W1D3 Lab Report: Divide and Conquer (DAC)

Introduction

In this lab, we explored the concept of Divide and Conquer (DAC) through the problem of searching for a key in a **'Sorted Square Matrix'**—a 2D array where both rows and columns are sorted in non-decreasing order. The goal was to compare two algorithms: an iterative search (searchSS) and a recursive divide-and-conquer search (DACsearchSS).

(a) Generating Sorted Squares

Three types of Sorted Square matrices (M1, M2, and M3) were generated using distinct patterns. Each matrix maintains the property that rows and columns are sorted in ascending order.

(b) Iterative Search (searchSS)

The iterative search algorithm starts at the top-right corner and moves left when the current value is greater than the key, or down when it is smaller. This approach efficiently eliminates entire rows or columns at each step.

```
Algorithm searchSS(M, key)
    Input:
        M → n×n matrix (rows and columns sorted in non-decreasing order)
        key → value to search for
    Output:
        true if key is found, false otherwise

    n ← number of rows in M
    i ← 0       // start at first row
    j ← n - 1  // start at last column

    while i < n and j ≥ 0 do
        if M[i][j] = key then
            return true
        else if M[i][j] > key then
            j ← j - 1   // move left, eliminate column j
        else
            i ← i + 1  // move down, eliminate row i
        end if
    end while
    return false
End Algorithm
```

Time Complexity: O(n)
Space Complexity: O(1)

(c) DAC Search (DACsearchSS)

The divide-and-conquer approach recursively splits the matrix into four quadrants and searches in three of them based on comparisons with the central element. The recurrence relation is **T(n) = 3T(n/2) + O(1).** Using the Master Theorem, this yields a time complexity of **O(n^1.585)** and a space complexity of **O(log n).**

```
Algorithm DACsearchSS(M, rs, re, cs, ce, key)
    Input:
        M → n×n matrix (rows and columns sorted)
        rs, re → start and end row indices
        cs, ce → start and end column indices
        key → value to search for
    Output:
        true if key is found, false otherwise

    if rs > re or cs > ce then
        return false
    end if

    if rs = re and cs = ce then
        return (M[rs][cs] = key)
    end if

    midR ← (rs + re) / 2
    midC ← (cs + ce) / 2
    pivot ← M[midR][midC]

    if pivot = key then
        return true
    else if key < pivot then
        // eliminate bottom-right quadrant (Q4)
        return
            DACsearchSS(M, rs,     midR - 1, cs,     midC - 1, key) OR   // Q1
            DACsearchSS(M, rs,     midR - 1, midC,   ce,       key) OR   // Q2
            DACsearchSS(M, midR,   re,       cs,     midC - 1, key)      // Q3
    else
        // eliminate top-left quadrant (Q1)
        return
            DACsearchSS(M, rs,     midR,     midC + 1, ce,       key) OR // Q2
            DACsearchSS(M, midR+1, re,       cs,       midC,     key) OR // Q3
            DACsearchSS(M, midR+1, re,       midC+1,   ce,       key)    // Q4
    end if
End Algorithm
```

Recurrence Relation : $\boldsymbol{T(n) = 3T(n/2) + O(1)}$

Time Complexity: $\boldsymbol{O(n^{\log_2 3}) \approx O(n^{1.585})}$

Space Complexity: $\boldsymbol{O(\log\ n)}$

(d1) Mathematical Comparison

Algorithm	Strategy	Recurrence /Formula	Time Complexity	Space Complexity
searchSS	Iterative staircase	`T(n) = O(n)`	`O(n)`	`O(1)`
DACsearchSS	Divide & Conquer	`T(n) = 3T(n/2) + O(1)`	`O(n^1.585)`	`O(log n)`

Mathematically, since n^1.585 grows faster than n, the iterative algorithm is asymptotically more efficient for large matrices.

(d2) Empirical Comparison

Empirical testing shows that searchSS executes fewer operations and uses less memory. Recursive overhead in DACsearchSS grows significantly with n.

Reflection – Appropriateness of DAC

The DAC approach is elegant and demonstrates problem decomposition well, but it is not ideal for this problem. The sorted property allows direct elimination of rows and columns, making recursion unnecessary. DAC is better suited when subproblems require independent solutions (e.g., matrix multiplication, image processing).

Conclusion

This lab highlights that Divide and Conquer is not universally optimal. For sorted matrices, a simple iterative approach achieves superior efficiency. Understanding when to apply DAC — and when to rely on structural properties—is key to algorithmic design.