1 Number Representation

1.1 What is the range of integers represented by a n-bit binary number? Your answers should include expressions that use 2^n.

(a) Unsigned:

$[0, 2^n - 1]$

(b) Two's Complement:

$[-2^{n-1}, 2^{n-1} - 1]$

(c) Bias (with bias b):

$[-b, 2^n - 1 - b]$

1.2 How many unique integers can be represented in each case?

(a) Unsigned:

2^n

(b) Two's Complement:

2^n

(c) Bias (with bias b):

2^n

For both unsigned and two's complement, each bit string corresponds to a different integer, so we have 2^n unique integers.

Bias is just a shifted version of unsigned and so it can represent the same number of integers.

2 Memory Addresses

2.1 Consider the following C program:

```
int a = 5;
int main()
{
    int b = 0;
    char* s1 = "cs61c";
    char s2[] = "cs61c";
    char* c = malloc(sizeof(char) * 100);
    return 0;
};
```

For each of the following values, state the location in the memory layout where they are stored. Answer with code, static, heap, or stack.

(a) `s1`

stack

(b) `s2`

stack

(c) `s1[0]`

static

(d) `s2[0]`

stack

(e) `c[0]`

heap

(f) `a`

static

2.2 Consider the C code here, and assume the malloc call succeeds. Rank the value of each variable (which may be an address!) from 1 to 5 (with 1 being the least) right before bar returns. Use the memory layout from class; Treat all addresses as unsigned numbers.

```
#include <stdlib.h>

int FIVE = 5;

int bar(int x) {
    return x * x;
}

int main(int argc, char *argv[]) {
    int *foo = malloc(sizeof(int));
    if (foo) free(foo);
    bar(10); //  snapshot just before it returns
    return 0;
}
```

```
foo:   _______
&foo:  _______
FIVE:  _______
&FIVE: _______
&x:    _______
```

```
foo:   3
&foo:  5
FIVE:  1
&FIVE: 2
&x:    4
```

Going in order numerically, `FIVE` itself contains the value "5", `&FIVE` contains the address of `FIVE` and since it is a global variable, it is stored statically, which means that it will be stored in the data segment. Since `foo` is a pointer, it contains the address of whatever it was assigned to, which in this case, is `malloc(sizeof(int))`. As a result, `foo` is stored on the heap, so it is above the data segment and so the value it contains will be larger than the address of `FIVE`, since the heap is above the data segment. `&foo` itself lives in on the stack, since the space that it takes to store the pointer to the data that foo holds is allocated on the stack. This is above the heap. `x` is a local variable, so it also gets allocated on the stack so its address is also greater than the value stored in `foo`; the reason `&x` is smaller than `&foo` is simply because the stack grows downwards, and during the execution of the program, the space for `foo` is allocated before the space for `x`, so foo lives in higher memory than `x`.

3 Linked Lists Revisited

3.1 Fill out the declaration of a singly linked linked-list node below.

```
typedef struct node {
    int value;
    ____________ next; // pointer to the next element
} sll_node;
```

```
struct node* next;
```

Remember the pointer to the next node in a linked list is one pointing to another node, so the type of next is a pointer to the same type as the first linked list node.

3.2 Let's convert the linked list to an array. Fill in the missing code.

```
int* to_array(sll_node *sll, int size) {
    int i = 0;
    int *arr = _________________;
    while (sll) {
        arr[i] = ___________;
        sll = ___________;
        _______;
    }
    return arr;
}
```

```
int* to_array(sll_node *sll, int size) {
    int i = 0;
    int *arr = malloc(size * sizeof(int));
    while (sll) {
        arr[i] = sll->value;
        sll = sll->next;
        i++;
    }
    return arr;
}
```

Converting the linked list to an array requires traversing the linked list. But first, you must allocate enough space to store `size` number of integers. Then, you can go ahead and iterate over the linked list. Assign to the array each corresponding linked list value. Move the pointer of the linked list and increment the array counter after each assignment.

