C Sc 70010 – Algorithms

Fall 2026 Midterm Exam – Solutions

October 21, 2026

Problem 1: Let $A = [A_1 < A_2 < \cdots < A_n]$ and $B = [B_1 < B_2 < \cdots < B_n]$ be two sorted arrays, each containing $n \geq 1$ distinct positive integers, and let k be a positive integer. Prove that your algorithms are correct and analyze their running times.

Problem 1(a): Design the most efficient algorithm you can that either returns a pair of indices (i, j), with $1 \leq i, j \leq n$, such that $A_i + B_j = k$, or reports that no such pair exists. If several such pairs exist, returning any one of them is sufficient.

A $\Theta(n^2)$ solution: brute force. For every $1 \leq i \leq n$ and every $1 \leq j \leq n$, check whether $A_i + B_j = k$. Return the first pair (i, j) that satisfies the equation, or report NO after all pairs have been checked.

Correctness: every pair is checked.

Complexity: there are n^2 pairs and each check takes $\Theta(1)$ time. When the answer is NO all pairs are checked, so the worst-case running time is $\Theta(n^2)$. This algorithm does not use the fact that the arrays are sorted.

A $\Theta(n \log n)$ solution: n binary searches. For every $1 \leq i \leq n$, use binary search to look for the value $k - A_i$ in the sorted array B. If it is found at index j, return (i, j). If none of the n searches succeeds, report NO.

Correctness: for a fixed i, there exists j with $A_i + B_j = k$ if and only if $k - A_i$ appears in B, and binary search on a sorted array finds a value if and only if it is there.

Complexity: each binary search makes at most $\lfloor \log_2 n \rfloor + 1$ comparisons, so the worst case (a NO answer) takes $\Theta(n \log n)$ time. This algorithm uses only the fact that B is sorted.

A $\Theta(n)$ solution: two indices moving toward each other. The algorithm uses both sorted orders. Index i scans A from left to right (from small to large values) and index j scans B from right to left (from large to small values).

- Initially, $i = 1$ and $j = n$.
- While $i \leq n$ and $j \geq 1$:
  - If $A_i + B_j = k$: return (i, j).
  - If $A_i + B_j > k$: set $j \leftarrow j - 1$.
  - If $A_i + B_j < k$: set $i \leftarrow i + 1$.
- Report NO (there is no pair (i, j) such that $A_i + B_j = k$).

Correctness. The algorithm maintains the following invariant: *at the beginning of every iteration, every solution (i^*, j^*) (that is, every pair with $A_{i^*} + B_{j^*} = k$) satisfies $i \leq i^*$ and $j^* \leq j$.*

- Initially $i = 1$ and $j = n$, so the invariant holds trivially.
- Suppose $A_i + B_j > k$. Let (i^*, j^*) be a solution. By the invariant $i^* \geq i$, and therefore $A_{i^*} + B_j \geq A_i + B_j > k$. Hence B_j cannot be part of any solution, that is, $j^* \neq j$, and so $j^* \leq j - 1$. Decreasing j keeps the invariant.
- Suppose $A_i + B_j < k$. Let (i^*, j^*) be a solution. By the invariant $j^* \leq j$, and therefore $A_i + B_{j^*} \leq A_i + B_j < k$. Hence A_i cannot be part of any solution, that is, $i^* \neq i$, and so $i^* \geq i + 1$. Increasing i keeps the invariant.

If the algorithm returns (i, j), then $A_i + B_j = k$ was verified directly. If the loop ends, then $i = n + 1$ or $j = 0$. By the invariant, a solution would need $n + 1 \le i^*$ or $j^* \le 0$, which is impossible. Therefore the NO answer is correct.

Complexity. Every iteration either returns, or increases i by 1, or decreases j by 1. While the loop runs, i can be increased at most $n - 1$ times and j can be decreased at most $n - 1$ times without leaving the arrays, and one more step ends the loop. Hence there are at most $(n-1)+(n-1)+1 = 2n-1$ iterations. Each iteration takes $\Theta(1)$ time, so the running time is $O(n)$. When the answer is NO, the loop runs until $i = n + 1$ or $j = 0$, which takes at least n iterations. Hence the worst-case running time is $\Theta(n)$.

