Here are 20 potential questions for your Data Structures class test, covering key topics from the lectures:

Lecture 01: Introduction & Analysis
1. Define data structure. Explain the difference between primitive and non-primitive data structures with examples.
2. Why is worst-case analysis preferred over average-case analysis in algorithm complexity? Provide an example.
3. Calculate the time complexity of the following code:

```python
sum = 0
for i in range(n):
    for j in range(i, n):
        sum += i + j
```

4. Explain the trade-offs between time and space complexity with an example.

Lecture 02: Strings & Pattern Matching
5. Compare fixed-length, variable-length, and linked storage for strings. What are the pros and cons of each?
6. Perform the **INSERT** operation on the string "ABCDEFG" at position 3 with the substring "XYZ". Show the result.
7. Trace the **INDEX** operation for the text "HELLOWORLD" and pattern "LOW". At which position does the pattern first appear?
8. Explain how the "fast" pattern matching algorithm (using a table) works for the pattern "abababab".

Lecture 03: Arrays & Searching/Sorting
9. Given a linear array `A[5:50]` with base address 300 and element size 4 bytes, compute the address of `A[15]` and `A[35]`.
10. Trace the steps of **binary search** to find `23` in the sorted array: `[2, 5, 8, 12, 16, 23, 38, 56, 72]`.
11. Perform **bubble sort** on the array `[5, 3, 8, 1, 4]`. Show the array after each pass.
12. Write pseudocode for inserting an element into a linear array at position `k`.

Lecture 04: 2D Arrays & LCS
13. Calculate the address of element `A[3][2]` in a 2D array stored in **row-major order** with base address 200, 4 columns, and element size 4 bytes.
14. Find the **Longest Common Subsequence (LCS)** of strings "ABCBDAB" and "BDCAB" using dynamic programming. Construct the table and write the LCS.

15. Explain why dynamic programming is efficient for solving the LCS problem compared to the brute-force approach.

Lecture 05: Sorting Algorithms
16. Trace **merge sort** on the array `[18, 3, 12, 9, 4, 7]`. Show the steps of splitting and merging.
17. Perform **counting sort** on the array `[4, 2, 2, 8, 3, 3, 1]`. Show the count array and the final sorted array.
18. How does **radix sort** work? Sort `[329, 457, 657, 839, 436, 720, 355]` using LSD radix sort.
19. Compare **insertion sort** and **selection sort** with examples. Which is more efficient for nearly sorted data?
20. Explain the steps of **bucket sort** for the input `[0.78, 0.17, 0.39, 0.26, 0.72, 0.94]`.

Bonus Question
21. Write a pseudocode algorithm for the naive pattern matching method and determine its worst-case time complexity.

Tips for Preparation:
- Practice tracing algorithms step-by-step (e.g., binary search, LCS table).
- Memorize formulas for memory address calculations (row-major/column-major).
- Review time complexity derivations for sorting algorithms (e.g., bubble sort: \(O(n^2)\), merge sort: \(O(n \log n)\)).

Good luck! 😊

Answer this question with detailed explanation into a pdf File.

Clarifying the concept

I'm thinking about how to generate a PDF with detailed explanations for each question. Since I can't create PDFs directly, I'll provide text answers and suggest steps for converting them.

Crafting detailed explanations

I'm focused on generating thorough answers for 21 questions, formatted in a markdown text that users can convert to PDF, including code blocks and concise explanations for better clarity.

Summarizing the process

I'm outlining the steps to create a detailed, markdown-formatted text document. It will include headers for each question, explanations, and instructions on converting it to a PDF.

Formatting answers

I'm setting up a text document with markdown headings, detailing answers for each question. The first one clarifies the concept of data structures and their types, contrasting primitive and non-primitive

structures.

Charting time complexity

I'm noting the time complexity of a code snippet, which involves nested loops. The outer loop iterates n times, and the inner loop iterates (n-i) times, resulting in O(n^2) complexity.

Weighing memory options

I'm evaluating fixed-length, variable-length, and linked storage for strings. Fixed-length allocates a set size, variable-length adapts to needs, and linked storages use pointers to link elements.

Evaluating insertion methods

Considering fixed-length, variable-length, and linked storage for substring insertion. Fixed-length efficiently places "XYZ" after the third character of "ABCDEFG," while variable-length may need dynamic allocation.

