Question 1:

```
int Classify(int a) {
    switch (a) {
        case 0:
            return 1;
        case 1:
            return 2;
        default:
            return 3;
    }
}
```

Question 2:

> a=1: 1 comparison (matches case 1 directly)
>
> a=2: 2 comparisons (checks case 0, then case 1, then goes to default)
>
> a=3: 2 comparisons (same as a = 2) Average Time:
>
> > Assuming a is uniformly distributed over {1, 2, 3}: (1×1+1×2+1×2)/3 = 5/3 ≈1.67

Question 3:

Proof:

> 1. **Given:**
> $f(x) = O(g(x)) \Rightarrow \exists M_1, x_1 > 0$ such that $|\ f(x)\ | \leq M_1\ |\ g(x)\ |, \forall x \geq x_1$
> $g(x) = O(h(x)) \Rightarrow \exists M_2, x_2 > 0$ such that $|\ g(x)\ | \leq M_2\ |\ h(x)\ |, \forall x \geq x_2$
> 2. **Let $x_0 = max(x_1, x_2)$, then for all $x \geq x_0$:**
> $|\ f(x)\ | \leq M_1\ |\ g(x)\ | \leq M_1 \times M_2\ |\ h(x)\ |$
>   **Let $M = M_1 \times M_2$, so:** $|\ f(x)\ | \leq M\ |\ h(x)\ |, \forall x \geq x_0$
>   **Thus:** f(x)=O(h(x))
> 3. **Proof completed.**

Question 4:

Proof:

> 1. Using the logarithm base change formula: $log_a n = \frac{log_b n}{log_b a}$
> 2. Since $log_b a$ is a constant (as a,b are fixed and greater than 1), we get:
> $log_a n = O(log_b n)$
> 3. Proof completed.

Complexity

The time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm. In modern era, most of instructions takes fixed time/cycles to perform.

Because the amount of time taken and the number of elementary operations performed by algorithm are taken to be related by a constant factor, time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm.

Considered followed function in C language:

```c
int GCD(int a, int b) {
    int temp;
    while (b != 0) {
        temp = a % b;
        a = b;
        b = temp;
    }
    return a;
}
```

We can count the number of statements and elementary operations as followed:

```c
int GCDCounters[8] = {0}
int GCD(int a, int b) {
    int temp;
    GCDCounters[2] += 1;
    while (GCDCounters[2] += 1 && b != 0) {
        temp = a % b;
        GCDCounters[3] += 1;
        a = b;
        GCDCounters[4] += 1;
        b = temp;
        GCDCounters[5] += 1;
        GCDCounters[6] += 1;
    }
    GCDCounters[7] += 1;
    return a;
}
int ProfileGCD() {
    int ans = 0;
    ans += GCDCounters[1] * 1; // Allocate static variable
    ans += GCDCounters[2] * 2; // Compare and Branch
    ans += GCDCounters[3] * 2; // Modulo and assign value
    ans += GCDCounters[4] * 1; // Assign value
    ans += GCDCounters[5] * 1; // Assign value
    ans += GCDCounters[6] * 1; // Jump back to loop begin
    ans += GCDCounters[7] * 1; // Return value
    return ans;
}
```

Since an algorithm's running time may vary among different inputs of the same size, one commonly considers the worst-case time complexity, which is the maximum amount of time required for inputs of a given size. Less common, and usually specified explicitly, is the average-case complexity, which is the average of the time taken on inputs of a given size.

Considered followed function in C language:

```c
int Classify(int a) {
    if (a == 0) return 1;
    else if (a == 1) return 2;
    else return 3;
}
```

It's trivial that the worst-case of the above function is the parameter larger than 0. In this case, we have 5 operations to be performed (2 comparisons, 2 branched, and 1 return).

Assume our input is in a discrete uniform distribution through 0 to 5, the average time complexity should be (3 × 1 + 5 × 1 + 5 × 4)/6 = 4

Question: Could you design a new approach to improve the time complexity of average case?

Question: What the average case would be if the input is through 1 to 3?

In both cases, the time complexity is generally expressed as a function of the size of the input. Since this function is generally difficult to compute exactly, and the running time for small inputs is usually not consequential, one commonly focuses on the behavior of the complexity when the input size increases – that is, the asymptotic behavior of the complexity. Therefore, the time complexity is commonly expressed using big O notation.

Big O notation

Definition: f(x) = O(g(x)) if there exists M,x0 R+ such that |f(x)| ≤ M|g(x)| for all x ≥ x0

Question: f(x),g(x),h(x) are 3 functions. Try to proof if f(x) = O(g(x)) and g(x) = O(h(x)), then f(x) = O(h(x)).

By the proposition of above question, we know that big O notions are kind of sets. For any 2 functions f(x) and g(x), if f(x) = O(g(x)) then O(f(x)) is the subset of O(g(x)). Thus, we can write either f(x) = O(g(x)) or f(x) ∈ O(g(x)).

Type of different complexities (in increasing order)

- constant time: $O(1)$
- log-logarithmic time: $O(\log \log n)$
- logarithmic time: $O(\log n)$
- polylogarithmic time: $O(\log n)^c$
- linear time: $O(n)$
- linearithmic time: $O(n \log n)$
- polynomial time: $O(n)^c$
- exponential time: $O(c^n)$
- factorial time: $O(n!)$, $O(n^n)$, $O(2^{n \log n})$

Question: ∀a,b that 1 < a,b ∈ N, proof $log_a n$ = $O(log_b n)$.