- HONOR CODE
- Questions Sheet.
- A Lets C. [6 Points]
  - 1. What type of address (heap,stack,static,code) does each value evaluate to Book1, Book1->name, Book1->author, &Book2? [4]
  - 2. Will all of the print statements execute as expected? If NO, write print statement which will not execute as expected?[2]
- B. Mystery [8 Points]
  - 3. When the above code executes, which line is modified? How many times? [2]
  - 4. What is the value of register a6 at the end ? [2]
  - 5. What is the value of register a4 at the end ? [2]
  - 6. In one sentence what is this program calculating ? [2]
- C. C-to-RISC V Tree Search; Fill in the blanks below [12 points]
- D. RISCV - The MOD operation [8 points]
  - 19. The data segment starts at address 0x10000000. What are the memory locations modified by this program and what are their values ?
- E Floating Point [8 points.]
  - 20. What is the smallest nonzero positive value that can be represented? Write your answer as a numerical expression in the answer packet? [2]
  - 21. Consider some positive normalized floating point number where p is represented as: What is the distance (i.e. the difference) between p and the next-largest number after p that can be represented? [2]
  - 22. Now instead let p be a positive denormalized number described asp = 2^y x 0.significand. What is the distance between p and the next largest number after p that can be represented? [2]
  - 23. Sort the following minifloat numbers. [2]
- F. Numbers. [5]
  - 24. What is the smallest number that this system can represent 6 digits (assume unsigned) ? [1]
  - 25. What is the largest number that this system can represent (assume unsigned). Expression is sufficient ? [1]
  - 26. Convert 122_10 to unsigned base 4. [1]
  - 27. 4s complement [1]
  - 28. Signed 4s. [1]
- G. Pointer Games [5]
  - 29. Line 5: x[strlen(x)] = '/ 0'; / *_**1** _ ***** /
  - 30. Line 9: printf("% d\n", *p); /** **2** **/
  - 31. printf("% s\n", cpy); /** **3** **/

- 32. printf("% s\n", a()); /** 4 **/
- 33. printf("% s\n", b()); /** 5 **/

HONOR CODE

- I have not used any online resources during the exam.
- I have not obtained any help either from anyone in the class or outside when completing this exam.
- No sharing of notes/slides/textbook between students.
- NO SMARTPHONES.

Questions Sheet.

Read all of the following information before starting the exam:

- For each question fill out the appropriate choice or write text on Canvas page. Also type clearly on in the exam on the appropriate text.
- IF THE MULTIPLE CHOICE ANSWER IS WRONG WE WILL MARK THE ANSWER WRONG. IF THE MULTIPLE-CHOICE ANSWER IS CORRECT, WE WILL READ THE WRITTEN PORTION.
- Show all work, clearly and in order, if you want to get full credit.
- I reserve the right to take off points if I cannot see how you logically got to the answer (even if your final answeris correct).
- Circle or otherwise indicate your final answers.
- Please keep your written answers brief; be clear and to the point.
- I will take points off for rambling and for incorrect or irrelevant statements. This test has six problems.

A Lets C. [6 Points]

```c
// You can assume the appropriate #includes have been done.
typedef struct Book{
  char *name;
  char *author;
} Book;

Book * createBook() {
    Book* Book = (Book*) malloc(sizeof(Book));
    Book->name = "this old dog";
    char author[100] = "mac demarco";
    Book->author = author;return Book;
}
int main(int argc, char **argv) {
      Book *Book1 = createBook();printf("%s\n", "Book written:");printf("%s\n", Book1->nam
    // print statement #1
    printf("%s\n", Book1->author);
    // print statement #2
    Book Book2;
    Book2.name = malloc(sizeof(char)*100);
    strcpy(Book2.name, Book1->name);
    Book2.author = "MAC DEMARCO";
    printf("%s\n", "Book written:");
    printf("%s\n", Book2.name);
    // print statement #3
    printf("%s\n", Book2.author);
    // print statement #4
    return 0;
}
```

1. What type of address (heap,stack,static,code) does each value evaluate to Book1, Book1->name, Book1->author, &Book2? [4]

Book1: Heap

Book1->title: Static

Book1->author: Stack

&Book2: Stack

2. Will all of the print statements execute as expected? If NO, write print statement which will not execute as expected?[2]

print statement #2

Let's break down the memory allocation details in the provided code:
This defines a structure Book with two fields: name and author, both of which are pointers to char (strings).

