University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Summer 2015 Instructor: Sagar Karandikar 2015-07-09

CS61C MIDTERM 1

After the exam, indicate on the line above where you fall in the emotion spectrum between "sad" & "smiley"...

Last Name	**ANSWER KEY**
First Name	
Student ID Number	
Login	**cs61c-**
*The name of your **SECTION** TA (please circle)*	Derek | Harrison | Jeffrey | Nathaniel | Rebecca
Name of the person to your Left	
Name of the person to your Right	
*All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet. **(please sign)***	

Instructions (Read Me!)

- This exam contains 8 numbered pages including the cover page. **The back of each page is blank and can be used for scratch-work but will not be looked at for grading.** (i.e. the sides of pages without the printed "SID: _____" header will not even be scanned into Gradescope).
- Please turn off all cell phones, smartwatches, and other mobile devices. Remove all hats & headphones. Place your backpacks, laptops and jackets under your seat.
- You have 80 minutes to complete this exam. The exam is closed book; you may not use any computers, phones, wearable devices, or calculators. You may use one page (US Letter, front and back) of handwritten notes in addition to the provided green sheet.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided. "IEC format" refers to the mebi, tebi, etc prefixes.

	Q1	Q2	Q3	Q4	Q5	Q6	Total
Points Possible	13	13	14	15	20	15	90

SID: _____

Q1) Number Representation (13 pts)

a) Fill in the blanks with a letter (a, b, c or d) to match each expression in the left column, with an equivalent expression from the right column:

i) `(x >> 16) << 16` = **d** _____

ii) `x ^ x ^ x` = **c** _____

iii) `x ^ -1` = **b** _____

iv) `x & ~x` = **a** _____

a. **0**

b. **~x**

c. **x**

d. **x & 0xFFFF0000**

b) Rewrite the following numbers using IEC prefixes or as approximations using IEC prefixes:

1024 = **1 Ki**_____ 10^6 = **1 Mi**_____ 2^{43} = **8 Ti**_____

c) Convert the following 8 bit two's complement numbers into their decimal equivalents.

0x80 = **-128**_____ 0x7E = **126**_____

d) Four's complement is very similar to two's complement in the sense that negation consists of flipping digits and adding one to the "flipped" version of the value. To flip a value in 4's complement, 0's become 3's, 1's become 2's, and vice versa.

e.g. $3332_4 = -(-3332_4) = -(0001_4 + 1) = -(0002_4) = -(0 + 0 + 0 + 2*4^0) = -2$

A quaternary (base 4) digit is known as a crumb (in relation to bytes, nibbles, and bits). For the following questions, we would like to convert the following **4-crumb 4's complement** numbers (0cXXXX) to decimal. You may leave values in expression format. (Hint: How do we convert from hex to binary?)

0c2000 = **-128**_____ 0c1333 = **127**_____

Now using **8-crumb 4's complement**, what would the quaternary representation of the following be?

270 = **0c00010032**_____ -17 = **0c33333233**_____

SID: _____

Q2) Pointers and Memory Management (13 pts)

In this question, assume mallocs are always successful, pointers are 4 bytes, ints are 4 bytes, doubles are 8 bytes, and chars are 1 byte.

a) Which is the most efficient implementation of f() given below? Explain in a sentence or two.

```
typedef struct { int vals[1000000]; } dataStruct;

1. int f(dataStruct d){ printf("%d", d.vals[0]); }
2. int f(dataStruct *d){ printf("%d", d->vals[0]); }
3. int f(dataStruct **d){ printf("%d",(*d)->vals[0]); }
```

2_____ Explain: **it passes a pointer by value, rather than passing the whole struct by value, and only performs one dereference**

_____

b) Some of the code samples below contain issues with memory management. Below, identify the first instance of such a mistake in each code sample and briefly describe the issue. If there is no issue in the code sample, write "None".

i.)
```
double *pi_ptr;
pi_ptr = malloc(sizeof(pi_ptr));
*pi_ptr = 3.14;
```

Circle above and explain here: **The third line could segfault because you only malloc'ed space for a pointer, not a double**

_____

ii.)
```
char *a = "abcdef";
char **c = &a;
int x = 0;
while(**c != NULL){
  printf("%s", *(c + x));
  x++;
}
```

Circle above and explain here: **The access on the 5th line may segfault because we are not necessarily allowed to access the portion of memory following the location where char **c is stored. To fix, *(c+x) should be *c + x**

_____

c) Below are two renditions of similar code for a simple function **`foo().`** Identify the number of bytes stored in the stack, heap, and static up to the line marked with "🡐 here" for each piece of code. Include allocations only from the lines shown; assume registers are not used. You can assume that all calls to malloc succeed (do not return NULL).

