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_0 such that for $n > n_0$, the following inequality holds:

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

If we let $f(n) = 64n \lg n - 8n^2$, we can use Newton's method to find the roots of the function and in turn infer values of n where $f(n) > 0$. For Newton's method, if the initial guess is given by x_0, the subsequent (and hopefully better) guess is given by:

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

In this case, $f(n) = 64n \lg n - 8n^2$ so that $f'(n) = 64 \lg n + 64 - 16n$.

The following Python script calculates a numerical approximation for the roots of $f(n)$ 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 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):
```

```
25          (x1, diff) = guess(x1)
26      return x1
27
28 # Find the root
29 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 $n <= 26$, insertion sort will run faster than merge sort.

Merge sort recursively splits an input array of size n into two subarrays of size (roughly) $n/2$, and then merges these two subarrays back together in such a way as to ensure that the combined array of size n is in sorted order.

A naive implementation of merge sort only terminates the recursion when the input array is of size $n = 1$. We can improve the performance of merge sort by having it instead terminate the recursion when the size of the input array is $n <= 26$. Smaller input or subarrays where $n <= 26$ could instead be sorted using the $T(n) = 8n^2$ insertion sort procedure.

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:

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

If we let $f(n) = 2^n - 100n^2$, we can use Newton's method to find the roots of the function and in turn infer values of n where $f(n) > 0$. For Newton's method, if the initial guess is given by x_0, the subsequent (and hopefully better) guess is given by:

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

In this case, $f(n) = 2^n - 100n^2$ so that $f'(n) = (\ln 2) \cdot 2^n - 200n$.

The following Python script calculates a numerical approximation for the roots of $f(n)$ 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)

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.