CS61C FA20 Quest

Instructors: Dan Garcia, Borivje Nikolic Head TAs: Stephan Kaminsky, Cece McMahon

Q1.1: Base Conversion

FOR THIS ENTIRE SECTION, YOU ARE NOT ALLOWED TO USE THE C PROGRAMMING LANGUAGE AS A CALCULATOR (OR A CALCULATOR EITHER, FOR THAT MATTER).

Convert 45_8 to base 3

Answer: `________`

Q1.2: Number Representation

FOR THIS ENTIRE SECTION, YOU ARE NOT ALLOWED TO USE THE C PROGRAMMING LANGUAGE AS A CALCULATOR (OR A CALCULATOR EITHER, FOR THAT MATTER).

How would you encode -18 using 6 bits in the following representations? Write `N/A` if it is impossible. Please write exactly 6 bits – include any leading zeros if needed, and DO NOT include a leading `0b`.

Representation	Answer
Unsigned	`________`
One's Complement	`________`
Two's Complement	`________`
Sign-magnitude	`________`
Bias (bias value: -31)	`________`

Q1.3: Matrix Addressing

FOR THIS ENTIRE SECTION, YOU ARE NOT ALLOWED TO USE THE C PROGRAMMING LANGUAGE AS A CALCULATOR (OR A CALCULATOR EITHER, FOR THAT MATTER).

You need to answer Option 1 AND Option 2 for full credit on this part.

Option 1: Matrix stored as 2D array

A matrix representation stores its values (for this problem, only bytes) such that the offset address of the element in row i and column j can be calculated by concatenating the binary representations of i and j, where row and column indicies start from 0. The size of the bit fields for row and column number should be big enough to accommodate the *highest possible row and column numbers.* **The amount of space allocated for this array will be the maxiumum number of elements that can be represented by this bit field, even if we won't use all of them.**

For example: A 3 (row) x 5 (column) matrix, A, will have a 2 bit row field and a 3 bit column field. The row bits will have values `0b00` through `0b10` (0-2), and the column bits will have values `0b000` through `0b100` (0-4). The element in row 2 and column 3 will have an offset address of `0b10011`. The element in row 1 and column 2 will have an offset address `0b01010`. **Even though this array only stores 15 bytes, space will allocated for 32 bytes due to the size of the 5 bit wide field.**

Part A

Matrix M has `maxrows` $= 13$ rows and `maxcols` $= 65$ columns. Find the offset address of the element in row `row` $= 7$ and column `col` $= 12$. Put your answer in *binary*, without the leading `0b`, and **without any leading zeros**, so if we calculated an offset address (from above) as `0b01010`, we would enter `1010`.

Answer: `0b` `________`

Part B

Which of the following C expressions will produce the desired result in Part A? (Choose one)

- `(maxrows << ceil(log2(maxcols))) & col`
- `(row << floor(log2(maxcols))) | col`
- `(row << ceil(log2(maxcols))) || col`
- `(maxrows << ceil(log2(maxcols))) && col`
- `(col << floor(log2(maxcols))) && row`
- `(col << ceil(log2(maxcols))) | row`
- `(row << ceil(log2(maxcols))) | col`
- `(maxrows << ceil(log2(maxcols))) || col`
- `(col << ceil(log2(maxcols))) || row`
- `(col << ceil(log2(maxrows))) && row`

Part C

What integer would you pass to `malloc` when allocating space for matrix `M`, which only stores a byte per value in the matrix? (Yes, we know there might be a lot of wasted space.)

Answer: `________`

Option 2: Matrix stored as 1D array

Another way of storing this array would be to store it in a column-major order one-dimensional array. A column-major order one-dimensional array is an array in which the columns of the given matrix are stacked one after another in the array. For example, given the matrix below:

1	2	3
4	5	6
7	8	9

We would get the array [1,4,7,2,5,8,3,6,9]

The amount of space allocated for an array is equivalent to the number of elements in the array. For example, if we were to store a 3 x 5 matrix, we would only need to allocate space for 15 elements. The rows would be numbered 0-2, and the columns would be numbered 0-4.

Part D

Recall matrix has `maxrows` $= 13$ rows and `maxcols` $= 65$ columns. Find the offset address of the element in row `row` $= 7$ and column `col` $= 12$. Just write the integer value, no need to write it in binary.