Createbook allocates memory for a Book structure on the heap and returns a pointer to this memory.
The allocated memory includes space for the pointers name and author.
Book->name = "this old dog";
This assigns the string literal "this old dog" to the name field of the Book structure. String literals are stored in the static data segment (or text segment), which is typically read-only and exists for the duration of the program.
char author[100] = "mac demarco";

This declares a local character array author of size 100 on the stack. The array is initialized with the string "mac demarco".
Book->author = author;
This assigns the address of the local author array to the author field of the Book structure. Since author is a local variable, it is stored on the stack.
Important Note: Since author is a local stack variable, it will go out of scope and be destroyed when the function createBook returns. This results in Book->author pointing to a memory location that is no longer valid (dangling pointer).

Book *Book1 = createBook();
The pointer Book1 points to the Book structure allocated on the heap by the createBook function.
Print Statements #1 and #2:
printf("%s\n", Book1->name); prints "this old dog".
printf("%s\n", Book1->author); may print garbage data or cause undefined behavior

since Book1->author points to a stack-allocated array that no longer exists after createBook returns.
Book Book2;

The variable Book2 is a Book structure allocated on the stack.

Book2.name = malloc(sizeof(char) * 100);
Memory is allocated on the heap for the name field of Book2.
strcpy(Book2.name, Book1->name);
The string "this old dog" is copied to the allocated memory for Book2.name.

Book2.author = "MAC DEMARCO";
The author field of Book2 is set to point to the string literal "MAC DEMARCO", which is stored in the static data segment.
Print Statements #3 and #4:
printf("%s\n", Book2.name); prints "this old dog".
printf("%s\n", Book2.author); prints "MAC DEMARCO".

Memory Summary
Book1: Allocated on the heap.
Book1->name: Points to a string literal in the static data segment.
Book1->author: Points to a now-invalid stack memory location (dangling pointer).
&Book2: The Book2 structure itself is allocated on the stack.
Book2.name: Points to a heap-allocated memory block where "this old dog" is copied.
Book2.author: Points to the string literal "MAC DEMARCO" in the static data segment.

```
 1  .globl main
 2
 3  .text
 4  main:
 5  li          a0,0
 6  li a1,1
 7
 8  la s0, L1
 9  lw  s1, 8(s0)
10  addi s2, zero, 6
11  addi s3, zero, 0
12  li   s4,8449
13  slli s4, s4,7
14
15  L1:
16  beq s3, s2, done
17  add s1, s1, s4
18  add a2, a0, a1
19  sw s1, 8 (s0)
20  addi s3, s3, 1
21  j L1
22
23  done:
24  li a0,10
25  ecall
```

3. When the above code executes, which line is modified? How many times? [2]

Line 18: add a2,a0,a1 6 times. Line 19 modifies line 18. The way it modifies is my adding s4 repeatedly to the encoding of line 18. Now expand s4 to a binary and see where the 1s line up. s4 in binary would have 00001 at rs1,rs2,rd fields. line 17 creates the new instruction encoding.

4. What is the value of register a6 at the end ? [2]

8. first iteration line 18 is `add a2,a0,a1` . Next iteration it is `add a3,a1,a2` , next iteration `add a4,a2,a3` ... and so on.

5. What is the value of register a4 at the end ? [2]

3

6. In one sentence what is this program calculating ? [2]

Fibonacci series

\pagebreak

C. C-to-RISC V Tree Search; Fill in the blanks below [12 points]

We're interested in running a search on a tree, and labeling the nodes in the order we finish examining them. Below we have the struct definition of a node in the tree, and the implementation of the function in C.

