William Romine

Analysis of Algorithms

Dr. Lewis

Jan. 31, 2024

Assignment 02

1. The value returned by this algorithm, expressed as function of n, would be (n(n+1))/2. This is because the algorithm consists of three nested loops. Each one leads to the innermost loop being incremented by 1 through the equation, 1 + 2 + 3 + 4 + ... + n. Thus, the value of r after all iterations of the loops is: r =(n(n+1))/2.

   For the O(n) of this algorithm, the dominant term in the expression for r is n^2, and

   constants are ignored in big O notation. Hence, the big O complexity of the algorithm is O(n^2).

2. We need to find constants c and no such that for all n ≥ no, f(n) ≤ c (n^3).

   Given f(n) = n^2 + 3n^3, we can simplify and find:

   f(n) = n^3 (3 + 1/n)

   Now, consider the terms 3 + 1/n. As n grows larger, the 1/n term becomes negligible, and we can choose c = 4 and no = 1 such that for all n ≥ 1:

   f(n) ≤ 4n^3

   Thus, f(n) is in O (n^3).

   We need to find constants c' and n'o such that for all n ≥ n'o, f(n) ≥ c' (n^3).
   Given f(n) = n^2 + 3n^3, we can simplify and find:
   f(n) = n^3 (3n^-1 + 1)
   Now, consider the term 3n^-1 + 1. As n grows larger, the 3n^-1 term becomes negligible, and we can choose c' = 2 and n'o = 1 such that for all n ≥ 1
   f (n) ≥ 2n^3
   So, f(n) is in Ω (n^3).

Therefore, f(n) is both in O (n^3) and Ω (n^3). Hence, we conclude that f(n) is in Θ(n3).

3. For the function f(n) = 2n + 1, you correctly expressed it as 2 x 2^n. Now, by the definition of Big O notation, 2n + 1 is O (2^n) because there exists a constant c ≥ 2 such that 2 x 2^n ≤ c x 2^n for sufficiently large n.

Now, if a is a constant, consider the function g(n) = a^n. Following a similar analysis, an+1 can be expressed as a x a^n, and for any c ≥ a, we have a x a^n ≤ c x a^n for sufficiently large n. Therefore, a^n+1 is O(a^n). If a is a constant, then a^n+1 is O(a^n). This generalizes the idea that when you have an exponent that increments by 1 (n+1), it remains within the same order of growth as the original exponent (n) when the base is a constant.

4. The worst-case time complexity of this algorithm can be expressed as O(n!), where n is the number of vertices in the graph.

   The recurrence relation for the worst-case time complexity is T(n) = n x T(n-1), where n is the number of vertices. In each recursive call, the algorithm iterates over the vertices, and the recursion is called n times. This results in a factorial growth in the number of operations.

   Hence, the worst-case time complexity of the given algorithm is O(n!).

5.

Algorithm Performance Analysis	
Data Size	Avg Run Time (ms)
5	0.0007
10	0.0005
50	0.0011
100	0.002
500	0.0122
1000	0.0268
5000	0.1632
10000	0.3504

The average run times are small, indicating efficient sorting even for larger data sets. The progression aligns with the expected O(n log (n)) behavior, as the run times increase at a rate that suggests a logarithmic growth pattern. The results also reflect the efficiency of the sorting algorithm, displaying its ability to handle larger data sets with acceptable run times. The algorithm's performance is consistent with the theoretical time complexity of the sorting function. Overall, the analysis suggests that the function performs close to the O(n log (n)) complexity.