Instructions:

1. Read through the entire examination first and determine the order in which you should answer them.
2. The exam will be graded out of 47.
3. Turn off all electronic media and store them under your desk.
4. If you are asked to author code, it must be implemented using C++. Unless we specifically ask for syntactically correct C++ code, we may ignore one or two small syntactical errors such as a missing semi colon.
5. You may use `cout` and `endl` instead of `std::cout` and `std::endl`.
6. At all times, you can declare and define helper functions to help simplify or clarify the solution you create.
7. If you need more room for one question, you can continue your solution below your solution to Question 11, but indicate that the solution continues there.
8. You may ask only one question during the examination: "May I go to the washroom?"
9. Asking any other question will result in a deduction of 5 marks from the exam grade.
10. If you think a question is ambiguous, write down your assumptions and continue.
11. Do not leave during first hour or after there are only 15 minutes left.
12. Do not stand up until all exams have been picked up.
13. If a question only asks for an answer, you do not have to show your work to get full marks; however, if your answer is wrong and no rough work is presented to show your steps, no part marks will be awarded.

1. (4 points) A *saturation function* limits a given input value x to a specified range $[A, B]$ where $A < 0$ and $B > 0$. Additional, the function defines a **deadzone**: if the absolute value of x is smaller than a threshold value d, the function returns 0.

   The behavior of the saturation function is specified as follows:

$$\text{sat}(x) = \begin{cases} B & x > B \\ x & d \leq x \leq B \\ 0 & -d < x < d \\ x & A \leq x \leq -d \\ A & x < A \end{cases}$$

   Your function must first make the following assertions:

   1. d is positive.
   2. B is greater than or equal to d.
   3. A is less than or equal to $-d$.

   Next, your function will implement this saturation function with as few conditional statements as possible.

```
double sat( double x, double d, double A, double B );

double sat( double x, double d, double A, double B ) {
```

2. (4 points) Write a function that returns the largest integer k such that such that $2^k \leq n$. Your code must work for all unsigned integers greater than 1. For example, if the argument is any value between 16 and 31, the returned value will be 4 because $2^4 = 16$ and $16 \leq 16$ and $16 \leq 31$, but $2^5 = 32$, and $32 > 31$. You may not call any standard library functions. One mark is awarded for returning the correct answer when the arguments greater than 2^{31}.

```
unsigned int max_pow_2_less_than_or_equal( unsigned int n ) {
    assert( n > 1 );
```

3. (7 points) Write a program that prints the prime factorization of a number n. If a prime factor appears more than once, it should be raised to its power in the output. For example, if the value of n is 2024, the output will be `2024 = 2^3 11 23` where $2 < 11 < 23$. You should not print a trailing space.

   Your solution need not be efficient in any way, but it should be a correctly formed program in all ways. If $n = 0$ or $n = 1$, just print `0 = 0` and `1 = 1`, respectively, as these numbers do not have prime factorizations. You do not have to include `std::` in front of `cout` or `endl`.

   Note, while not necessary, if you prefer, you can assume that there exists a function `bool is_prime( unsigned int n );` that returns `true` if `n` is prime, and `false` otherwise.

```
#include <iostream>

int main();

int main() {
    unsigned int n;
    cin >> n;        // std:: implied for exam purposes
```

4. (3 points) Write a function that takes an array and its capacity as arguments and two double-precision floating-point numbers a and b. The function should:

   1. Asserts that the capacity of the array is greater than or equal to two.
   2. Assigns the first entry of the array to a, and the last to b.
   3. Initialize the remaining elements of the array with equally spaced values between a and b.

   For example, if the capacity of the array was 5 and the arguments were 3.0 and 17.0, then the array would end up being filled with the values:

   ```
   3.0, 6.5, 10.0, 13.5, 17.0
   ```

   You will notice the spacing is 3.5. How do you get 3.5 from the values a, b and the capacity of five?

   Note that if a equals b, all the entries in the array should be the same value. If $a > b$, then the values will be decreasing linearly.

   ```
   void linspace( double a, double b, double array[], unsigned int cap );

   void linspace( double a, double b, double array[], unsigned int cap ) {
   ```

5. (6 points) Implement the function `void peaks( int n )` that draws a sequence of n peaks seen from the side, each of height n, again, when seen from the side. The expected output is shown below.

   In this question, you can use `cout` and `endl` without prefixing them by the `std::` namespace, and remember that you can declare and define helper functions, if you wish.

   When $n \leq 0$, the output is nothing.
   When $n = 1$, the output is:

```
\                           // You can start your solution here...
/                           void peaks( int n ) {
```

   When $n = 2$, the output is:

```
\
 \
 /
/
\
 \
 /
/
```

   When $n = 3$, the output is:

```
\
 \
  \
  /
 /
/
\
 \
  \
  /
 /
/
\
 \
  \
  /
 /
/
```

   The pattern is similar
   for larger values of n.

