1. Algorithm A

Let A be an algorithm to solve the following problem:

Given: number n, **Question:** Is there a number k such that $n = 3k$?

Input and Output of A

The most straightforward answer is that the given input to the algorithm A is any number n, and the ideal output would be to determine whether there exists an integer k such that $n = 3k$. The output could be a simple "yes" or "no," or it could return the specific value of k if it exists. However, algorithm A can technically solve this condition without returning a helpful or meaningful output.

For this to be an algorithm, it still qualifies as one even if it does not return the solution directly. An algorithm is defined as a set of written instructions that halts on all inputs and produces an output. The output doesn't necessarily have to be "yes" or "no"; it can be any defined response, as long as the algorithm eventually stops and returns a result. For example, the algorithm could return "Chicken" if there exists a k such that $n = 3k$, and "Fish" if no such k exists. The key requirement is that the algorithm produces a consistent and defined output for every possible input n.

However, I also consider an algorithm A, that solves this but does not output the information in a meaningful way. For example, let's say the outputs were 1 for yes and 1 for no. Would this be an algorithm?

2. Stupid C-function

Consider the following C-function:

```
Bool Stupid(int i){
    int k=1;
    while(1)
        if k++ ==i  break;
    ret 1;
}
```

Is it an algorithm? Briefly state your reasoning. How much memory does k use?

This is an algorithm because it halts on all inputs, which is a key requirement by definition. The algorithm increments k until it matches i or until an integer overflow occurs. In cases where i is greater than or equal to k, such as when $i = 1$, the loop will break immediately as the condition $k + + == i$ is satisfied on the first iteration, returning 1.

In cases where i is less than k (for example, when $i = 0$), the loop will not break immediately. Instead, k will continue to increment until it reaches the maximum value an integer can hold. Once k surpasses the maximum integer, it will overflow, causing k to wrap around to a negative value or zero. Eventually, k will match i, and the loop will break, returning 1. C's handling of integer overflow ensures that the algorithm will terminate even in cases where i is initially not equal to k.

In terms of memory, k uses the space allocated for an integer variable, which is typically 4 bytes on most systems. The memory constraints are tied to the maximum value that an integer can hold, beyond which the integer overflow occurs, leading to the wrapping behavior.

3. Algorithm Foo

Mr. X describes his algorithm called Foo using pseudocode as follows:

input: integer i Let k be an integer variable initialized to be 1;

```
while(1)
    if k++ ==i  break;
ret  yes;
```

Is Foo an algorithm? Briefly state your reasoning. How much memory does k use?

In this case, Foo is not an algorithm because it does not guarantee to halt on all inputs, which is a key requirement by definition. Since this is pseudocode and not a specific programming language like C, we cannot assume the same behavior regarding integer overflow or the order of operations in the condition $k++==i$. For instance, if $i=0$, the variable k will increment indefinitely, never satisfying the condition to break the loop, causing the algorithm to run forever. Similarly, if $i=-1$, k will continuously increase without ever reaching the value of i, leading to an infinite loop.

The space complexity of the algorithm is a constant amount of memory, independent of how large k becomes. The definition of space complexity focuses on how memory usage grows with the input size n, not on the actual value of variables like k. Therefore, even if k were to grow indefinitely, the space complexity remains $O(1)$ because the memory required to store k is fixed and does not scale with the input size.

4. Memory Size for Student

Memory size is measured in bits. Assume that we have 64 students in our class. Consider the following pseudocode segment:

```
for each STUDENT in our class
    if STUDENT is sleepy, wake him/her up;
```

What is the minimal memory size that `STUDENT` takes? Briefly state your reasoning.

We are simply identifying and tracking each student, and need the minimal memory required to distinguish between 64 unique students. We would only need 6 bits. This is because, where $n = 64$, the minimal number of bits needed to uniquely represent or distinguish between n elements is given by $\log_2(n)$. In this case, $\log_2(64) = 6$.