Locating pattern occurrence

I'm pinpointing the pattern "LOW" in the text "HELLOWORLD." Considering 0-based indexing, "LOW" starts at index 2. This leads me to confirm the insertion point is after the second character.

Mapping the pattern

OK, let me see. I'm tracing "LOW" in "HELLOWORLD." It starts at index 2 (0-based), just after the second character. The insertion point for "XYZ" is after the third character of "ABCDEFG," resulting in "ABCXYZDEFG."

Mapping out pattern location

I'm analyzing the pattern "LOW" in "HELLOWORLD." Interestingly, it starts at index 2 (0-based). Also, I'll explain the "fast" pattern matching algorithm's table approach.

Refining the analysis

Confirming "LOW" starts at index 2 (0-based) in "HELLOWORLD." Next, I'll discuss the "fast" pattern matching algorithm, likely the KMP algorithm, which builds a prefix function table for efficient matching.

Mapping pattern shifts

I'm thinking through how the 'fast' pattern matching algorithm, like the KMP algorithm, uses a prefix function table to anticipate shifts and efficiently locate the pattern 'abababa' in the text.

Confirming pattern detection

Interestingly enough, "LOW" surfaces at index 2 (0-based) in "HELLOWORLD". Moving forward, I'll detail the KMP algorithm's prefix function and its pattern shift strategy.

Emptying my thought container

I'm working through calculating the addresses of A[15] and A[35] in a linear array with a base address and element size. Next, I'll walk through finding 23 in a sorted array using binary search.

Sorting and Progressing

I'm sorting the array [5, 3, 8, 1, 4] using bubble sort, making comparisons and swaps over multiple passes. Each pass brings us closer to the sorted array.

Writing pseudocode

I'm detailing pseudocode to insert an element into a linear array by shifting elements and then inserting the new element. Next, I'll calculate the address of A[3][2] in a 2D array in row-major order.

Rolling calculations

First, I calculate the address of A[3][2] with 0-based indexing using the row-major order formula, resulting in an address of 256. Then, I find the LCS of 'ABCBDAB' and 'BDCAB' using dynamic programming, arriving at "BCAB".

Tracing merge sort

I'm dividing the array [18, 3, 12, 9, 4, 7] into halves recursively, ensuring each smaller array is sorted step-by-step using merge sort.

Applying counting sort

First, I identify the maximum value (8) and create a count array. Then, I count occurrences and calculate cumulative counts to sort the array [4, 2, 2, 8, 3, 3, 1] effectively.

Sorting numbers by digits

I'm diving into LSD radix sort, beginning with the least significant digit. The array [329, 457, 657, 839, 436, 720, 355] progressively sorts from smallest to largest.

Choosing the best fit

I compare insertion sort, which excels with nearly sorted data, to selection sort, which consistently performs at O(n^2). This comparison highlights their unique efficiencies and characteristics.
Below is the complete, detailed explanation and answers for all the questions. You can copy the content into your favorite editor (for example, Microsoft Word or a Markdown editor) and export/save it as a PDF file.

Data Structures Class Test – Detailed Solutions

Lecture 01: Introduction & Analysis

1. Define data structure. Explain the difference between primitive and non-primitive data structures with examples.

Answer:

A **data structure** is a specialized format for organizing, processing, retrieving, and storing data. It defines the relationship between the data and the operations that can be performed on it.

- **Primitive Data Structures:**
  These are the basic data types provided by most programming languages. They directly operate upon machine instructions.
  Examples:
  - **Integer:** `` `int` `` in C, Java
  - **Character:** `` `char` ``
  - **Boolean:** `` `bool` ``
  - **Floating point:** `` `float` ``, `` `double` ``

- **Non-Primitive Data Structures:**
  These are more complex structures that are derived from primitive data types. They are used to store collections of data and have relationships among the elements.
  Examples:
  - **Arrays:** A collection of elements of the same type stored in contiguous memory.
  - **Linked Lists:** A series of connected nodes, where each node contains data and a pointer to the next node.
  - **Trees:** Hierarchical structures with a root and subtrees (e.g., binary trees, AVL trees).
  - **Graphs:** Consist of nodes (vertices) connected by edges.