**Note that initially, all nodes in the graph have their label set to -1. The address width of our machine is 32 bits.
**

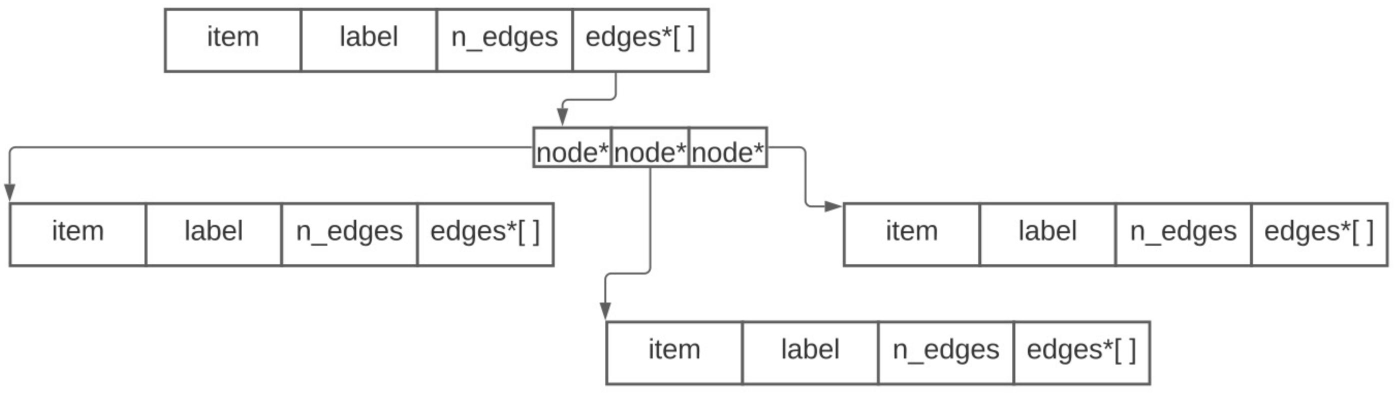
```c
struct node {
  int item;
  int label;
  int n_edges;
  struct node** edges[];
}

int label(struct node* nd, int counter) {

  if (nd->label != -1) {
    return counter;
  }

  for (int i = 0; i < nd->n_edges; ++i) {
    counter = label(nd->edges[i], counter);
  }

  nd->label = counter++;
  return counter;
}
```



Fill in the blanks in the code below

```
dfs_label:
# prologue
addi sp sp, -12   # Q7
# Q1. We need to make space for 3 registers on stack.
# 3*4bytes/register = 12
sw ra, 0(sp)
sw s0, 4(sp)
sw s1, 8(sp)
sw s2, 12(sp)

# a1 is counter. We do not need temporaries for counter
# since it is always overwritten
# by a call. a0 is node*
# Base case
addi t1, zero, -1 # Q8: Base case load immediate  -1
lw t0, 4 (a0)     # Q9 Load n->label for comparison to base case.
add a0,a1,zero    # Q10. Why have this instruction?
bne t0, t1, epilogue

# Loop header
add s0, a0, zero # Q11 Save node in a saved register.
# a0 is going to be eventually overwritten.
addi s1, zero, 0

loop:
lw t0 _8(s0)_    # Q12 Load in n_edges in t0
beq t0, s1, fin  # Q13 Iterator. If s1 end of loop iteration. We stop
lw a0, 12(s0)    # Q14 load edges into a0
sll t0, s1, 2    # Q15 Shift by 2 to get byte offset
add a0, a0, t0   # Get ptr for next node
lw a0, 0(a0)     # Q16 Load ptr into a0
jal label
addi a1, a0, 0   # Q17 Put return value into second argument for next call.
addi s1, s1, 1   # Increment counter after returning from recursion
j loop

fin:
sw a1 _4(s0)_    # Q18.
addi a1, a1, 1
# counter++;
# you have to increment after storing the value.
addi a0, a1, 0

epilogue:
lw ra (sp)
lw s0 4(sp)
lw s1 8(sp)
add sp, sp, 12
jr ra
```

\pagebreak

D. RISCV - The MOD operation [8 points]

We will be introducing a new instruction called "mod" in RISC-V which calculates the remainder. The semantics of the mod instruction (`mod dst,src1,src2`) are dst = src1%src2. e.g., lets say s1=5 s2=2 `mod s3,s1,s2` stores the value 2 in s3.