The reason for using $\log_2(n)$ instead of n is that we are not storing any additional information about each student (such as whether they are sleepy), but rather we only need to process or identify each student uniquely.

5. Program P

Let Q be an arbitrary C-program without input. Consider the following "code" P:

input: Q and integer i

```
if Q runs forever
    return 2i;
else
    return i;
```

Is P an algorithm? Briefly state your reasoning.

Yes, P is an algorithm. Algorithms, as defined, are Turing machines that halt on all inputs. In this case, although the logic of P allows for the possibility of returning $2i$, C's memory constraints ensure that Q will always stop, meaning that P will ultimately halt and return i. Therefore, P satisfies the definition of an algorithm because it halts and produces a defined output for any input. Even if Q is a C program designed to run indefinitely (e.g., a `while` loop), C has finite memory limits. Eventually, Q will run out of memory and stop, which guarantees that P will halt regardless of how Q behaves.

6. Speedup in Multiplication

1. Defining $T_1(n)$

In traditional multiplication, two numbers with n digits are multiplied, followed by adding up the partial products. Each digit in the first number is multiplied by every digit in the second number. Thus, the number of multiplication operations is $n \times n = n^2$. The number of additions is represented by:

$$0 + 1 + 2 + \ldots + (n-1)$$

This is the sum of an arithmetic series:

$$\text{Sum of additions} = \frac{(n-1) \cdot n}{2}$$

Therefore, the total time complexity for traditional multiplication is:

$$T_1(n) = n^2 + \frac{(n-1) \cdot n}{2}$$

2. Defining $T_2(n)$

For faster multiplication, we group the digits into pairs and then multiply these groups. First, we divide the digits into groups. For n digits, there are $n/2$ groups of 2 digits, so the cost of grouping is proportional to n, which results in n cuts. Now, instead of multiplying individual digits, we multiply groups of digits. Since there are $n/2$ groups in each number, the number of multiplications is:

$$\left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right) = \frac{n^2}{4}$$

The addition process follows the same principle as the traditional method, but for fewer terms. The number of additions is:

$$0 + 1 + 2 + \ldots + \left(\frac{n}{2} - 1\right)$$

This sum simplifies to:

$$\frac{\left(\frac{n}{2} - 1\right) \cdot \frac{n}{2}}{2} = \frac{\left(\frac{n}{2}\right) \cdot \left(\frac{n}{2} - 1\right)}{2}$$

Thus, the total time complexity for faster multiplication is:

$$T_2(n) = \frac{n^2}{4} + \frac{\left(\frac{n}{2}\right) \cdot \left(\frac{n}{2} - 1\right)}{2} + n$$

Thus we have:

$$T_1(n) = n^2 + \frac{(n-1) \cdot n}{2}$$

$$T_2(n) = \frac{n^2}{4} + \frac{\left(\frac{n}{2}\right) \cdot \left(\frac{n}{2} - 1\right)}{2} + n$$

Now, we compute the limit of the ratio:

$$\lim_{n \to \infty} \frac{T_1(n)}{T_2(n)} = \lim_{n \to \infty} \frac{n^2 + \frac{(n-1) \cdot n}{2}}{\frac{n^2}{4} + \frac{\left(\frac{n}{2}\right) \cdot \left(\frac{n}{2} - 1\right)}{2} + n}$$

For large n, the n^2 terms dominate in both $T_1(n)$ and $T_2(n)$. Thus, the asymptotic speedup is:

$$\lim_{n \to \infty} \frac{T_1(n)}{T_2(n)} = \frac{n^2}{\frac{n^2}{4}} = 4$$

Thus, the faster multiplication method using grouping is approximately 4 times faster than the traditional method for large n, concluding that by applying the principles of Blum's theorem, we get:

$$\lim_{n \to \infty} \frac{T_1(n)}{T_2(n)} \approx 4$$