---

2. Why is worst-case analysis preferred over average-case analysis in algorithm complexity? Provide an example.

Answer:

- **Worst-case analysis** provides a guarantee on the maximum amount of time or space an algorithm will require, regardless of the input. This is crucial for ensuring system performance and reliability, especially in real-time or mission-critical applications.
- **Average-case analysis** relies on probabilistic assumptions about input distributions and may not provide a reliable guarantee in the worst-case scenario.

Example:
Consider **Quick Sort**:

- **Average-case time complexity:** $O(n \log n)$
- **Worst-case time complexity:** $O(n^2)$
  If an application must avoid delays regardless of input (e.g., embedded systems), knowing the worst-case behavior is essential to avoid performance bottlenecks.

---

3. Calculate the time complexity of the following code:

```
python

`sum = 0
for i in range(n):
    for j in range(i, n):
        sum += i + j
`
```

Explanation & Answer:

- The **outer loop** runs n times.
- For each iteration of i, the **inner loop** runs from $j = i$ to $n - 1$, making $n - i$ iterations.

Total iterations:

$$\sum_{i=0}^{n-1} (n - i) = n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}.$$

Thus, the time complexity is $O(n^2)$.

---

4. Explain the trade-offs between time and space complexity with an example.

Answer:

Time and space are two critical resources in algorithm design. Often, optimizing one may lead to an increased requirement of the other:

- **Time Efficiency:**
  An algorithm that runs faster might use extra memory to store precomputed values or auxiliary data structures.
- **Space Efficiency:**
  An algorithm that minimizes memory usage might perform additional computations, thereby taking more time.

Example:
Memoization in Recursive Algorithms (e.g., Fibonacci):

- **Without Memoization:**
  Recursive Fibonacci computation has exponential time complexity ($O(2^n)$) but uses very little extra space.
- **With Memoization:**
  Storing previously computed results reduces the time complexity to $O(n)$ but requires additional space to hold the memoization table.

Lecture 02: Strings & Pattern Matching

5. Compare fixed-length, variable-length, and linked storage for strings. What are the pros and cons of each?

Answer:

- **Fixed-Length Storage:**
  - **Pros:**
    - Simple and fast index-based access.
    - Memory allocation is straightforward.
  - **Cons:**
    - Wastes memory if strings are shorter than the fixed length.
    - Cannot accommodate strings longer than the allocated size.
- **Variable-Length Storage:**
  - **Pros:**
    - Memory usage is efficient as only as much space as needed is used.
    - Suitable for strings of widely varying lengths.
  - **Cons:**
    - May require dynamic memory allocation, which can be slower.
    - More complex management (e.g., tracking length and end-of-string markers).
- **Linked Storage (e.g., linked lists of characters):**

- **Pros:**
  - Efficient for operations like insertion and deletion in the middle of the string.
  - No need for contiguous memory allocation.
- **Cons:**
  - Increased memory overhead due to storage of pointers.
  - Slower access time (cannot use random access; must traverse the list).

---

6. Perform the INSERT operation on the string "ABCDEFG" at position 3 with the substring "XYZ". Show the result.

Assumption:
Positions are 1-indexed (i.e., position 3 means after the 2nd character) or 0-indexed (inserting before the element at index 3). Here, we assume **insertion after the third character** (1-indexed) for clarity.

Original String: `"ABCDEFG"`

Operation: Insert `"XYZ"` after the 3rd character (`"ABC"`)

Result:

```
arduino

`"ABC" + "XYZ" + "DEFG" = "ABCXYZDEFG"
`
```

---

7. Trace the INDEX operation for the text "HELLOWORLD" and pattern "LOW". At which position does the pattern first appear?

Explanation:

- **Text:** `"HELLOWORLD"`
  (Indexes: 0→H, 1→E, 2→L, 3→L, 4→O, 5→W, 6→O, 7→R, 8→L, 9→D)
- **Pattern:** `"LOW"` (length 3)

Matching Process:

- Check starting at index 0: `"HEL"` – does not match `"LOW"`.
- Starting at index 1: `"ELL"` – does not match.
- Starting at index 2: `"LLO"` – does not match.
- **Starting at index 3:**
  - Characters: index 3 → `L`, index 4 → `O`, index 5 → `W`
  - This forms `"LOW"`, which is a match.

