MTech DSA Midterm Exam Answer Sheet

(a) Prove or disprove: 2^(n+1) = O(2^n). Mathematical derivation may be awarded full marks.

Answer:

Claim: The statement 2^(n+1) = O(2^n) is **true**.

Proof:

1. **Expand the left side of the equation:**

2^(n+1) = 2^n * 2^1 = 2 * 2^n

2. **We need to show that there exist positive constants *c* and n_0 such that for all $n \geq n_0$, the following inequality holds:**

2^(n+1) ≤ c * 2^n

3. **Substitute the expanded form:**

2 * 2^n ≤ c * 2^n

4. **Choose a constant *c* = 2.**

5. **With *c* = 2, the inequality becomes:**

2 * 2^n ≤ 2 * 2^n

6. **This inequality is true for all values of *n*.**

7. **Choose $n_0 = 1$ (or any positive integer).**

8. **Therefore, we have found constants *c* = 2 and $n_0 = 1$ such that for all $n \geq n_0$, 2^(n+1) ≤ c * 2^n.**

Conclusion:

Thus, 2^(n+1) = O(2^n).

(b) Prove or disprove: 2^(2n) = O(2^n). Mathematical derivation may be awarded full marks.

Answer:

Claim: The statement 2^(2n) = O(2^n) is **false**.

Proof by contradiction:

1. **Assume, for the sake of contradiction, that 2^(2n) = O(2^n).**

2. **By the definition of Big-O notation, this implies that there exist positive constants *c* and n_0 such that for all $n \geq n_0$, the following inequality holds:**

2^(2n) ≤ c * 2^n

3. **Divide both sides of the inequality by 2^n (assuming 2^n > 0, which is true for all n):**

2^(2n) / 2^n ≤ c

4. **Using the exponent rule a^m / a^n = a^(m-n), we simplify the left side:**

2^(2n - n) ≤ c

2^n ≤ c

5. **This inequality, 2^n ≤ c, implies that 2^n is bounded by a constant *c* for all $n \geq n_0$. However, this is a contradiction.**
6. **We know that 2^n grows without bound as *n* increases. Therefore, there is no constant *c* that can bound 2^n for all sufficiently large *n*.**
7. **Since our initial assumption leads to a contradiction, it must be false.**

Conclusion:

Therefore, 2^(2n) is not O(2^n).

(c) Establish relationship between f(n) and g(n) using any three asymptotic notations: Big-Oh (O), Omega (Ω) and Theta (Θ). Mathematical derivation may be awarded full marks. f(n) = log(n^2), g(n) = log(n + 5)

Answer:

Analysis:

- **f(n) = log(n^2) = 2 * log(n)**
- **g(n) = log(n + 5)**

As n grows large, log(n + 5) behaves similarly to log(n).

Relationship:

- **f(n) = O(g(n))**
    - Since 2 * log(n) grows no faster than log(n + 5), up to a constant factor.
- **f(n) = Ω(g(n))**
    - Since 2 * log(n) grows at least as fast as log(n + 5), up to a constant factor.
- **f(n) = Θ(g(n))**
    - Since f(n) = O(g(n)) and f(n) = Ω(g(n)), they have the same asymptotic growth rate.

Conclusion:

f(n) and g(n) have the same asymptotic growth rate.

(d) Establish relationship between f(n) and g(n) using any three asymptotic notations: Big-Oh (O), Omega (Ω) and Theta (Θ). Mathematical derivation may be awarded full marks. f(n) = 10, g(n) = log(10)

Answer:

Analysis:

- **f(n) = 10** (Constant)
- **g(n) = log(10)** (Constant)

Both are constant functions.

Relationship:

- **f(n) = O(g(n))**
    - A constant grows no faster than another constant.
- **f(n) = Ω(g(n))**
    - A constant grows at least as fast as another constant.
- **f(n) = Θ(g(n))**
    - Constants have the same growth rate.

Conclusion:

f(n) and g(n) have the same asymptotic growth rate. They are both constant functions.

(e) Consider the following function. What would this function return? Express the time complexity of this function in Big-Oh (O) notation. Appropriate mathematical derivation may be awarded full mark.

```
function mystery(n)
 r := 0
 for i := 1 to n - 1 do
  for j := i + 1 to n do
   for k := 1 to j do
    r := r + 1
```

Answer:

What the function returns:

The function calculates the sum of j for all valid combinations of i and j in the nested loops.

Mathematical Derivation:

The number of times r is incremented can be represented as:

$r = \Sigma$ (from i=1 to n-1) Σ (from j=i+1 to n) j

This summation is complex, but the order of growth is what we need.

Time Complexity:

- Innermost loop: O(j), which is at most O(n).
- Middle loop: O(n - i), which is at most O(n).
- Outer loop: O(n - 1), which is O(n).

