IMPORTANT: Explain your answer briefly. Do not just write a short answer or fill the assembly code.

1. [10pts] The following code fragment has a potential vulnerability.

```
void* copy_elements(void *ele_src[], int ele_cnt, size_t ele_size) {
    /*
     * Allocate buffer for ele_cnt objects, each of ele_size bytes
     * and copy from locations designated by ele_src
     */
    void *result = malloc(ele_cnt * ele_size);
    if (result == NULL)
             /* malloc failed */
             return NULL;
    void *next = result;
    int i;
    for (i = 0; i < ele_cnt; i++) {
        /* Copy object i to destination */
        memcpy(next, ele_src[i], ele_size);
             /* Move pointer to next memory region */
             next += ele_size;
    }
    return result;
}
```

A. [4pts] Write possible values for ele_cnt and ele_size to crash the application/system. Explain the reason.

B. [6pts] Write extra C codes to add before the malloc call to prevent the crash.

2. [10pts][floating point data representation]

   A. [4pts] Assume variables x, f, and d are of type `int`, `float`, and `double`, respectively. (Neither f nor d equals to +infinity, -infinity, or NaN). For each of the following expressions, either argue that it is always true or give a counterexample if it is not.

      `A-1) x == (int) (double) x`

      `A-2) f == -(-f)`

      `A-3) 1.0/2 == 1/2.0`

      `A-4) d*d >= 0.0`

      `A-5) (f+d) – f == d`

   B. [3pts] Write the rounded binary numbers for the following values. They should be rounded to nearest 1/4 (2 bits fright of binary point, and must use "round-to-even" rule.

      Explain the advantage of such "round-to-even" rule, compared to round-down or round-up.

      10.00011 => _______________
      10.00110 => _______________
      10.11100 => _______________
      10.10100 => _______________

   C. [3pts] Explain how a floating point compare instruction (fcmp) can be implemented for the IEEE fp format. How will it be different from the integer compare instruction (cmp)?

3. [5pts] In the following code fragment, argue whether b and c always have the same value or not.

```
typedef union {
  float f;
  unsigned u;
} bit_float_t

float bit2float (unsigned u) {
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}

unsigned  a = random();
float b = bit2float(a);
 float c = (float) a;
```

4. [8pts] Answer the two questions for the following C function and the corresponding assembly code.

```
long rfun(unsinged long x) {

   if ( ______________________ )
       return ________________;

   unsigned long nx = _____________;
   long rv = rfun(nx);
   return ____________;
}
```

```
rfun:
     pushq     %rbx
     moveq     %rdi, %rbx
     movl      $0, %eax
     testq     %rdi, %rdi
     je        .L2
     shrq      $2, %rdi
     call      rfun
     addq      %rbx, %rax
.L2:
     popq      %rbx
     ret
```

A. [3pts] What value in the C code does rfun store in %rbx?

B. [5pts] Fill in the missing expressions in the C code.

5. [7pts] The following two C code fragments show two different ways of supporting 2D data structures.

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};

int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3

int *univ[UCOUNT] = {mit, cmu,
ucb};

int get_univ_digit (size_t index,
size_t digit){

  return univ[index][digit];

}
```

A. [4pts] Complete the following assembly codes for the two access functions. (explain your answer)

```
1)________________________
addl %rax, %rsi
movl  2)__________, %eax
ret
```

```
salq    $2, %rsi
addq    3)__________, %rsi
4)___________________________
ret
```

B. [3pts] Discuss the pros and cons of the two approaches.

6. Find the minimum number of operations to implement the given functions.

- Points will not be given if functions are implemented more than your minimum number of operations.
- You should justify your solution by implementing the functions.
- Assignment operator '=' is legal. But it will not be counted as an operator.

(1) addOK [5pts]

Description	Determine if can compute x+y without overflow
Examples	addOK(0x80000000,0x80000000) = 0 addOK(0x80000000,0x70000000) = 1
Legal Ops	! ~ & ^ | + << >>

The minimum number of operations to implement this function is _______.

int addOK(int, int)
{
 	
}

7. [10pts] Fill in the blanks to implement float_half.

- Each blank does not exceed a single C statement.
- Some blanks are fake. In other words, some blanks does not need to be filled in.

Description	Return bit-level equivalent of expression 0.5*f for floating point argument f. Both the argument and result are passed as unsigned int's, but they are to be interpreted as the bit-level representation of single-precision floating point values. When argument is NaN, return argument.
Legal Ops	Any integer/unsigned operations incl. ||, &&. also if, while