Remark: the one-array version. The more common version of this problem has only *one* sorted array $A = [A_1 < A_2 < \cdots < A_n]$, and the goal is to find two indices $1 \le i, j \le n$ such that $A_i + A_j = k$. The linear-time algorithm for that version uses two indices ℓ and r of the *same* array. Initially $\ell = 1$ and $r = n$. While $\ell \le r$: if $A_\ell + A_r = k$, return (ℓ, r); if $A_\ell + A_r > k$, set $r \leftarrow r - 1$; and if $A_\ell + A_r < k$, set $\ell \leftarrow \ell + 1$. When $\ell > r$, report NO.

The differences between the two versions are:

- *The stopping condition.* In the one-array version, the pairs (i, j) and (j, i) give the same sum $A_i + A_j = A_j + A_i$, so once $\ell > r$ every remaining pair is a pair that was already ruled out, with its two indices swapped. Therefore the search can stop when $\ell > r$. In the two-array version, i is an index of A and j is an index of B, so comparing i with j is meaningless: the pairs (i, j) and (j, i) represent different sums $A_i + B_j$ and $A_j + B_i$. The search must continue until one of the indices leaves its array, that is, until $i = n + 1$ or $j = 0$.
- *The number of iterations.* In the one-array version, every iteration shrinks the range $[\ell..r]$ by one, so there are at most n iterations. In the two-array version, there are at most $2n - 1$ iterations. Both are $\Theta(n)$.
- *The case $i = j$.* In the one-array version, $i = j$ means that the same integer A_i is used twice, and the problem must specify whether this is allowed (the condition $\ell \le r$ allows it). In the two-array version, A_i and B_i are different integers and there is no such issue.

Problem 1(b): Design an algorithm that either returns a pair of indices (i, j), with $1 \leq i, j \leq n$, such that $A_i - B_j = k$, or reports that no such pair exists. If several such pairs exist, returning any one of them is sufficient.

$\Theta(n^2)$ and $\Theta(n \log n)$ solutions. The brute-force algorithm checks all n^2 pairs in $\Theta(n^2)$ time. A faster algorithm uses binary search to look for the value $A_i - k$ in B for every $1 \leq i \leq n$, in $\Theta(n \log n)$ time. The correctness of both is as in part (a).

A $\Theta(n)$ solution: two indices moving in the same direction. In part (a), increasing i increases the sum $A_i + B_j$ and decreasing j decreases it, which is why i starts at 1 and j starts at n. For the difference $A_i - B_j$, increasing i increases the difference and increasing j *decreases* it. Therefore both indices can start at 1 and both move forward. Alternatively, both indices can start at n and both move backward. Without loss of generality, the algorithm below starts both indices at 1.

- Initially, $i = 1$ and $j = 1$.
- While $i \leq n$ and $j \leq n$:
  - If $A_i - B_j = k$: return (i, j).
  - If $A_i - B_j < k$: set $i \leftarrow i + 1$.
  - If $A_i - B_j > k$: set $j \leftarrow j + 1$.
- Report NO (there is no pair (i, j) such that $A_i - B_j = k$).

Correctness. The algorithm maintains the following invariant: *at the beginning of every iteration, every solution (i^*, j^*) (that is, every pair with $A_{i^*} - B_{j^*} = k$) satisfies $i \leq i^*$ and $j \leq j^*$.*

- Initially $i = 1$ and $j = 1$, so the invariant holds trivially.
- Suppose $A_i - B_j < k$. Let (i^*, j^*) be a solution. By the invariant $j^* \geq j$, and therefore $A_i - B_{j^*} \leq A_i - B_j < k$. Hence A_i cannot be part of any solution, that is, $i^* \neq i$, and so $i^* \geq i + 1$. Increasing i keeps the invariant.
- Suppose $A_i - B_j > k$. Let (i^*, j^*) be a solution. By the invariant $i^* \geq i$, and therefore $A_{i^*} - B_j \geq A_i - B_j > k$. Hence B_j cannot be part of any solution, that is, $j^* \neq j$, and so $j^* \geq j + 1$. Increasing j keeps the invariant.

If the algorithm returns (i, j), then $A_i - B_j = k$ was verified directly. If the loop ends, then $i = n + 1$ or $j = n + 1$. By the invariant, a solution would need $i^* \geq n + 1$ or $j^* \geq n + 1$, which is impossible. Therefore the NO answer is correct.

