire heap seg
```

Answer each of the following questions using the space on the right. Some of the questions are short answer, and some require you to implement code.

a. [3 points] First off, note that the #define value for SIZE is blank. What expression—which you must frame in terms of FREE and LEFT—should be used so that SIZE is a
mask of 2 0's followed by 62 1's? (The SIZE mask can then be used to isolate the payload-size portion of a header or footer.)

b. [3 points] You wonder whether it make sense to #define FREE, LEFT, and SIZE to be 0x8000000000000000, 0x4000000000000000, and 0x3FFFFFFF000000, respectively, so that repeated reevaluation of 1 << 63, 1 << 62, and your expression for SIZE doesn’t impact allocator throughput. After using callgrind to profile the number of instructions executed on test scripts, you note that it doesn’t seem to make a difference, even when your allocator is compiled at –Og? Give a reasonable explanation why that might be.

c. [6 points] Complete the implementation of the count_available_bytes function, which scans the heap from front to back and returns the total number of available payload bytes. Your implementation will need to examine all nodes—both free and allocated—to compute the answer, since the allocator is an implicit one.

```c
size_t count_available_bytes() {
```

d. [6 points] Complete the implementation of coalesce_left, which accepts the address of a free node header and, if the node to its left is also free, merges the two into one larger node. If the node to the left isn’t free, then coalesce_left simply returns without doing anything.

```c
void coalesce_left(size_t *header) {
```
C Reference

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);

x86-64

Common instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov src, dst</td>
<td>dst = src</td>
</tr>
<tr>
<td>movsbl src, dst</td>
<td>byte to int, sign-extend</td>
</tr>
<tr>
<td>movzbl src, dst</td>
<td>byte to int, zero-fill</td>
</tr>
<tr>
<td>cmov src, reg</td>
<td>reg = src when condition holds, using same condition suffixes as jmp</td>
</tr>
<tr>
<td>lea addr, dst</td>
<td>dst = addr</td>
</tr>
<tr>
<td>add src, dst</td>
<td>dst += src</td>
</tr>
<tr>
<td>sub src, dst</td>
<td>dst -= src</td>
</tr>
<tr>
<td>imul src, dst</td>
<td>dst *= src</td>
</tr>
<tr>
<td>neg dst</td>
<td>dst = -dst (arith inverse)</td>
</tr>
<tr>
<td>imulq S</td>
<td>signed full multiply</td>
</tr>
<tr>
<td></td>
<td>R[rdx]:R[rax] &lt;- S * R[rax]</td>
</tr>
<tr>
<td>mulq S</td>
<td>unsigned full multiply</td>
</tr>
<tr>
<td></td>
<td>same effect as imulq</td>
</tr>
<tr>
<td>idivq S</td>
<td>signed divide</td>
</tr>
<tr>
<td></td>
<td>R[rdx] &lt;- R[rdx]:R[rax] mod S</td>
</tr>
<tr>
<td></td>
<td>R[rax] &lt;- R[rdx]:R[rax] / S</td>
</tr>
<tr>
<td>divq S</td>
<td>unsigned divide – same effect as idivq</td>
</tr>
<tr>
<td>cqto</td>
<td>R[rdx]:R[rax] &lt;- SignExtend(R[rax])</td>
</tr>
</tbody>
</table>
sal count, dst dst <<= count
sar count, dst dst >>= count (arith shift)
shr count, dst dst >>= count (logical shift)
and src, dst dst &= src
or src, dst dst |= src
xor src, dst dst ^= src
not dst dst = ~dst (bitwise inverse)

cmp a, b b-a, set flags
test a, b a&b, set flags

set dst sets byte at dst to 1 when condition holds, 0 otherwise, using same condition suffixes as jmp

jmp label jump to label (unconditional)
je label jump equal ZF=1
jne label jump not equal ZF=0
