1. Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort? How might one rewrite the merge sort procedure to make it even faster on small inputs?

We wish to find n such that for $n > n_0$, the following inequality holds:

$$8n^2 < 64n \lg n$$

Defining $f(n) = 64n \lg n - 8n^2$, this is equivalent to finding values of n for which $f(n) > 0$. Newton's method is a useful numerical technique for calculating the roots of transcendental functions like this one. Starting with an initial guess x_0, the subsequent (hopefully better) guess is given by:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

In this case, $f(n) = 64n \lg n - 8n^2$ and $f'(n) = 64 \lg n + 64 - 16n$. The following Python script calculates a numerical approximation for the roots of this function using Newton's method:

```python
import math

def newton(F, dF, x):
    """ Return next iterative approximation. """
    return x - F(x)/dF(x)

def F(x):
    """ Value of 64x*lg(x) - 8x^2 """
    return 64*x*math.log(x) - 8*math.pow(x,2)

def dF(x):
    """ Value of derivative of 64x*lg(x) - 8x^2. """
    return 64*math.log(x) + 64 - 16*x

def find_root(F, dF, initial_guess, tolerance):
    """ Find root using Newton's method to specified tolerance, using initial guess. """
    def guess(x0):
        x1 = newton(F, dF, x0)
        diff = abs(x0-x1)
        return (x1, diff)

    # Iterate until we obtain a result within tolerance.
    (x1, diff) = guess(initial_guess)
    while (diff > tolerance):
```

```
        (x1, diff) = guess(x1)
    return x1

# Find the root
print find_root(F, dF, 20, 0.01)
```

The script returns 26.0934 as an approximate root, which indeed is valid:

$$8 * 26 * 26 = 5,408 < 5,421.47 \approx 64 * 26 * \lg(26)$$

$$8 * 27 * 27 = 5,832 > 5,695.21 \approx 64 * 27 * \lg(27)$$

The answer to the question is that for inputs $n < 27$, insertion sort will run faster than merge sort.

[Working]

> 2. What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

We wish to find n_0 such that for $n > n_0$ the following inequality holds:

$$100n^2 < 2^n$$

Defining $f(n) = 2^n - 100n^2$, this is equivalent to finding values of n for which $f(n) > 0$. Newton's method is a useful numerical technique for calculating the roots of transcendental functions like this one. Starting with an initial guess x_0, the subsequent (hopefully better) guess is given by:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

In this case, $f(n) = 2^n - 100n^2$ and $f'(n) = (\ln 2) \cdot 2^n - 200n$. The following Python script calculates a numerical approximation for the roots of this function using Newton's method:

```
import math

def newton(F, dF, x):
    """ Return next iterative approximation. """
    return x - F(x)/dF(x)

def F(x):
    """ Value of 2^x - 100x^2. """
    return math.pow(2,x) - 100*math.pow(x,2)

```

```python
def dF(x):
    """ Value of derivative of 2^x - 100x^2. """
    return math.log(2)*math.pow(2,x) - 200*x

def find_root(F, dF, initial_guess, tolerance):
    """ Find root using Newton's method. """
    def guess(x0):
        x1 = newton(F, dF, x0)
        diff = abs(x0-x1)
        return (x1,diff)

    # Iterate until we obtain a result within tolerance
    (x1,diff) = guess(initial_guess)
    while (diff > tolerance):
        (x1, diff) = guess(x1)
    return x1

# Find the root with initial guess of 20
print find_root(F, dF, 20, 0.01)
```

The script returns 14.3247 as an approximate root, which indeed is valid:

$$100 * 14 * 14 = 19,600 > 16,384 = 2^{14}$$

$$100 * 15 * 15 = 22,500 < 32,768 = 2^{15}$$

The answer to the question is that for $n = 15$ and higher the $T(n) = 100n^2$ procedure will outperform the $T(n) = 2^n$ procedure.

The problem illustrates that an exponential procedure grows much more rapidly than a quadratic function, and that even for $n = 15$ a very bad quadratic procedure will greatly outperform an exponential one.