i) Snippet #1:

```
char * str = "this string";

int foo (int a) {
     char* thisPtr = malloc(sizeof(char) * 12);
     thisPtr = str;
     char* thatPtr = "that string";
     int i = 0;
     while (thisPtr[i]) {
          if (thisPtr[i] != thatPtr[i])
               thisPtr[i] = thatPtr[i];
          i++;
     }
     ...// 🡐 here
```

Stack: 4+4+4+4____ Heap: 12_____ Static: 11+1+11+1+4_____

ii) Snippet #2: Now, follow the same steps as in the previous code snippet for the code below. However, the code below may contain bugs. If you find any potential bugs in the code below, explain what the bug is and then give the number of bytes in stack, heap, and static right before the bug occurs. You may assume that everything malloc'd is eventually freed (for example in the "…" section of the code). Therefore, "memory leak" is not a potential bug.

```
char *str = "this string";
char *str2 = "that string";
typedef struct{ char* str; int size;} bigStr;

int foo () {
     bigStr* thisPtr = malloc(sizeof(bigStr));
     bigStr* thatPtr = malloc(sizeof(bigStr));
     thisPtr->str = str;
     thatPtr->str = str2;
     int i;
     for(i=12; i>=0; i--) {
          thatPtr->str[i] = thisPtr->str[i];
     }
     ...// 🡐 here
}
```

Stack: 4+4+4_____ Heap: 4+4+4+4_____ Static: 11+1+11+1+4+4____

Errors (if any):

Potential segfault because index 12 is off the char array

SID: _____

Q3) Linked List (14 pts)

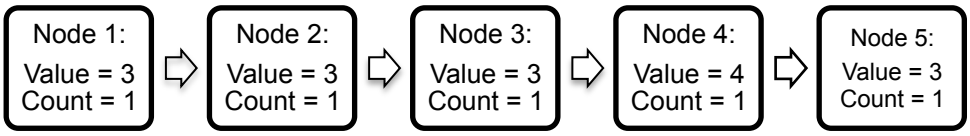
Run Length Encoding

Complete the **`compress_ll()`** function so that it will compress consecutive values in a linked list according to the provided struct below and update the count field in the struct, which represents the number of consecutive copies of the associated value we have seen so far. This compression is known as run-length encoding and is typically used for data compression.

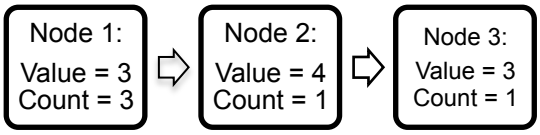
```
struct ll {
    int value;
    int count; // count used for following problem
    struct ll* next; // pointer to next element
};
```

For example, given the lists below:

Before:



After:



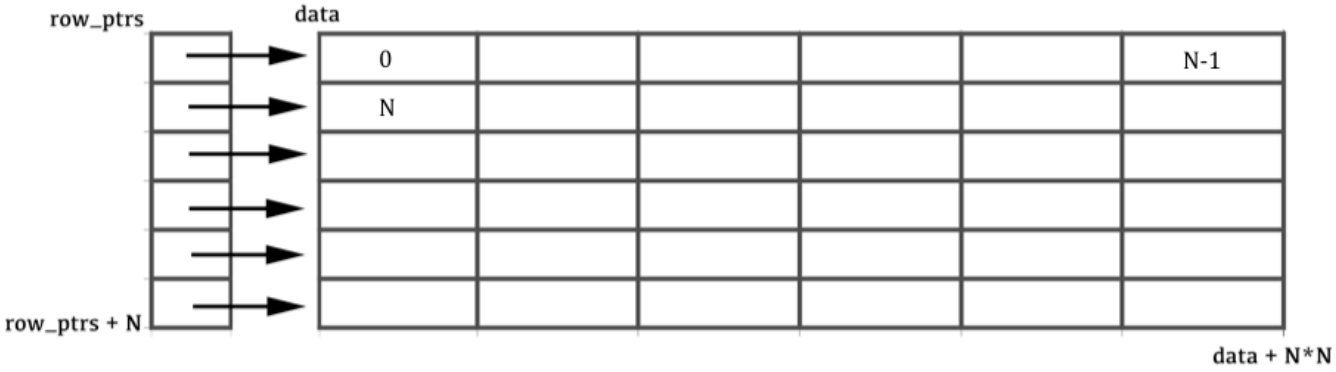
You may assume that we will pass in a valid node so you do not need to check that the initial node passed in is not NULL. You might not need all of the blank lines.
Note that all list nodes were created via dynamic memory allocation.

```
void compress_ll(struct ll** node, int curr_value) {
    struct ll* next_node = (*node)->next;

    if (next_node != NULL_____________________________________________________)
{
        int next_value = next_node->value;

        ____________________________________________________

        if (next_value == curr_value_____________________________________) {
            (*node)->count += 1;

            (*node)->next = next_node->next;___________

            free(next_node);_____________________________

            compress_ll(node, next_value);_____________
    } else {

           compress_ll(&next_node, next_value)___________

        ____________________________________________________
        }
    }
```

```
}
```