Answer:

- **0-indexed:** Pattern found at index **3**.
- **1-indexed:** Pattern found at position **4**.

---

8. Explain how the "fast" pattern matching algorithm (using a table) works for the pattern "ababab".

Answer:

This description refers to algorithms like **Knuth–Morris–Pratt (KMP)** that use a preprocessing table (often called the **"failure function"** or **"prefix table"**) to avoid redundant comparisons.

How it Works:

1. **Preprocessing (Building the Prefix Table):**
   - For the pattern `` `"ababab"` ``, the algorithm computes the length of the longest proper prefix which is also a suffix for every prefix of the pattern.
   - For `` `"ababab"` ``, the prefix table might look like this (0-indexed positions):

Position	Character	Longest proper prefix-suffix length
0	a	0
1	b	0
2	a	1 (matching "a")
3	b	2 (matching "ab")
4	a	3 (matching "aba")
5	b	4 (matching "abab")

2. **Pattern Matching:**
   - When a mismatch occurs during the search in the text, instead of starting over from the next character in the text, the algorithm uses the prefix table to determine the next position in the pattern to compare.
   - This avoids rechecking characters that are known to match, thus speeding up the search.

Summary:
For `` `"ababab"` ``, the prefix table helps determine how many characters can be skipped when a mismatch occurs, leading to an overall time complexity of $O(n)$ for searching in a text of length n.

Lecture 03: Arrays & Searching/Sorting

9. Given a linear array `` `A[5:50]` `` with base address 300 and element size 4 bytes, compute the address of `` `A[15]` `` and `` `A[35]` ``.

Explanation:

For an array with lower bound $L = 5$, the address of element $A[i]$ is computed by:

$$\text{Address}(A[i]) = \text{base address} + (i - L) \times \text{element size}$$

- **For `` `A[15]` ``:**

$$\text{Offset} = 15 - 5 = 10 \quad \Rightarrow \quad \text{Address} = 300 + 10 \times 4 = 300 + 40 = 340$$

- **For `` `A[35]` ``:**

$$\text{Offset} = 35 - 5 = 30 \quad \Rightarrow \quad \text{Address} = 300 + 30 \times 4 = 300 + 120 = 420$$

Answer:

- Address of `A[15]` is **340**.
- Address of `A[35]` is **420**.

---

10. Trace the steps of binary search to find `23` in the sorted array: `[2, 5, 8, 12, 16, 23, 38, 56, 72]`.

Explanation:

- **Initial Array:** Indices 0–8.

1. **First Step:**
   - **Low = 0**, **High = 8**
   - **Mid =** $\lfloor (0+8)/2 \rfloor = 4$
   - **Element at mid:** 16
   - Since $23 > 16$, search the right half.
2. **Second Step:**
   - **New Low = 5**, **High = 8**
   - **Mid =** $\lfloor (5+8)/2 \rfloor = 6$
   - **Element at mid:** 38
   - Since $23 < 38$, search the left half of this segment.
3. **Third Step:**
   - **New Low = 5**, **New High = 5**
   - **Mid = 5**
   - **Element at mid:** 23, which is the target.

Answer:
The element `23` is found at **index 5** (0-indexed).

---

11. Perform bubble sort on the array `[5, 3, 8, 1, 4]`. Show the array after each pass.

Explanation & Pass-by-Pass Sorting:

- **Initial Array:** `[5, 3, 8, 1, 4]`

Pass 1:

- Compare **5** and **3** → swap → `[3, 5, 8, 1, 4]`
- Compare **5** and **8** → no swap → `[3, 5, 8, 1, 4]`
- Compare **8** and **1** → swap → `[3, 5, 1, 8, 4]`
- Compare **8** and **4** → swap → `[3, 5, 1, 4, 8]`

Array after Pass 1: `[3, 5, 1, 4, 8]`

Pass 2:

- Compare **3** and **5** → no swap → `[3, 5, 1, 4, 8]`
- Compare **5** and **1** → swap → `[3, 1, 5, 4, 8]`
- Compare **5** and **4** → swap → `[3, 1, 4, 5, 8]`
- Compare **5** and **8** → no swap → `[3, 1, 4, 5, 8]`

Array after Pass 2: `[3, 1, 4, 5, 8]`

Pass 3:

- Compare **3** and **1** → swap → `[1, 3, 4, 5, 8]`
- Compare **3** and **4** → no swap → `[1, 3, 4, 5, 8]`
- Compare **4** and **5** → no swap → `[1, 3, 4, 5, 8]`
- Compare **5** and **8** → no swap

Array after Pass 3: `[1, 3, 4, 5, 8]`

Since no further swaps are needed, the sorted array is complete.

