Chapter 16 Developing Efficiency Algorithms

1.

$$O(\frac{(n^2+1)^2}{n}) = O(n^3), \quad O(\frac{(n^2+\log^2 n)^2}{n}) = O(n^3), \quad O(n^3+100n^2+n) = O(n^3),$$
$$O(2^n+100n^2+45n) = O(2^n), \quad O(n2^n+n^2 2^n) = O(n^2 2^n)$$

2.

$$500, \quad 44\log n, \quad 3n, \quad 10n\log n, \quad 2n^2, \quad \frac{5n^3}{4032}, \quad \frac{2^n}{45}$$

3.

(A)
5

(B)
1

(C)
The ceiling of $\log_2 n$ times

(D)
The ceiling of $\log_3(n/15)$ times

4. if n is 10: (a) 10 (b) 10^2 (c) 10^3 (d) 10*10^2
if n is 20: (a) 20 (b) 20^2 (c) 20^3 (d) 20*20^2

Using Big-O notation:
O(n), O(n^2), O(n^3), O(n^2)

5. mA: O(n). mB: O(n^2). mC: O(n). mD: O(n)

6 Adding two matrices: O(n*m). Multiplying two matrices: O(nmk)

7. The algorithm can be designed as follows: Maintain two variables, max and count. max stores the current max number, and count stores its occurrences. Initially, assign the first number to max and 1 to count. Compare each subsequent number with max. If the number is greater than max, assign it to max and reset count to 1. If the number is equal to max, increment count by 1. Since each element in the array is examined only once, the complexity of the algorithm is O(n).

8. The algorithm can be designed as follows: For each element in the input array, store it to a new array if it is new. If the

number is already in the array, ignore it. The time for checking whether an element is already in the new array is O(n), so the complexity of the algorithm is O(n^2).

9. This is similar to bubble sort. Whenever a swap is made, it goes back to the beginning of the loop. In the worst case, there will be O(n^2) of swaps. For each swap, O(n) number of comparisons may be made in the worst case. So, the total is O(n^3) in the worst case.

10.

```
result = a
i = 2

while (i <= n):
    result = result * result
    i *= 2

for j in range(i // 2 + 1, n + 1):
    result = result * a
```

Assume that $2^{k-1} \le n < 2^k$. The while loop is executed k-1 times. The for loop is executed at most $2^k - 2^{k-1} = 2^{k-1}$ times. So, the total complexity is O(n).

Consider another implementation:

```
def f(a, n):
    if (n == 1):
        return a
    else:
        temp = f(a, n / 2)
        if (n % 2 == 0):
            return temp * temp
        else:
            return a * temp * temp
```

This implementation results in O(logn) complexity.

11. See the definition and example in the text.
12. The recursive Fibonacci algorithm is inefficient, because the subproblems in the recursive Fibonacci algorithm overlaps, which causes redundant work. The non-recursive Fibonacci algorithm is dynamic algorithm that avoids redundant work.
13. See the definition and example in the text.
14. Yes. Finding the minimum in the first half and the second half of the list and return the minimum of these two. So, the time complexity is O(n) = 2 * O(n/2) + O(1) = O(n).
15. See the definition and example in the text.
16. O(n!)