Therefore, the total time complexity is O(n * n * n) = O(n^3).

Big-Oh Notation:

The time complexity of the function mystery(n) is **O(n^3)**.

Let's break down this problem and provide a detailed answer sheet.

Question 2:

(a) In grade school, you learned to multiply long numbers on a digit-by-digit basis. e.g. 127 x 211 = 127 x 1 + 127 x 10 + 127 x 200 = 26,397. Derive the time complexity of multiplying two n-digit numbers with this method as a function of n (assume constant base size). Assume that single-digit by single-digit addition or multiplication takes O(1) time.[1] Use of the given hint may be awarded full mark. (Hint: Consider polynomial representation of both the numbers to derive the time complexity).

Answer:

Polynomial Representation:

Let's represent two n-digit numbers, A and B, as polynomials:

- $A = a_0 + a_1x + a_2x^2 + ... + a<0xE2><0x82><0x99><0x88>x^(n-1)$
- $B = b_0 + b_1x + b_2x^2 + ... + b<0xE2><0x82><0x99><0x88>x^(n-1)$

Here, a_i and b_i are the digits of the numbers, and x is the base (e.g., 10 for decimal numbers).

Multiplication Process:

When we multiply A and B, we are essentially multiplying these polynomials:

$A * B = (a_0 + a_1x + ... + a<0xE2><0x82><0x99><0x88>x^(n-1)) * (b_0 + b_1x + ... + b<0xE2><0x82><0x99><0x88>x^(n-1))$

To get the product, we multiply each term of A with each term of B:

- $a_0 * (b_0 + b_1x + ... + b<0xE2><0x82><0x99><0x88>x^(n-1))$
- $a_1x * (b_0 + b_1x + ... + b<0xE2><0x82><0x99><0x88>x^(n-1))$
- ...

- a<0xE2><0x82><0x99><0x88>x^(n-1) * (b_0 + b_1x + ... + b<0xE2><0x82><0x99><0x88>x^(n-1))

Time Complexity Analysis:

1. **Multiplications:** We perform n multiplications for each term of A. Since A has n terms, we perform n * n = n^2 multiplications. Each single-digit multiplication takes O(1) time.
2. **Additions:** After multiplying, we need to add the resulting terms. The number of additions is also related to the number of terms we get after multiplication. The resulting polynomial will have terms up to x^(2n-2). The number of additions is proportional to n^2. Each single-digit addition takes O(1) time.

Overall Time Complexity:

Since both multiplications and additions take O(n^2) time, the overall time complexity of multiplying two n-digit numbers using this method is **O(n^2)**.

(b) Apply your answer in part a) and find out the absolute number of multiplications and additions to compute 10011001 x 11000001.

Answer:

Number of Digits:

- 10011001 has 8 digits (n = 8).
- 11000001 has 8 digits (n = 8).

Multiplications:

Using the grade-school method, we perform n * n multiplications:

- 8 * 8 = 64 multiplications.

Additions:

The number of additions is also related to n^2, but it's a bit trickier to calculate the exact number without performing the full multiplication. However, we know it will be proportional to n^2. In the worst case, we might have around $(n-1)^2$ additions.

- $(8 - 1)^2 = 7^2 = 49$ additions.

Absolute Number of Operations:

- **Multiplications:** 64
- **Additions:** Approximately 49 (we can get the exact number by performing the multiplication, but the question asks for an application of the time complexity, which is n^2).

Note: The exact number of additions will depend on the actual values of the digits and how many carries occur during the addition process. But for the purpose of demonstrating the application of the time complexity, we can say it's proportional to n^2.

Let's break down this problem and provide a detailed answer sheet.

Question (a): Design a dictionary data structure in which search, insertion, and deletion can all be processed in O(1) time in the worst[1] case (a suitable graphical representation is preferable). You may assume the set elements are integers drawn from non-negative integers, and initialization can take O(n) time assuming there are n elements. Write down the formula f(x) to index an item x into the dictionary.

Answer:

Data Structure: Direct Address Table (Array)

We can use a Direct Address Table, which is essentially an array.

Assumptions:

- The keys are non-negative integers.
- We know the maximum possible key value, say maxKey.

Graphical Representation:

Array (Dictionary):

| Index | Value |

| 0 | |

| 1 | |

| 2 | |

| ... | |

| maxKey| |

Formula f(x):

The formula to index an item x into the dictionary is simply:

f(x) = x

Initialization:

1. Create an array (dictionary) of size maxKey + 1.
2. Initialize all values in the array to null or a sentinel value indicating an empty slot. This takes O(maxKey) time, which is O(n) if maxKey is proportional to n.

Question (b): Prove in no more than 3 to 4 sentences that each of search, insert and delete takes O(1) time.