We will be using the mod operation in the program below called cipher.
You want to impress your friend, so you predict the result of executing the program as it is written, just by looking at it. If the program is guaranteed to execute without crashing, describe what it prints, otherwise explain the bug that may cause a crash.

```
.globl main
.data
a: .string "happy"
init: .string "XXXXX" # 12 Xs

.text

# C : Cipher(char* str).

cipher:
addi s3,zero,25

loop_header:
lbu s2, 0(a0) # Read character ch
beqz s2, end
addi s7,a0,0
addi a0,a0,1

loop:
addi s1, s2, -97 #
bltu s3,s1,loop_header
addi s2,s2,13
mod s2,s2,26 # New instruction
addi s2,s2,97
sb s2, 0(a1)
addi a1,a1,1
jal loop_header
end:
ret

main:
la a0, a
la a1,init
jal cipher
li a0,10
ecall
```

19. The data segment starts at address 0x10000000. What are the memory locations modified by this program and what are their values ?

ngvve : 0x10000006 0x100000A

Let’s break down the provided assembly program step-by-step. The program appears to implement a simple Caesar cipher, where each letter in a string is shifted by a certain number of positions in the alphabet.

Section Breakdown

Data Section

```
.data
a: .string "happy"
init: .string "XXXXX" # 5 Xs (not 12 as comment suggests)
```

- **`.data`** : This section is used for declaring initialized data or constants. Data declared in this section will remain fixed during runtime.
- **`a: .string "happy"`** : This declares a string “happy” with the label `a`. The string is stored in memory with a null terminator at the end.
- **`init: .string "XXXXX"`** : This declares a string “XXXXX” with the label `init`. It appears to serve as a placeholder for the encrypted output.

Cipher Function

Label `cipher`

```
cipher:
```

This is the entry point for the function `cipher`.

Initialization

```
addi s3, zero, 25
```

- **`addi s3, zero, 25`** : This sets register `s3` to 25. This value represents the wrap-around point for the Caesar cipher shift, indicating that if we exceed 25 (0-indexed), we need to loop back to the beginning of the alphabet.

Loop Header

```
loop_header:
lbu s2, 0(a0)       # Read character from string 'a' (str)
beqz s2, end        # If the character is null (end of string), jump to 'en
addi s7, a0, 0      # Store the current character pointer in s7
addi a0, a0, 1      # Increment pointer to the next character
```

- **`lbu s2, 0(a0)`** : Load the byte (character) at address `a0` into register `s2` .
- **`beqz s2, end`** : If `s2` is zero (indicating the end of the string), branch to `end` .
- **`addi s7, a0, 0`** : Copy the current value of `a0` to `s7` . This is not strictly necessary here and might be a remnant of debug or tracing code.
- **`addi a0, a0, 1`** : Increment `a0` to point to the next character in the string.

Inner Loop (Character Transformation)

```
loop:
addi s1, s2, -97   # Convert character to 0-25 range
bltu s3, s1, loop_header # If character not lowercase, restart
addi s2, s2, 13    # Shift character by 13 positions
mod s2, s2, 26     # Wrap around after 'z'
addi s2, s2, 97    # Convert back to ASCII
sb s2, 0(a1)       # Store the modified character in the output string
addi a1, a1, 1     # Move to next position in output string
jal loop_header    # Jump back to loop_header
```

- **`addi s1, s2, -97`** : Subtract 97 (ASCII value of ‘a’) from the character in `s2` to convert it to a range of 0-25. This assumes the input is a lowercase letter.
- **`bltu s3, s1, loop_header`** : If `s1` is less than `s3` (25), continue. This is actually redundant here because the check is against 25, which is the maximum valid `s1` value for lowercase letters.

- **`addi s2, s2, 13`** : Shift the character forward by 13 positions in the alphabet (ROT13 cipher).
- **`mod s2, s2, 26`** : This is a pseudo-code representation of modulo 26 operation. In actual RISC-V, there isn't a direct modulo instruction, but this implies taking the remainder after dividing by 26.
- **`addi s2, s2, 97`** : Convert back to ASCII by adding 97.
- **`sb s2, 0(a1)`** : Store the modified character at the current position in the output string (pointed by `a1`).
- **`addi a1, a1, 1`** : Move to the next position in the output string.
- **`jal loop_header`** : Jump back to the start of the loop to process the next character.