---

12. Write pseudocode for inserting an element into a linear array at position `k`.

Pseudocode:

```
plaintext

`procedure InsertElement(array, n, k, element)
    // Assume array has capacity for n+1 elements
    // k is the position at which to insert (0-indexed)

    if k < 0 or k > n then
        print("Invalid position")
        return

    // Shift elements from the end to position k
    for i from n downto k+1 do
        array[i] = array[i-1]

    // Insert the element at position k
    array[k] = element

    // Increase the count of elements
    n = n + 1
end procedure
`
```

Lecture 04: 2D Arrays & LCS

13. Calculate the address of element `A[3][2]` in a 2D array stored in row-major order with base address 200, 4 columns, and element size 4 bytes.

Explanation:

For a 2D array stored in row-major order, the address of element $A[i][j]$ (assuming 0-indexing) is calculated as:

$$\text{Address}(A[i][j]) = \text{base address} + ((i \times \text{number of columns}) + j) \times \text{element size}$$

Given:

- $i = 3, j = 2$
- Number of columns = 4
- Base address = 200
- Element size = 4 bytes

Calculation:

$$\text{Offset} = (3 \times 4) + 2 = 12 + 2 = 14$$

$$\text{Address} = 200 + 14 \times 4 = 200 + 56 = 256$$

Answer: The address of `A[3][2]` is **256**.

---

14. Find the Longest Common Subsequence (LCS) of strings "ABCBDAB" and "BDCAB" using dynamic programming. Construct the table and write the LCS.

Explanation:

The LCS problem can be solved using a dynamic programming table where the cell `c[i][j]` represents the length of the LCS of the first i characters of string X and the first j characters of string Y.

Let:

- $X =$ `"ABCBDAB"` (length 7)
- $Y =$ `"BDCAB"` (length 5)

Step 1: Construct the DP Table

A simplified version of the DP table is as follows (rows represent characters from X and columns from Y). (The full table is often large; here we outline the key aspects.)

```
mathematica

`       B   D   C   A   B
    ------------------------
  ''|  0   0   0   0   0
A   |  0   0   0   1   1
B   |  1   1   1   1   2
C   |  1   1   2   2   2
B   |  1   1   2   2   3
D   |  1   2   2   2   3
A   |  1   2   2   3   3
B   |  1   2   2   3   4
`
```

- The value in the bottom-right corner (4) indicates the length of the LCS is **4**.

Step 2: Trace Back to Find the LCS

One possible LCS is `"BCAB"`.

- For example, one can trace from `c[7][5]` back to find the characters:
  - B (from the last row), then A, then C, and finally B from an earlier match.

Answer:
A possible LCS is **"BCAB"**.

---

15. Explain why dynamic programming is efficient for solving the LCS problem compared to the brute-force approach.

Answer:

- **Brute-force Approach:**
  Involves checking all possible subsequences of one string to see if they are a subsequence of the other. Since a string of length n can have 2^n subsequences, this approach is **exponential** in time complexity.
- **Dynamic Programming Approach:**
  By breaking the problem into overlapping subproblems and storing their solutions in a table, dynamic programming avoids redundant calculations.
  - **Time Complexity:** $O(m \times n)$ for strings of lengths m and n.
  - This method is **polynomial** in time and far more efficient for practical input sizes.

