Karim Maamari

0.

(a). In this case, we want to prove that 2 √(n) + 6 = O(√(n)). To do this, we need to find a constant c and n0 such that for all n > n0, |2 \* √(n) + 6| <= c \* |√(n) |.

Let's take c = 3 and n0 = 36. Then, for all n > n0, we have:

|2 \* √(n) + 6| <= 3 \* |√(n) |

2 \*√(n) + 6 <= 3 \*√(n)

2 \* √(n) <= 3 \* √(n) - 6

2 \* √(n) <= 3 \* √(n) - 6

√(n) <= (3 \* √(n) - 6) / 2

n <= (3 \* √(n) - 6) ^2 / 4

n <= 9 \* n - 54

-6 \* n <= -54

n >= 9

So, for all n > 36, 2 \* √(n) + 6 <= 3 \* √(n), which means that 2 \* √(n) + 6 = O(√(n))

(b).

I believe the easiest way to answer this question is using a contradiction.

To show that 10n is not O(2n), we need to demonstrate that there is no positive constant c such that 10n ≤ c \* 2n for all sufficiently large n.

Let's assume, for the sake of the contradiction, that there exists a positive constant c such that:

10 n ≤ c \* 2 n for all sufficiently large n.

We can rearrange this inequality as follows:

10 n /2 n ≤ c

5 n ≤ c

This implies that 5 n is bounded by a constant, which is not true.

Thus, we can conclude that 10n is not O (2 n).

1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | O | Ω | Theta |
| log9 n | n 0.9 |  | x |  |
| 2n 10 | 2 n | x |  |  |
| 3 3n | 3 4n |  | x |  |
| ln n | log n |  |  | x |
| log(n!) | log (n n) |  |  | x |
| (5/4) n | (4/5) n | x | x | x |
| n 2 | 4 log n | x |  |  |
| n 0.1 | (0.1) n |  | x |  |
| log log n | √ (log n) | x |  |  |
| n 1/log n | 1 |  | x |  |

2.

(a).

Base case: For n = 0, F (0) = 0, which is clearly O (30). For n = 1, F (1) = 1, which is also O (31).

Inductive step: Assume that F (n - 1) = O (3(n – 1)) and F (n - 2) = O (3(n - 2)). Then,

F(n) = F (n - 1) + F (n - 2)

<= c1 \* 3 (n - 1) + c2 \* 3(n - 2) for some constants c1 and c2

= (c1 + c2) \* 3(n - 2) \* 3

= c3 \* 3n for some constant c3.

In conclusion, we have shown that F(n) = O(3n), which means that the Fibonacci sequence is indeed O(3n).

(b).

Base case: For n = 0, we have F (2 \* 0) = F (0) = 0, which is less than or equal to

20 = 1.

Induction step: Assume that the It holds for some value of n, F(2n) = Ω(2n). We need to show that it also holds for the next value, n + 1.

We have F (2n + 2) = F (2n + 1) + F(2n). By the inductive hypothesis, F(2n) = Ω(2n). To show that F (2n + 1) = Ω(2n), we need to prove that F (2n + 1) >= c \* 2n for some constant c.

From the definition of the Fibonacci sequence, we have F(n) = F(n-1) + F(n-2). Substituting n = 2n + 1 into this equation, we get: F (2n + 1) = F(2n) + F (2n - 1).

Since F(2n) = Ω(2n), there exists a constant c1 such that F(2n) >= c1 \* 2n. In the same way, there exists a constant c2 such that F (2n - 1) >= c2 \* 2(n-1).

Thus, F (2n + 1) = F(2n) + F (2n - 1) >= c1 \* 2n + c2 \* 2(n-1) >= c2 \* 2n, where

c = minimum of (c1, c2). We can conclude that F (2n + 1) = Ω(2n).

3.

(a)

Steps:

1.Initialize a count variable to 0.

2.For each user i from 1 to n, do the following:

3. For each user j from i+1 to n, do the following:

4. If the enter time of user i is less than or equal to the leave time of user j and the leave time of user i is greater than or equal to the enter time of user j, increment the count variable by 1.

5.Return the count variable as the result.

Explanation:

The algorithm checks each pair of users to see if they overlap in time on the site. If they overlap, the count variable is incremented. By checking each pair of users, the algorithm guarantees that each pair is counted only once.

Runtime analysis:

The algorithm has a nested loop structure, so the time complexity is O(n2).

(b)

1.Create an array of events, where each event is either a user entering or leaving the site. Store the event type (enter or leave), the time of the event, and the user's index.

2.Sort the array of events in ascending order by time.

3.Initialize a count variable to 0.

4.Initialize a variable to keep track of the number of users currently on the site to 0.

5.For each event in the sorted array, do the following:

a. If the event is an enter event, increment the count of users on the site and add the user's index to a set of indices.

b. If the event is a leave event, decrement the count of users on the site and remove the user's index from the set of indices.

c. Increment the count variable by the number of users currently on the site, since each user on the site is overlapping with the current user.

6.Return the count variable as the result.

Explanation:

The algorithm sorts the events by time, so that events are processed in the order that they are happening in. As each event is processed, the count of users on the site is updated. If the event is an enter event, the user's index is added to a set of indices so that the number of overlapping users can be counted easily. If the event is a leave event, the user's index is removed from the set of indices. Finally, the count variable is incremented so we can keep track of the number of overlapping pairs.

Runtime analysis:

The time complexity of the algorithm is O (n log(n)), because the sorting step takes O (n log(n)) time, and the rest of the algorithm takes linear time.