-
-

Main Function

```
main:
la a0, a          # Load address of the string 'a' into a0
la a1, init       # Load address of the string 'init' into a1
jal cipher        # Jump and link to the 'cipher' function
li a0, 10         # Load immediate 10 into a0
ecall             # Environment call (typically for program exit)
```

- **`main:`** : Label marking the entry point of the main function.
- **`la a0, a`** : Load the address of the string `a` ("happy") into register `a0` .
- **`la a1, init`** : Load the address of the output buffer `init` ("XXXXX") into register

`a1`.
- **`jal cipher`**: Call the `cipher` function to process the string.
- **`li a0, 10`**: Load the value 10 into `a0`, which typically signals a program exit in certain systems.
- **`ecall`**: System call to terminate the program.

Summary

The program implements a Caesar cipher (specifically a ROT13 cipher) that encrypts the string “happy” into a new string by shifting each letter by 13 positions in the alphabet. The output is stored in the `init` buffer. The program then exits after performing the encryption.

E Floating Point [8 points.]

The TAs get tired of having to convert floating-point values into 32 bits. As a result they propose the following smaller floating-point representation which is useful in a number of machine learning applications. It consists of a total of 8 bits as show below.

Sign	Exponent	Mantissa
1 bit	3 bits. Bias 3.	4 bits.

Exponent bias is 3.

Exponent	Significand	Meaning
0	0	0
7	non-zero	NaN
7	0	+- inf
0	non-zero	Denorm

- The largest exp remains reserved as in traditional floating point
- The smallest exp follows the same denormalized formula as traditional floating point
- Numbers are rounded to the closest representable value. Any numbers that have 2 equidistant representations are rounded down towards zero.

20. What is the smallest nonzero positive value that can be represented? Write your answer as a numerical expression in the answer packet? [2]

Exponent - 0
.0001 - Denormalized
= 2^{-2} * 0.0001 = 2^{-6}

21. Consider some positive normalized floating point number where p is represented as: What is the distance (i.e. the difference) between p and the next-largest number after p that can be represented? [2]

```
p= $2^y$ x 1.significand.
```

$2^{(y-4)}$

22. Now instead let p be a positive denormalized number described asp = 2^y x 0.significand. What is the distance between p and the next largest number after p that can be represented? [2]

2^{-6}

23. Sort the following minifloat numbers. [2]

```
0x04, 0xb2, 0x62, 0x45, 0x32
```

0x32 (1.125), 0xb2 (-1.125), 0x45 (2.625), 0x62 (8.75), 0x04 (0.0625)

0xb2 < 0x04 < 0x32 < 0x45 < 0x62

Sorting IEEE 754 floating-point numbers as if they were unsigned integers can be done due to the specific way floating-point numbers are represented in memory. This representation allows for a lexicographical ordering of the bits, which corresponds to the numerical ordering of the floats when interpreted as unsigned integers. Let's break down why this works

The exponent and mantissa are stored in a way that lexicographical ordering matches numerical ordering. The exponent's biased representation ensures that smaller exponents come before larger ones, and within the same exponent, the mantissa determines the ordering.

e.g., 0x45 and 0x62 (0x62 has higher exponent and hence will be a larger number). 0xb2 despite having larger exponent is a negative number and hence comes before these other numbers

20. In an IEEE-like floating-point system, the smallest nonzero positive value is usually represented by the smallest exponent (for denormalized numbers) and the smallest possible mantissa.

Steps to Find the Smallest Nonzero Positive Value:

1. **Sign Bit**: Since we are looking for a positive value, the sign bit is `0`.
2. **Exponent**:
   - The exponent is stored with a bias. For 3 bits, the bias is $2^{3-1} - 1 = 3$.
   - Exponent values can range from -3 (denormalized) to 4.
   - **Denormalized Numbers**: When the exponent is all zeros (`000`), the value is considered denormalized, meaning there is no implicit leading `1` in the mantissa. The actual exponent value in this case is $1 - \text{bias} = 1 - 3 = -2$.
3. **Mantissa**:
   - In a denormalized number, the mantissa does not have an implicit leading 1. The smallest nonzero mantissa is `0001` in binary, which corresponds to 1×2^{-4}.