Answer: `________`

Part E

Which of the following C expressions will produce the desired result in Part D? (Choose one)

- `col + maxrows + row`
- `row + maxcols + col`
- `row + col`
- `(col * maxrows) + row`
- `(row * maxcols) + col`
- `row + maxcols`

Part F

What integer would you pass to `malloc` when allocating space for matrix `M`, which only stores a byte per value in the matrix? (Yes, we know there might be a lot of wasted space.)

Answer: `________`

Q2: Bit Manipulation

Write a function `bit_manip` that takes in an arbitrary 64-bit unsigned integer, and for every `N = 5` set of bits starting from LSB (considering the number zero-extended to a multiple of `N` bits), we rotate them right by `R = 3` bits and then turn on bit 2 and turn off bit 1 (within each group of 5 bits) and then return the final value.

As an example, if our input were a 16-bit number whose bits were `0bABCDEFGHIJKLMPQS` and `N = 5`, we rotate it right by `R = 1` bit, and `ON = 1` and `OFF = 3` then:

- `0bABCDEFGHIJKLMPQS` …being thought of as groups of `N = 5` bits would become
- `A BCDEF GHIJK LMPQS` …then zero-extended to a multiple of `N = 5` bits would become
- `0000A BCDEF GHIJK LMPQS` …after the rotate right by 1 bit, `R = 1`, would become
- `A0000 FBCDE KGHIJ SLMPQ` …after turning on bit `ON = 1`
- `A0010 FBC1E KGH1J SLM1Q` …after turning off bit `OFF = 3`
- `A0010 F0C1E K0H1J S0M1Q` …and returning the lowest 16 bits means we'd return…
- `0b0F0C1EK0H1JS0M1Q`

Download the included files: (shown below)

You may edit `main.c` to test your code. Do not edit the `bit_manip` signature. You will only be submitting `bit_manip.c` and only the code in that file will be graded.

`bitmanip.c`

```c
#include <inttypes.h>

uint64_t bit_manip(uint64_t num) {
    //YOUR CODE HERE
    return 0;
}
```

`main.c`

```c
#include <inttypes.h>

extern uint64_t bit_manip(uint64_t);

int main(int argc, char *argv[]) {
    return 0;
}
```

Q3: Split

Write a function `split` that takes a singly-linked list of words (strings of lower-cased characters terminated appropriately) and splits it into two new singly-linked lists based on whether the **first** letter of the word is a vowel (a, e, i, o, or u) or consonant (everything else). **All of the inital storage (nodes and strings) must be freed; you must copy strings to new locations in memory. Also, the output list should keep the same relative ordering of the input strings.** You must use `CS61C_malloc()` and `CS61C_free()` instead of `malloc()` and `free()`.

As an example, consider the following list (using Python-style list representation):

`words = ["zebra", "ant", "walrus", "bat", "emu"]`

If you called `split(words, &consonants, &vowels)`, it would set `consonants` to `["ant", "walrus", "bat"]`, and `vowels` to `["zebra", "emu"]` (with the strings stored in new locations in memory). Additionally, all of the space (nodes and strings) originally allocated for `words` will be freed.

You may find the following functions useful: `strcpy` and `strlen`:

- `char* strcpy(char* destination, const char* source);`
- `size_t strlen(const char *str);`

Download the included files: (shown below)

You may edit `main.c` to test your code. Your code should be able to run without modifications in `split.h`, or the `split` signature. You will only be submitting `split.c` and only the code in that file will be graded.

`split.c`

```c
#include "split.h"
#include <string.h>

void *CS61C_malloc(size_t size);
void CS61C_free(void *ptr);

/*
For reference, this is the Node struct defined in split.h:
typedef struct node {
  char *data;
  struct node *next;
} Node;
*/
void split(Node *words, Node **consonants, Node **vowels) {
    //YOUR CODE HERE
    return;
}
```

`split.h`

```c
typedef struct node {
  char *data;
  struct node *next;
} Node;

void split(Node * words, Node ** consonants, Node ** vowels);
```

---

main.c

```c
#include "split.h"
#include <stdlib.h>

void *CS61C_malloc(size_t size) {
    return malloc(size);
}

void CS61C_free(void *ptr) {
    free(ptr);
}

int main(int argc, char *argv[]) {
    return 0;
}
```