```
unsigned float_half(unsigned uf) {
  unsigned sign = uf>>31;
  unsigned exp = uf>>23 & 0xFF;
  unsigned frac = uf & 0x7FFFFF;
  unsigned something = (frac & 0x3) == 3;

  [                 ( a )                 ]

if (exp == 0) {
  [                 ( b )                 ]
   frac = [             ( c )             ] ;
  } else if (exp < 0xFF) {
  [                 ( d )                 ]
   if (exp == 0) {
  [                 ( e )                 ]
     frac = [            ( f )            ] ;
   }
  }
  return  [   ( g )   ]  |  [   ( h )   ]  |  [   ( i )   ]  ;
}
```

8. [10pts] Solve the following problems with the given assembly code.

C code

```
void phase(char *input)
{
   int i;
   int numbers[6];
   read_six_numbers(input, numbers);
   for(i = 1; i < 6; i++) {
     if ( [                    ] )
        explode_bomb();
   }
}
```

Assembly

```
Dump of assembler code for function phase:
   0x000055555555522a <+0>:     push   %rbp
   0x000055555555522b <+1>:     push   %rbx
   0x000055555555522c <+2>:     sub    $0x28,%rsp
   0x0000555555555230 <+6>:     mov    %fs:0x28,%rax
   0x0000555555555239 <+15>:    mov    %rax,0x18(%rsp)
   0x000055555555523e <+20>:    xor    %eax,%eax
   0x0000555555555240 <+22>:    mov    %rsp,%rbp
   0x0000555555555243 <+25>:    mov    %rsp,%rsi
   0x0000555555555246 <+28>:    callq  0x555555555820 <read_six_numbers>
   0x000055555555524b <+33>:    mov    %rsp,%rbx
   0x000055555555524e <+36>:    add    $0x14,%rbp
   0x0000555555555252 <+40>:    jmp    0x55555555525d <phase+51>
   0x0000555555555254 <+42>:    add    $0x4,%rbx
   0x0000555555555258 <+46>:    cmp    %rbp,%rbx
   0x000055555555525b <+49>:    je     0x55555555526f <phase+69>
   0x000055555555525d <+51>:    mov    (%rbx),%eax
   0x000055555555525f <+53>:    lea    0x2(%rax,%rax,2),%eax
   0x0000555555555263 <+57>:    cmp    %eax,0x4(%rbx)
   0x0000555555555266 <+60>:    je     0x555555555254 <phase+42>
   0x0000555555555268 <+62>:    callq  0x5555555557fa <explode_bomb>
   0x000055555555526d <+67>:    jmp    0x555555555254 <phase+42>
   0x000055555555526f <+69>:    mov    0x18(%rsp),%rax
   0x0000555555555274 <+74>:    xor    %fs:0x28,%rax
   0x000055555555527d <+83>:    jne    0x555555555286 <phase+92>
   0x000055555555527f <+85>:    add    $0x28,%rsp
   0x0000555555555283 <+89>:    pop    %rbx
   0x0000555555555284 <+90>:    pop    %rbp
   0x0000555555555285 <+91>:    retq
   0x0000555555555286 <+92>:    callq  0x555555554ed8
End of assembler dump.
```

A. Fill the C statement in the blank, based on the corresponding assembly codes. [5pts]

B. What is the solution if the first number of the solution is “1”? [5pts]

[5pts] For the following 4 cases, explain possible errors or risks. Both p1 and p2 have references to variable x (and y if y is defined).

```
int x;
p1() {}
```

```
p1() {}
```

```
int x;
p1() {}
```

```
int x;
p2() {}
```

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

9. [5pts] The following assembly code is generated from the C file, and it is not yet linked to the final program.