Calculation:

For the smallest nonzero positive value:

- **Exponent**: -2 (since `000` is used for denormalized numbers)
- **Mantissa**: 0.0001 (binary), which is $\frac{1}{16}$ in decimal.

The value is given by:

$$\text{Value} = (-1)^{\text{sign}} \times (0.\text{mantissa}) \times 2^{\text{exponent}}$$

$$\text{Value} = 1 \times 0.0001_2 \times 2^{-2}$$

$$\text{Value} = \frac{1}{16} \times \frac{1}{4} = \frac{1}{64} = 2^{-6}$$

21. To calculate the distance between a positive normalized floating-point number

p and the next-largest representable number after p, we need to understand how floating-point numbers are structured.

Floating-Point Number Representation:

- Consider a normalized floating-point number p in the form:

$$p = 2^y \times (1.\text{mantissa})$$

where:
  - y is the exponent.
  - The mantissa is a fractional part in the binary form $1.m$, where m is the mantissa bits.

Next-Largest Representable Number:

- The next-largest representable number after p will have the same exponent y, but the smallest possible increment in the mantissa.
- The smallest increment in the mantissa for a normalized number is $\frac{1}{2^n}$, where n is the number of bits in the mantissa.

Thus, the next largest number can be represented as:

$$p_{\text{next}} = 2^y \times \left(1.\text{mantissa} + \frac{1}{2^n}\right)$$

Distance (Difference) Between p and p_{next}:

The difference between p_{next} and p is:

$$\Delta p = p_{\text{next}} - p = 2^y \times \left(1.\text{mantissa} + \frac{1}{2^n}\right) - 2^y \times (1.\text{mantissa})$$

$$\Delta p = 2^y \times \frac{1}{2^n} = 2^{y-n}$$

Conclusion:

The distance between a positive normalized floating-point number p and the next-largest representable number after p is $\boxed{2^{y-n}}$, where y is the exponent and n is the number of bits in the mantissa.

Example:

Step 1: Represent the Number p

Given: - **Exponent** $y = 3$ - **Mantissa** 1.0000_2 (with 4 bits)

The floating-point number p can be expressed as:

$$p = 2^3 \times 1.0000_2$$

Convert the binary mantissa 1.0000_2 to a sum of powers of 2:

$$1.0000_2 = 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4}$$

Step 2: Find the Next-Largest Representable Number After p

The next number after 1.0000 is 1.0001

Convert the new mantissa 1.0001_2 to a sum of powers of 2:

$$1.0001_2 = 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4}$$

2^{-1} or 2^{3-4} where y = 3

22. Now instead let p be a positive denormalized number described asp = 2y x 0.significand. What is the distance between p and the next largest number after p that can be represented?

When dealing with denormalized (or subnormal) numbers in a floating-point representation, the situation is slightly different from normalized numbers. Denormalized numbers are represented without an implicit leading 1 in the significand (mantissa). Instead, the significand begins with a leading 0.

Structure of a Denormalized Number:

- The general form of a denormalized number is:

$$p = 2^y \times 0.\text{significand}$$

  where:
  - y is the exponent, and for denormalized numbers, the exponent is the smallest possible value (usually $y = 1 - \text{bias}$).
  - 0.significand represents the significand, which is a fractional binary value.

Distance Between p and the Next-Largest Representable Number:

For a denormalized number, the next largest representable number is obtained by incrementing the significand by the smallest possible value, which is $\frac{1}{2^n}$, where n is the number of bits in the significand.

Thus, the next-largest number can be represented as:

$$p_{\text{next}} = 2^y \times \left(0.\text{significand} + \frac{1}{2^n}\right)$$

Calculation of the Distance:

The distance between p and p_{next} is:

$$\Delta p = p_{\text{next}} - p = 2^y \times \left(0.\text{significand} + \frac{1}{2^n}\right) - 2^y \times 0.\text{significand}$$

$$\Delta p = 2^y \times \frac{1}{2^n} = 2^y \times 2^{-n} = 2^{y-n}$$

Conclusion:

For a positive denormalized floating-point number $p = 2^y \times 0.\text{significand}$, the distance between p and the next-largest representable number after p is $\boxed{2^{y-n}}$, where y is the exponent for denormalized numbers (typically $y = 1 - \text{bias}$), and n is the number of bits in the significand.