6. (5 points) Draw the call stack the first time the program gets to the comment that says "Draw the call stack here..." Your call stack need only identify which sections are for which function call, what are the parameters and local variables on the call stack, together with their values.

   Note, you don't have to write down addresses, and you don't have to show the intermediate call stacks, so for example, if a call to f does not result in a call to g, just do the computation and use the return value.

```
int main();
int f( int n );
int g( int k );

int main() {
    int sum{};

    for ( int k{ 0 }; k < 10; ++k ) {
        sum += f( k*k );
    }

    return 0;
}

int f( int n ) {
    if ( n > 10 ) {
        int tmp{};

        if ( n%2 == 0 ) {
            tmp = g( n );
        } else {
            tmp = g( n + 1 );
        }

        if ( tmp > 20 ) {
            tmp = 20;
        }

        return tmp;
    } else {
        if ( n%2 == 0 ) {
            return n/2;
        } else {
            return 3*n + 1;
        }
    }
}

int g( int k ) {
    int m{ 42 };

    while ( m%k > 0 ) {
        m -= k;
    }

    // Draw the call stack here...

    return m;
}
```

7. (4 points) Demonstrate that the sum of two small negative numbers in two's complement still produces a negative number by:

    1. determining the negative decimal integers these two binary numbers m and n represent and the sum of these two negative numbers,
    2. adding these two binary numbers together using binary addition as explained in class,
    3. taking the absolute value of this two's complement sum, and then determining the decimal integer which this represents, showing that the sum does represent value you got in Item 1.

    ```
    short m{ 0b1111111111011011 };
    short n{ 0b1111111111100101 };
    short sum{ m + n };
    ```

8. (5 points) Write a function that swaps two bits within an unsigned integer. Recall that Bit 0 is the least-significant bit (LSB), and Bit 31 is the most significant bit (MSB). The function should assert that both bit positions m and n are between 0 and 31, inclusive. Note that if the bits are already equal, there is nothing to do, otherwise, each has to be flipped.

```
void swap_bits( unsigned int &num, unsigned int m, unsigned int n ) {
```

9. (4 points) Write a function that takes an array `endpts` of floating-point numbers as input. You are guaranteed that the capacity `cap` of the array is an even number. You are also guaranteed that the floating-point numbers are increasing in the array; for example,

```
{-10.3, -5.2, -3.0, 2.5, 5.7, 6.1, 18.8, 23.0}
```

This array of eight floating-point numbers represents four intervals

$$[-10.3, -5.2], [-3.0, 2.5], [5.7, 6.1], [18.8, 23.0].$$

That is, all numbers between -10.3 and -5.2 inclusive, all numbers between -3.0 and 2.5 inclusive, and so on. An array of $2n$ floating-point numbers would represent n closed intervals.

You are asked to write a function to determine if the argument assigned to `x` is in one of these intervals. If it is found, you will indicate the interval by a value between `0` and `cap/2 - 1`. If it is not in one of these intervals, you will return `cap/2`.

For example, if the parameter `x` is assigned `-9.3` with the above array, the returned value would be `0`; if it was `6.1`, `2` would be returned; and if it was `-15.2`, `-4.0` or `6.103`, in all of these, `4` would be returned.

```
unsigned int is_in( double x, double endpts[], int cap );

unsigned int is_in( double x, double endpts[], int cap ){
    // The array has an even capacity
    assert( cap%2 == 0 );
```

10. (4 points) Write one function that takes an array of integers, and returns the number of entries in the longest consecutive sequence of positive integers (integers **greater** than zero). For example, given these arrays:

```
{  1,    -10,  -6,  [3,   4,]  -3,  -8,   0,   8,  -7 }
{  8,     -9,   8,   0,   9,   -1,  [6,  10,   5,   9]}
{ -8,     -5,  -8,  -1,   0,   -8,  -1,  -8,  [4,   4]}
{  4,     -3,   0,  [9,   6,    2,   7,   3,   7,] -9 }
{  1,      8,  -8,   3,   7,   -1,  [1,   7,   2,]  0 }
{  0, [0xece,]  0,   0,   0, [150,]  0,   0,   0,   0 }
{ -2,     -9,  -7,  -7,  -5,   -9,   0,  -8,  -5,  -1 }
```

The correct returned values are 2, 4, 2, 6, 3, 1 and 0, respectively, with corresponding entries that satisfy these. You can use `std::max( m, n )` and `std::min( m, n )` that returns the maximum and minimum of the arguments `m` and `n`, respectively.

Note, as you step through the array, you will have to count how long the current sequence of positive integers is and also track the current maximum length of a sequence of positive integers.

```
unsigned int count_pos( int array[], unsigned int cap ) {
```

11. (1 point) What is the output when this program is compiled and executed?

```
#include <iostream>

int main();

int main() {
    std::cout << "\"Question 11!\"" << std::endl;
    return 0;
}
```