Work 3 Solutions

Question 1

```
Algorithm generateM1(n,start)
Create matrix m[n][n]
for i←0 to n-1 do
     for j←0 to n-1 do
          m[i][j]←start
          start←start+1
return m
```

```
Algorithm generateM3(n,start)
Create matrix m[n][n]
for i←0 to n-1 do
     for j←0 to n-1 do
          m[j][i]←start
          start←start+1
return m
```

```
Algorithm generateM2(n,start)
Create matrix m[n][n]
for d←0 to 2*n-2 do
   if d % 2 = 0 then
     for i←0 to n-1 do
       j←d-i
       if j>= 0 AND j < n then
          m[i][j]←start
          start←start + 1
    else
      for i←n-1 to i=0 do
        j←d-i
        if j>= 0 AND j < n then
          m[i][j]←start
          start←start + 1

return m
```

Question b

```
 Algorithm searchSS(matrix m, integer key)
    n ← number of rows (or columns) in m
    if n = 0 OR key < m[0][0] OR key > m[n-1][n-1] then
        print "Not Found!"
        return
    i ← 0
    j ← n − 1
    while i < n AND j ≥ 0 do
        if m[i][j] = key then
            print (i,j)
```

```
            return
        else if m[i][j] > key then
            j ← j - 1
        else
            i ← i + 1
    print "Not Found!"
```

b1 ← The time complexity is O(n+n) i.e O(n) the size fo the matrix , at most it goes n + n steps

b2 → The space complexity is O(1), because of variables n, i, j

C

```
DACsearchSS(matrix m, key, startRow, endRow, startCol, endCol)
    if startRow > endRow OR startCol > endCol then
        print "Not Found"
        return

    midRow ← (startRow + endRow)/ 2
    midCol ← (startCol + endCol)/ 2
    midVal ← m[midRow][midCol]

    if midVal = key then
        print (midRow,midCol)
        return
    else if midVal > key then
        searchDivideAndConquer(m, key, startRow, midRow - 1,
startCol, midCol - 1)
        searchDivideAndConquer(m, key, startRow, midRow - 1,
midCol, endCol)
        searchDivideAndConquer(m, key, midRow, endRow,
startCol, midCol - 1)
    else
```

```
        searchDivideAndConquer(m, key, midRow + 1, endRow,
midCol + 1, endCol)
        searchDivideAndConquer(m, key, startRow, midRow,
midCol + 1, endCol)
        searchDivideAndConquer(m, key, midRow + 1, endRow,
startCol, midCol)
```

c1 Time complexity is $O(n^{\log_2 3})$. As this is divided into 3 subproblems and nearly n/2 size of input so T(n) = 3T(n/2) + f(n) . where f(n) = O(1) , a =3 , b=2 , k=0; So the DAC approach is **worse than O(n)** but better than brute-force $O(n^2)$.

c2 . Space Complexity Recursive calls are nested → recursion depth = $\log_2(n)$ (matrix halves each time) Space for recursion stack: $\boldsymbol{S(n) = O(\log\ n)}$ So the algorithm uses **O(log n)** auxiliary space.

d1 Mathematical Comparison

Algorithm	Time Complexity	Space Complexity	Notes
searchSS	O(n)	O(1)	Linear scan along staircase; optimal for sorted 2D matrix
DACsearchSS	O(n^{1.585})	O(log n)	Recursive; subdivides into 3 quadrants; worse than staircase

Mathematical Insight:

- searchSS is asymptotically faster for large matrices.
- DACsearchSS has slightly higher space overhead (recursion stack) and worse exponent.
- So for practical purposes, searchSS is **more efficient** for this problem.

d2 Empirical Comparison

If we were to test on n x n matrices:

- searchSS would perform **≈ 2n comparisons** in the worst case.
- DACsearchSS would perform **≈ 3^{$\log_2$ n} ≈ n^{1.585} comparisons**, noticeably more for large n.
- Memory usage:

- searchSS uses almost no extra memory.
- DACsearchSS uses a recursion stack of depth ~log n.

Empirical conclusion: staircase search is simpler, faster, and more memory-efficient in practice.

Report Discussion

1. Concepts Learned

- **Matrix search strategies** depend heavily on data structure and ordering.
- Staircase search exploits **monotonicity along rows and columns**.
- Divide-and-conquer is **not always optimal**, even though it's elegant.
- Recursive algorithms can be **slower than iterative ones** if they explore multiple overlapping subproblems.

---

2. Appropriateness of DAC

- DAC divides the matrix into quadrants to prune search space.
- But each call recursively explores **3 sub-quadrants** → overhead accumulates.
- DAC is elegant for **fully independent subproblems** but inefficient for a **matrix with cross-quadrant constraints**.
- For 2D sorted matrices, **staircase search is simpler and faster**.

---

3. Key Takeaways

1. **Algorithm choice depends on structure**:
    - Staircase search: O(n) for sorted 2D matrix.
    - DAC: O(n^{1.585}) – elegant but less efficient here.
2. **Recursive vs Iterative**:
    - Recursion adds stack overhead.
    - Iterative approaches can be more memory-efficient.
3. **Master theorem** is useful for analyzing DAC recurrences.
4. **Empirical testing** confirms theoretical predictions.

Conclusion: For searching a 2D sorted matrix, **staircase search is superior to DAC** in both time and space.

Question 2

$2^n < 2^{n+1} < 2^{2n} < 2^{2^n}$