```
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

```
0000000000000000 <main>:
   0:  48 83 ec 08          sub    $0x8,%rsp
   4:  be 02 00 00 00       mov    $0x2,%esi
   9:  bf 00 00 00 00       mov    $0x0,%edi      # %edi = &array
                 a: R_X86_64_32 array            # Relocation entry

   e:  e8 00 00 00 00       callq  13 <main+0x13> # sum()
                 f: R_X86_64_PC32 sum-0x4        # Relocation entry
  13:  48 83 c4 08          add    $0x8,%rsp
  17:  c3                   retq
```

- [2pts] For the first relocation entry (bf 00 00 00 00), what information is necessary during the linking process?

- [3pts] Suppose the starting address of main is 0x4003b0 and the starting address of sum is 0x4003e8 in the final relocated binary. How should the second relocation entry be completed? (e8 00 00 00 00)

10. [7pts] Solve the following problems with the given C code and assembly code

C code

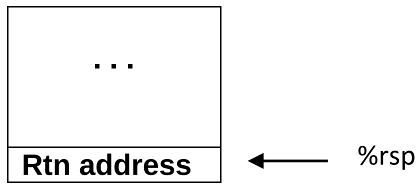
```
long mul(long *p,long val)
{
        long x=*p;
        long y=x*val;
        *p=y;
        return x;
}
long call_mul(long x)
{
        long v1=10;
        long v2=mul(&v1,300);
        return x+v2;
}
```

Assembly

```
mul:
        movq (%rdi), %rax
        imul   %rax,  %rsi
        movq %rsi, (%rdi)
        ret
call_mul:
        subq $16, %rsp
        movq $1000, 8(%rsp)
        [                    ]
        [                    ]
        call mul  ←
        addq 8(%rsp), %rax
        addq $16, %rsp
        ret
```

A. [3pts] Fill the assembly code in the blank(Assembly) [3pts]

B. [4pts] Assume that the program starts with the call_mul function. Draw the stack frame when the program **finishes executing "`call mul`" instructions.**



Initial Stack Structure

11. [13pts] Answer the questions related to the following C and assembly codes.

```
/* Get string from stdin */
char *gets(char *dest)
{
   int c = getchar();
   char *p = dest;
   while (c != EOF && c != '\n') {
      *p++ = c;
      c = getchar();
   }
   *p = '\0';
   return dest;
}
```

```
void call_echo()
{
        echo();
}
void echo()
{
         char buf[8];
         gets(buf);
         puts(buf);
}
```

```
00000000004006cf <echo>:
  4006cf:  48 83 ec 18          sub    $0x18,%rsp
  4006d3:  48 89 e7             mov    %rsp,%rdi
  4006d6:  e8 a5 ff ff ff       callq  400680 <gets>
  4006db:  48 89 e7             mov    %rsp,%rdi
  4006de:  e8 3d fe ff ff       callq  400520 <puts@plt>
  4006e3:  48 83 c4 18          add    $0x18,%rsp
  4006e7:  c3                   retq
00000000004006e8 <call_echo>:
  4006e8:  48 83 ec 08          sub    $0x8,%rsp
  4006ec:  b8 00 00 00 00       mov    $0x0,%eax
  4006f1:  e8 d9 ff ff ff       callq  4006cf <echo>
  4006f6:  48 83 c4 08          add    $0x8,%rsp
  4006fa:  c3                   retq
```

A. [2pts] Find an input for “gets” to change the return address set by “call <echo>” instruction in the stack to 0x0 (8B).

B. [2pts] Describe how an attacker can execute a short sequence of instructions in the example by exploiting the vulnerability.

C. [2pts] Explain how the stack address randomization can thwart the attack?

D. [3pts] Explain how return-oriented programming (ROP) works and how it can allow the attacker to bypass the non-executable permission setting on the stack and to execute a sequence of instructions the attacker wants to run?

E. [4pts] Fill the blacks in the following assembly code which adds the stack canary protection.

```
00000000004006cf <echo>:
  40072f:  sub    $0x18,%rsp
  400733:  mov    %fs:0x28,%rax
  40073c:  mov    %rax,0x8(%rsp)
  400741:  xor    %eax,%eax
  400743:  mov    %rsp,%rdi
  400746:  callq  4006e0 <gets>
  40074b:  mov    %rsp,%rdi
  40074e:  callq  400570 <puts@plt>
  400753:  _______________________
  400758:  _______________________
  400761:  je     400768 <echo+0x39>
  400763:  callq  400580 <__stack_chk_fail@plt>
  400768:  add    $0x18,%rsp
  40076c:  retq
```