Lecture 05: Sorting Algorithms

16. Trace merge sort on the array `[18, 3, 12, 9, 4, 7]`. Show the steps of splitting and merging.

Explanation & Steps:

1. **Splitting Phase:**
   - **Initial Array:** `[18, 3, 12, 9, 4, 7]`
     Split into two halves:
     - Left: `[18, 3, 12]`
     - Right: `[9, 4, 7]`
2. **Sort Left Half** `[18, 3, 12]`**:**
   - Split into `[18, 3]` and `[12]`
   - **Sort** `[18, 3]`**:**
     - Split into `[18]` and `[3]`
     - Merge to form `[3, 18]`
   - Merge `[3, 18]` with `[12]`:
     - Compare: 3 vs. 12 → 3 comes first
     - Then 12 vs. 18 → 12 comes next, followed by 18
     - Result: `[3, 12, 18]`
3. **Sort Right Half** `[9, 4, 7]`**:**
   - Split into `[9, 4]` and `[7]`
   - **Sort** `[9, 4]`**:**
     - Split into `[9]` and `[4]`
     - Merge to form `[4, 9]`
   - Merge `[4, 9]` with `[7]`:
     - Compare: 4 vs. 7 → 4
     - Then 7 vs. 9 → 7, then 9
     - Result: `[4, 7, 9]`
4. **Final Merge:**
   Merge the two sorted halves `[3, 12, 18]` and `[4, 7, 9]`:
   - Compare 3 and 4 → 3 is smaller
   - Next compare 12 and 4 → 4 is smaller
   - Next compare 12 and 7 → 7 is smaller
   - Next compare 12 and 9 → 9 is smaller

- Finally, append the remaining elements 12 and 18
- **Final Sorted Array:** `[3, 4, 7, 9, 12, 18]`

---

17. Perform counting sort on the array `[4, 2, 2, 8, 3, 3, 1]`. Show the count array and the final sorted array.

Explanation & Steps:

1. **Determine the Range:**
   Maximum value is 8.
2. **Initialize Count Array:**
   Create an array `count[0..8]` initialized to 0.
3. **Count Occurrences:**
   - For 4: `count[4] = 1`
   - For 2: `count[2] = 1` (first occurrence)
   - For 2: `count[2] = 2` (second occurrence)
   - For 8: `count[8] = 1`
   - For 3: `count[3] = 1` (first occurrence)
   - For 3: `count[3] = 2` (second occurrence)
   - For 1: `count[1] = 1`

   Count Array (by index):

   ```
   `count[0] = 0
   count[1] = 1
   count[2] = 2
   count[3] = 2
   count[4] = 1
   count[5] = 0
   count[6] = 0
   count[7] = 0
   count[8] = 1
   `
   ```

4. **(Optional) Cumulative Count (if building the sorted output array directly):**
   - `count[1] = 1`
   - `count[2] = 1 + 2 = 3`
   - `count[3] = 3 + 2 = 5`
   - `count[4] = 5 + 1 = 6`
   - `count[5] = 6 + 0 = 6`
   - `count[6] = 6 + 0 = 6`
   - `count[7] = 6 + 0 = 6`
   - `count[8] = 6 + 1 = 7`
5. **Build the Sorted Array:**
   Using the counts, place each number in its correct position. The sorted array becomes:
   Sorted Array: `[1, 2, 2, 3, 3, 4, 8]`

---

18. How does radix sort work? Sort `[329, 457, 657, 839, 436, 720, 355]` using LSD radix sort.

Explanation:

Radix sort (LSD - Least Significant Digit first) sorts numbers by processing individual digits from the least significant to the most significant. At each pass, a stable sorting algorithm (often counting sort) is used.

Given Array: `[329, 457, 657, 839, 436, 720, 355]`

Assume all numbers have three digits.

Pass 1: Sort by Units (Least Significant Digit)

- Extract the units digit:
  - 329 → **9**
  - 457 → **7**
  - 657 → **7**
  - 839 → **9**
  - 436 → **6**
  - 720 → **0**
  - 355 → **5**
- **Stable sort by units digit:**
  Sorted order based on units:

```scss
`720 (0), 355 (5), 436 (6), 457 (7), 657 (7), 329 (9), 839 (9)
`
```

- **Array after Pass 1:** `[720, 355, 436, 457, 657, 329, 839]`

Pass 2: Sort by Tens Digit

- Extract tens digit:
  - 720 → **2**
  - 355 → **5**
  - 436 → **3**
  - 457 → **5**
  - 657 → **5**
  - 329 → **2**
  - 839 → **3**
- **Stable sort by tens digit:**
  Grouping:
  - Tens = 2: 720, 329
  - Tens = 3: 436, 839
  - Tens = 5: 355, 457, 657
- **Array after Pass 2:** `[720, 329, 436, 839, 355, 457, 657]`

