**Question 1**

Instead of computing as simply multiplied by itself many times, we can rather approach it using this idea:

*We split the computation by using a base-2 binary representation of the exponent, .*

If we have , for example, we express the component in base-2 binary as .

We know that in all cases, the exponent will always have, at worst case, ) + 1 many digits.

We can verify this in our case: Our exponent is , which is 3 digits.

For cases which do not yield an integer, we take its floor

And so,

We can see via this representation, that can be computed as a multiplication of each individual binary spot, in our case: .

Since we don’t need to compute the base, which is just , we need to perform only multiplications, again taking its floor for non-integer results.

i.e. =

To calculate these multiplications, we notice each element that we compute is the square of the previous element.

And so, by doing a maximum of multiplications (taking the floor for non-integer results), we can compute our answer by multiplying the relevant powers.

In our example, .

The complexity of the program is ): The computation of the powers is at most and the multiplication as well.

For the algorithm, we divide the problems into subproblems of size and call them recursively. Two cases are required; for if is even, or if is odd (we take its floor)

See pseudocode on next page.

**Pseudocode**

int question1(int , int )

if () // *base case*

return

else if () // *n is even*

return question1() \* question1()

else // *n is odd*

return \* question1() \* question1()

end

The reason that the odd case returns *M \* question1(M, n/2) \* question1(M, n/2)*, is because all odd powers of will have a 1 as their rightmost bit, representing , so the result will always need to be multiplied by . Even cases of , will not.