3.3 Finally, complete the function `delete_even()` that will delete every second element of the list. For example, given the lists below:

```
Before: Node 1 → Node 2 → Node 3 → Node 4
After: Node 1 → Node 3
```

Calling `delete_even()` on the list labeled "Before" will change it into the list labeled "After". All list nodes were created via dynamic memory allocation.

```
void delete_even(sll_node *s11) {
    sll_node *temp;
    if (!sll || !sll->next) {
        return;
    }
    temp = ________;
    sll->next = ________;
    free(________);
    delete_even(________);
}
```

```
void delete_even(sll_node *s11) {
    sll_node *temp;
    if (!sll || !sll->next) {
        return;
    }
    temp = sll->next;
    sll->next = temp->next (or sll->next->next);
    free(temp);
    delete_even(sll->next);
}
```

4 Floating Point Intro

The IEEE standard defines a binary representation for floating point using **sign**, **significant**, and **mantissa**.

Sign	Exponent	Significand
1 bit	8 bits	23 bits

For normalized floats:
Value = $(-1)^{\text{Sign}} \times 2^{(\text{Exponent} - \text{Bias})} \times 1.\text{significand}_2$

For denormalized floats:
Value = $(-1)^{\text{Sign}} \times 2^{(\text{Exponent} - \text{Bias} + 1)} \times 0.\text{significand}_2$

Exponent	Significand	Meaning
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

4.1 (a) How would 10.625 be represented in floating point format?

$10.625 = 8 + 2 + \frac{1}{2} + frac18ß1010.101_2$ in binary
$1010.101_2 = 1.010101_2 x 2^3 \rightarrow$ Exp $-127 = 3$
Sign: 0, Exponent: $130 = 10000010_2$, Significand: 010101_2
`0100 0001 0010 1010 0000 0000 0000 0000 = 0x412A0000`

(b) What decimal number is encoded as `0xC0A80000`?

`0xC0A80000` $= 11000000101010000000000000000000_2$
Sign: 1, Exponent: $10000001_2 = 129$, Significand: 0101_2
$(-1) \times 1.0101_2 \times 2^{129-127} = -1.0101_2 \times 2^2 = -101.01_2 = -5.25$

(c) How many non-negative floats are strictly less than 2?

Only possibility is if the exponent is within the range $[0, 127]$, as any value > 127 would make the float $>= 2$. Then, we can have any permutation of the significand without increasing the number by more than 1. Thus, there are $2^7 \times 2^{23} = 2^{30}$ floats < 2.

(d) What is the smallest positive value that can be stored using a single precision float?

`0x00000001` $= 2^{-23} * 2^{-126}$

5 More Floating Point (Sp18 Final)

5.1 IEEE 754-2008 introduces half precision, which is a binary floating-point representation that uses 16 bits: 1 sign bit, 5 exponent bits (with a bias of 15) and 11 significand bits. This format uses the same rules for special numbers that IEEE754 uses. Considering this half-precision floating point format, answer the following questions:

(a) For 16-bit half-precision floating point, how many different valid representations are there for NaN?

$2^{11} - 2$

(b) What is the smallest non-infinite number it can represent? You can leave your answer as an expression.

bias $= 2^{5-1} - 1 = 2^4 - 1 = 15$ Binary representation is: 0 00000 0000000001 $= 2^{-14} * 2^{-10} = 2^{-24}$

(c) What is the largest non-infinite number it can represent? You can leave your answer as an expression.

Binary representation is: 0 11110 1111111111 $= 2^{16} - 2^5 = 65504$

(d) How many floating point numbers are in the interval [1, 2) (including 1 but excluding 2?

2^{10} 0b0 01111 0000000000 $= 1$ 0b0 10000 0000000000 $= 2$ Keeping the exponent fixed at 01111 and allowing the significand to change, we can reach any number in the range [1, 2). There are 2^{10} possible significands.