Q4) Array Hacking (15 pts)

Write code to dynamically allocate an NxN integer array laid out in contiguous memory that will function with the bracket notation **`array[x][y]`**. Specifically the array of N **`row_ptrs`** should be contiguous in memory and the entire data array of N*N elements should be laid out in contiguous memory (**`data[N*N - 1]`** should give you the last element and **`data[0]`** should give you the first element). **`row_ptrs[0]`** should be a pointer to the first element in data, **`row_ptrs[1]`** should be a pointer to the N-th element in **`data`**. You might not need all the blank lines.




```
int**_________ allocate_2d_array(int N){

     int*_________ data = malloc(sizeof(int_________)*N*N);

     int**_________ row_ptrs = malloc(sizeof(int*_________)*N);
     int i = 0;
     while(i < N){

          *row_ptrs = data + i*N_____________________________________;

          _____________________________________;
          i++; row_ptrs++;
     }

     ____________________________________________;

     return row_ptrs - N_____________________________________;
}
```

Write code to free all of the memory allocated in **`allocate_2d_array`**:

```
void deallocate_2d_array(int **row_ptrs){

     free(*row_ptrs)_____________________________________;

     free(row_ptrs)_____________________________________;
}
```

SID: _____

Q5) MIPS Bit Manipulation (20 pts)

Complete the MIPS function that **prints AND returns** the number of bits that are different between the upper 16 bits and the lower 16 bits of a 32-bit unsigned number passed in as an argument. Assume there's a print function (defined elsewhere, called `printer`) that prints the one integer argument given.

Please follow calling conventions; you might not need all of the lines below. On a line that is followed by a comment, you must write an instruction that obeys the operation indicated by the comment.

Two examples: `bitsDifferent(0xFFFF0001)` → 15. `bitsDifferent(0xFFFFFFFF)` → 0.

```
bitsDifferent:

    addi ____$sp, $sp, -16___________
    ___sw $ra, 0($sp)_________________

    ________________________
    ___sw $s0, 4($sp)_________________
    ___sw $s1, 8($sp)_________________

    ________________________
    li $s0, 0xFFFF

    _srl $s1, $a0, 16____#logical right shift $a0 by 16, put result in $s1
    __and $v0, $zero, $zero___________________
    __and $a0, $a0, $s0_______________________
    xor $s1, _$s1, $a0____________
loop:
    beq __$s1_______, $zero, exit
    __andi $t1, $s1, 1________________________
    __add $v0, $v0, $t1_______________________

    ________________________
    srl __$s1, $s1_____________, 1
    j loop
exit:
    move __$a0, $v0_________________
    sw ____$v0, 12__________($sp)
    __jal___________ printer
    __lw $ra, 0($sp)__________________________
    __lw $s0, 4($sp)__________________________
    __lw $s1, 8($sp)__________________________
    __lw $v0, 12($sp)_________________________
    __addi $sp, $sp, 16_______________________
    __jr $ra_________________ # return control to caller of bitsDifferent
```

SID: _____

Q6) C ⇔ MIPS (15 pts)

a) Fill in the blanks to translate between MAL MIPS, TAL MIPS, and Machine Code. Wherever you are given space to write both binary and hex for machine code, we will only grade the hex.

Address	MAL MIPS	TAL MIPS
0x102cff00	**sll $v0, $v0, 5**	sll $v0, $v0, 5
	0b **000000|00000|00010|00010|00101|000000 == 0x21140**	
0x102cff04	**beq___ $a0, $a1, Else**	**beq_____** $a0, $a1, Else
	0b 000100 | 00100 | 00101 | **0000000000000011**	
0x102cff08	**mul $t0, $a0, $a1**	**mul $a0, $a1**
0x102cff0c		**mflo_____** $t0
0x102cff10	**j Exit**	j Exit
	0b **000010|0000 0010 1100 1111 1111 0001 10 == 0x80b3fc6**	
0x102cff14	**Else:** <some instruction here>	
0x102cff18	**Exit: j Far**	**lui $at, 0x2000_____**
0x102cff1c		**ori $at, $at, 0x0004 _____**
0x102cff20		jr $at
	0b 000000 | **00001_____** | 00000 | 00000 | 00000 | 001000	
.....		
.....		
0x20000004	**Far: lw $v1, 0($v0)**	lw $v1,0($v0)
	0b **000010|0000 0010 1010 1111 1111 0001 10 = 0x8c430000**	