Pass 3: Sort by Hundreds Digit

- Extract hundreds digit:
  - 720 → **7**
  - 329 → **3**
  - 436 → **4**
  - 839 → **8**

    - 355 → **3**
    - 457 → **4**
    - 657 → **6**
- **Stable sort by hundreds digit:**
  Grouping:
    - Hundreds = 3: 329, 355
    - Hundreds = 4: 436, 457
    - Hundreds = 6: 657
    - Hundreds = 7: 720
    - Hundreds = 8: 839
- **Final Sorted Array:** `[329, 355, 436, 457, 657, 720, 839]`

---

19. Compare insertion sort and selection sort with examples. Which is more efficient for nearly sorted data?

Answer:

- **Insertion Sort:**
    - **Process:** Builds the final sorted array one element at a time by comparing each new element with the already sorted elements and inserting it into the correct position.
    - **Time Complexity:** Worst-case $O(n^2)$, but best-case $O(n)$ for nearly sorted data.
    - **Example:** Sorting `[1, 3, 2, 4, 5]` requires only a few comparisons and shifts.
    - **Efficiency:** Performs very well on nearly sorted data because fewer shifts are needed.
- **Selection Sort:**
    - **Process:** Repeatedly selects the smallest (or largest) element from the unsorted portion and swaps it with the first unsorted element.
    - **Time Complexity:** Always $O(n^2)$ regardless of initial order.
    - **Example:** Sorting `[1, 3, 2, 4, 5]` still involves scanning through the unsorted part in each pass.
    - **Efficiency:** Does not take advantage of nearly sorted order.

Conclusion:
Insertion sort is more efficient for nearly sorted data due to its adaptive nature.

---

20. Explain the steps of bucket sort for the input `[0.78, 0.17, 0.39, 0.26, 0.72, 0.94]`.

Explanation:

1. **Divide the Interval:**
   For inputs in the range $[0, 1)$, create several buckets. For example, if we use 5 buckets:
    - Bucket 1: $[0.0, 0.2)$
    - Bucket 2: $[0.2, 0.4)$
    - Bucket 3: $[0.4, 0.6)$
    - Bucket 4: $[0.6, 0.8)$
    - Bucket 5: $[0.8, 1.0)$
2. **Distribute the Elements:**
    - **0.78** → Bucket 4

- **0.17** → Bucket 1
- **0.39** → Bucket 2
- **0.26** → Bucket 2
- **0.72** → Bucket 4
- **0.94** → Bucket 5

3. **Sort Each Bucket:**
   Use a simple sorting algorithm (like insertion sort) on each bucket:
   - Bucket 1: `[0.17]` (already sorted)
   - Bucket 2: `[0.39, 0.26]` → sorted becomes `[0.26, 0.39]`
   - Bucket 3: Empty
   - Bucket 4: `[0.78, 0.72]` → sorted becomes `[0.72, 0.78]`
   - Bucket 5: `[0.94]`
4. **Concatenate Buckets:**
   Combine the sorted buckets to obtain the final sorted array:

$$[0.17, 0.26, 0.39, 0.72, 0.78, 0.94]$$

Bonus Question

21. Write a pseudocode algorithm for the naive pattern matching method and determine its worst-case time complexity.

Pseudocode:

```
plaintext

`procedure NaivePatternMatch(text, pattern)
    n = length(text)
    m = length(pattern)

    for i from 0 to n - m do
        j = 0
        while j < m and text[i + j] == pattern[j] do
            j = j + 1
        end while

        if j == m then
            print("Pattern found at index", i)
            // Optionally, return i or continue to find further matches
        end if
    end for
end procedure
`
```

Worst-case Time Complexity:

- In the worst-case scenario, for each of the $n - m + 1$ positions, all m characters are compared.
- **Total comparisons:** $O((n - m + 1) \times m)$, which is $O(n \times m)$.

Final Remarks

These detailed explanations cover all the key topics discussed in your lectures. You are encouraged to practice tracing the algorithms step-by-step, understand the memory address calculations, and review

the time complexity of the various sorting and pattern matching algorithms.

Instructions to Create a PDF File:

1. Copy the above text into your preferred text editor (e.g., Microsoft Word, Google Docs, or a Markdown editor).
2. Format the document if needed.
3. Export or save the document as a PDF file.

Good luck with your test preparation!