F. Numbers. [5]

We are going to be creating a new base system for numbers based on 4s.
So every number is going to represented as a power of 4. A base 4 system has 3 symbols at most (0…3). e.g., 22_4 is equal to 2 \times 4^0 + 2 \times 4^1 = 2 + 8 = 10_{10}.

24. What is the smallest number that this system can represent 6 digits (assume unsigned) ? [1]

0

25. What is the largest number that this system can represent (assume unsigned). Expression is sufficient ? [1]

4095. $4^6 - 1$

333333_4
$\Sigma_{i=0}^{5} 3 * 4^i$

$3 * 4^5 + 3 * 4^4 + 3 * 4^3 + 3 * 4^2 + 3 * 4^1 + 3 * 4^0$ = 3*(1024+256+64+16+4+1)=

26. Convert $122_1 0$ to unsigned base 4. [1]

122 = $1 \times 4^3 + 3 \times 4^2 + 2 \times 4^1 + 2 \times 4^0$

1322_4

27. 4s complement [1]

Recall 2s complement where we add a bias to represent -ve numbers.
Consider a 4 digit 2s complement system. In unsigned form it can represent all numbers in range 0…15. In 2s complement form we use a negative bias to represent integers in the negative range. For instance in 2s complement with 4 digit numbers we use a negative bias of -16.

Decimal Number	Two’s Complement
-8	1000
-7	1001

Decimal Number	Two’s Complement
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Suppose we wanted to use a 4s complement notation. If we are working with a 4 digit base 4 number, what should we choose as our bias? Our bias should create roughly equal amounts of negative and positive numbers for our range.

Answer: 4^4

28. Signed 4s. [1]

For each number, we will reserve the most significant digit to strictly be used as a sign bit [1: negative 0:postivie]. (e.g., 0001: +1, 1020: -8, 2020 - Not valid sign bit something other than 1/0). What is the number of numbers that can be represented using this notation.

Answer : 2*4^3=128

G. Pointer Games [5]

```c
char *feels;
char *message(char *msg) {
  char *x = malloc(sizeof(char) * (strlen(msg) + 1));
  strncpy(x, msg, strlen(msg));
  x[strlen(x)] = ‘/ 0’; / ****1 * *** /
  return x;
}
void p_int(int *p) {
  printf("% d\n", *p); /**** 2 ****/
  }

void p_msg(char *str) {
  char *cpy = calloc(strlen(str) + 1, 1);
  strncpy(cpy, str, strlen(str));
  printf("% s\n", cpy); /**** 3 ****/
}
char *a() {
  char res[7] = " rules";
  return res;
}
char *b() {
  char *var = "cmpt295";
  return var;
}
void c() {
  printf("% s\n", a()); /**** 4 ****/
  printf("% s\n", b()); /**** 5 ****/
}
int main() {
  int y;
  feels = malloc(3);
  strcpy(feels, "hi");
  message(feels);
  p_int(&y);
  p_msg(feels);
  c();
}
```

There are comments on lines with numbers from 1-5. Each of these lines dereferernce a pointer. Characterize if these memory accesses are legal c. We will use the following terminology

- Legal:
- **Initialized**: Is there actual meaningful data in contents (data at each address) or is it garbage.
- **Illegal**: This line will always dereference an address the program doesn’t have explicit access to

- **Possibly Legal**: The operation could result in only dereferences of legal addresses but it’s also possible that in other runs on the program illegal accesses occur.

Mark which of the following apply to questions below:

29. Line 5: x[strlen(x)] = ‘/ 0’; / *_1 _ * /

Possibly legal. Malloc of x could fail

30. Line 9: printf("% d\n", *p); /** 2 **/

Legal, but uninitialized.

31. printf("% s\n", cpy); /** 3 **/

Legal and initialized. strncpy would implicitly copy

32. printf("% s\n", a()); /** 4 **/

illegal

33. printf("% s\n", b()); /** 5 **/

legal and initalized.