js label jump negative SF=1
jns label jump not negative SF=0
jg label jump > (signed) ZF=0 and SF=OF
jge label jump >= (signed) SF=OF
jl label jump < (signed) SF!=OF
jle label jump <= (signed) ZF=1 or SF!=OF
ja label jump > (unsigned) CF=0 and ZF=0
jae label jump >= (unsigned) CF=0
jb label jump < (unsigned) CF=1
jbe label jump <= (unsigned) CF=1 or ZF=1

push src add to top of stack
Mem[--%rsp] = src

pop dst remove top from stack
dst = Mem[%rsp++]
call fn push %rip, jmp to fn
call fn push %rip, jmp to fn

Instruction suffixes
b  byte
w  word (2 bytes)
l  long/doubleword (4 bytes)
q  quadword (8 bytes)

Suffix is elided when can be inferred from operands e.g. operand \%rax implies q, \%eax implies l, and so on

**Condition codes/flags**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZF</td>
<td>Zero flag</td>
</tr>
<tr>
<td>SF</td>
<td>Sign flag</td>
</tr>
<tr>
<td>CF</td>
<td>Carry flag</td>
</tr>
<tr>
<td>OF</td>
<td>Overflow flag</td>
</tr>
</tbody>
</table>

**Registers**

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rip</td>
<td>Instruction pointer</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rdi</td>
<td>1st argument</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument</td>
</tr>
<tr>
<td>%rdx</td>
<td>3rd argument</td>
</tr>
<tr>
<td>%rcx</td>
<td>4th argument</td>
</tr>
<tr>
<td>%r8</td>
<td>5th argument</td>
</tr>
<tr>
<td>%r9</td>
<td>6th argument</td>
</tr>
<tr>
<td>%r10,%r11</td>
<td>Callee-owned</td>
</tr>
<tr>
<td>%rbx,%rbp,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>%r12–%r15 Caller-owned</td>
</tr>
</tbody>
</table>

**Addressing modes**

Example source operands to `mov`

<table>
<thead>
<tr>
<th>Immediate</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mov $0x5, dst</td>
<td></td>
</tr>
<tr>
<td>$val</td>
<td></td>
</tr>
</tbody>
</table>

source is constant value
<table>
<thead>
<tr>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov %rax, dst</td>
</tr>
<tr>
<td>%R</td>
</tr>
<tr>
<td>R is register</td>
</tr>
<tr>
<td>source in %R register</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Direct</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov 0x4033d0, dst</td>
</tr>
<tr>
<td>0xaddr</td>
</tr>
<tr>
<td>source read from Mem[0xaddr]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov (%rax), dst</td>
</tr>
<tr>
<td>(%R)</td>
</tr>
<tr>
<td>R is register</td>
</tr>
<tr>
<td>source read from Mem[%R]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Indirect displacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov 8(%rax), dst</td>
</tr>
<tr>
<td>D(%R)</td>
</tr>
<tr>
<td>R is register</td>
</tr>
<tr>
<td>D is displacement</td>
</tr>
<tr>
<td>source read from Mem[%R + D]</td>
</tr>
</tbody>
</table>

| Indirect scaled-index |
mov 8(%rsp, %rcx, 4), dst
D(%RB,%RI,S)

RB is register for base
RI is register for index (0 if empty)
D is displacement (0 if empty)
S is scale 1, 2, 4 or 8 (1 if empty)
source read from Mem[%RB + D + S*%RI]

64-bit register, 32-bit sub-register, 16-bit sub-register, 8-bit sub-register

%rax, %eax, %ax, %al
%rbx, %ebx, %bx, %bl
%rcx, %ecx, %cx, %cl
%rdx, %edx, %dx, %dl
%rsi, %esi, %si, %sil
%rdi, %edi, %di, %dil
%rbp, %ebp, %bp, %bpl
%rsp, %esp, %sp, %spl
%r8, %r8d, %r8w, %r8b
%r9, %r9d, %r9w, %r9b
%r10, %r10d, %r10w, %r10b
%r11, %r11d, %r11w, %r11b
%r12, %r12d, %r12w, %r12b
%r13, %r13d, %r13w, %r13b
%r14, %r14d, %r14w, %r14b
%r15, %r15d, %r15w, %r15b