Answer:

1. **Search (O(1)):** To search for an element x, we directly access dictionary[x]. This is a direct array access, taking constant time.
2. **Insert (O(1)):** To insert an element x with value v, we simply set dictionary[x] = v. This is a direct array assignment, taking constant time.
3. **Delete (O(1)):** To delete an element x, we set dictionary[x] = null (or a sentinel value). This is a direct array assignment, taking constant time.

Question (c): Consider the following sequence of numbers to be inserted in the dictionary designed by you in part a). Show the graphical representation of the final dictionary appropriately after all the insertions. 1, 5, 7, 1, 9, 2, 5, 7, 3, 7, 0, 2, 2, 4

Answer:

Assuming the maximum number in the sequence is maxKey, we need an array of size maxKey + 1. In this case, maxKey = 9.

Graphical Representation:

Dictionary:

| Index | Value |

| 0 | 1 |

| 1 | 1 |

| 2 | 2 |

| 3 | 1 |

| 4 | 1 |

| 5 | 1 |

| 6 | null |

| 7 | 1 |

| 8 | null |

| 9 | 1 |

Explanation:

- We iterate through the sequence and set dictionary[x] = 1 for each number x in the sequence. We are simply marking the presence of the number.
- If a number appears multiple times, we set dictionary[x] to 1 only once.

Question (d): Consider the dictionary in part c) and perform the following search operations and specify the exact number of comparisons (consider the check for the

existence of the value corresponding to a key as count 1 in addition to the check for the values) you perform to search each of the following items. Fill in the table in your answer sheet corresponding to the items.

Items: 5, 0, 6, 8, 4

Answer:

Search Operations:

- **5:** We access dictionary[5]. One comparison (index access).
- **0:** We access dictionary[0]. One comparison (index access).
- **6:** We access dictionary[6]. One comparison (index access).
- **8:** We access dictionary[8]. One comparison (index access).
- **4:** We access dictionary[4]. One comparison (index access).

Table:

Items	#comparisons
5	1
0	1
6	1
8	1
4	1

Thank you! I have received all three screenshots. Now I have the complete question and the graph. Let's solve it.

Question:

Consider the following acyclic digraph. Assume the adjacency lists are in sorted order; for example, when iterating through the edges pointing from 0, consider[1] the edge 0 -> 1 before 0 -> 6 or 0 -> 7.[2]

(a) Compute the topological order by running the DFS-based algorithm and listing the vertices in reverse post order.

(b) Run breadth-first search on the digraph, starting from vertex 2. List the vertices in the order in which they are de-queued from the FIFO queue.

Solution:

1. Graph Representation (Adjacency Lists):

From the image, let's create the adjacency lists, keeping in mind the sorted order:

- 0: 1, 6, 7
- 1: 2, 3
- 2: 4
- 3: 4
- 4: 5
- 5: (empty)
- 6: 4
- 7: 4

2. Topological Sort (DFS):

We will use Depth-First Search (DFS) to find the topological order.

Algorithm:

1. Initialize all nodes as "unvisited".
2. For each unvisited node, perform a DFS traversal.
3. During the post-processing step of DFS (after visiting all neighbors), add the node to a stack (or list).
4. The topological order is the reverse of the order in which nodes are popped from the stack (or the reverse of the list).

DFS Traversal:

- **Start at 0:**
    - 0 -> 1 -> 2 -> 4 -> 5

- 0 -> 1 -> 3 -> 4 (already visited)
- 0 -> 6 -> 4 (already visited)
- 0 -> 7 -> 4 (already visited)

Reverse Post Order (Stack):

Following the DFS traversal, the nodes are added to the stack in reverse post order:

1. 5
2. 4
3. 2
4. 3
5. 1
6. 6
7. 7
8. 0

Topological Order:

The topological order is the reverse of the stack:

0, 7, 6, 1, 3, 2, 4, 5

(b) Breadth-First Search (BFS) from Vertex 2:

We will use Breadth-First Search (BFS) starting from vertex 2.

Algorithm:

1. Initialize a queue and a list to store the visited nodes.
2. Enqueue the starting vertex (2) and mark it as visited.
3. While the queue is not empty:
    - Dequeue a vertex from the queue and add it to the visited list.
    - Enqueue all unvisited neighbors of the dequeued vertex and mark them as visited.

BFS Traversal:

1. Queue: [2]
2. Dequeue 2, Visited: [2], Enqueue 4
3. Queue: [4], Visited: [2, 4], Enqueue 5
4. Queue: [5], Visited: [2, 4, 5]

BFS Order (De-queued Order):

2, 4, 5

Final Answer:

(a) Topological Order (DFS): 0, 7, 6, 1, 3, 2, 4, 5

(b) BFS Order (from vertex 2): 2, 4, 5