Complexity. Every iteration either returns or increases i or j by 1. As in part (a), there are at most $(n - 1) + (n - 1) + 1 = 2n - 1$ iterations, each taking $\Theta(1)$ time. When the answer is NO, there are at least n iterations. Hence the worst-case running time is $\Theta(n)$.

Problem 2: For $n \geq 2$, let M be an $n \times n$ matrix containing n^2 distinct positive integers. For $1 \leq i, j \leq n$, the entry in row i and column j of M is denoted by $M[i, j]$. In M, the integers in each row are sorted in increasing order (from left to right) and the integers in each column are sorted in increasing order (from top to bottom). That is, for any three indices $1 \leq k \leq n$ and $1 \leq i < j \leq n$, the following hold:

$$M[k, i] < M[k, j] \quad \text{and} \quad M[i, k] < M[j, k]$$

Let x be a positive integer. The goal is to either find the position (i, j) such that $M[i, j] = x$, or report that x does not appear in M. The cost of an algorithm is the number of comparisons it makes between x and entries of M; all other operations are free. Each comparison of x with $M[i, j]$ has one of three outcomes:

$$x < M[i, j] \quad \text{or} \quad x > M[i, j] \quad \text{or} \quad x = M[i, j]$$

You may assume that n is a power of 2 and may ignore floors and ceilings. You must justify your answers.

Problem 2(a): Describe the most efficient algorithm you can (fewest comparisons in the worst case) that finds the position (i, j) of x in M, or reports that x is not in M. What is the exact worst-case number of comparisons made by your algorithm, as a function of n?

A $\Theta(n^2)$ solution: brute force. Compare x with every entry of M. Return (i, j) if $x = M[i, j]$ is found, and report NO otherwise.

Correctness: every entry is checked.

Complexity: the worst case is when x is not in M. Then all n^2 entries are compared, so the worst case is exactly n^2 comparisons. This algorithm uses neither the order of the rows nor the order of the columns.

A $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$ solution: divide and conquer. Split M into four $\frac{n}{2} \times \frac{n}{2}$ quadrants (top-left, top-right, bottom-left, bottom-right) and compare x with $c = M[\frac{n}{2}, \frac{n}{2}]$, the bottom-right entry of the top-left quadrant.

- If $x = c$: return $(\frac{n}{2}, \frac{n}{2})$.
- If $x < c$: every entry of the bottom-right quadrant is larger than c (it is below and to the right of c), so x is not there. Recurse on the other three quadrants.
- If $x > c$: every entry of the top-left quadrant is at most c (c is its largest entry), so x is not there. Recurse on the other three quadrants.

Correctness: each quadrant is itself sorted by rows and by columns, so the recursion is valid, and each eliminated quadrant cannot contain x.

Complexity: the worst case is when x is not in M. Then the algorithm recurses on three quadrants at every level. A 1×1 matrix needs one comparison, and the worst-case number of comparisons satisfies

$$T(1) = 1, \qquad T(n) = 3\,T(n/2) + 1.$$

Unfolding the recurrence k times gives

$$T(n) = 3\,T(n/2) + 1 = 3^2\,T(n/4) + 3 + 1 = \cdots = 3^k\,T(n/2^k) + \sum_{i=0}^{k-1} 3^i.$$

For $k = \log_2 n$, the sub-problems are of size $n/2^k = 1$, so $T(n/2^k) = 1$, and the sum is a geometric series:

$$T(n) = 3^{\log_2 n} + \frac{3^{\log_2 n} - 1}{3 - 1} = \frac{3 \cdot 3^{\log_2 n} - 1}{2}.$$

Since $3^{\log_2 n} = \left(2^{\log_2 3}\right)^{\log_2 n} = \left(2^{\log_2 n}\right)^{\log_2 3} = n^{\log_2 3}$, the exact solution is

$$T(n) = \frac{3n^{\log_2 3} - 1}{2} = \Theta(n^{\log_2 3}),$$

where $\log_2 3 \approx 1.585$.

Remark: applying the master theorem. The recurrence has the form $T(n) = a\,T(n/b) + f(n)$ with $a = 3$, $b = 2$, and $f(n) = 1$. Since $f(n) = 1 = O(n^{\log_2 3 - \epsilon})$ for $\epsilon = 1$, the first case of the master theorem applies, and $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$. The master theorem gives only the order of growth, whereas the unfolding above gives the exact number of comparisons.

This algorithm uses the order of both the rows and the columns, but with a wrong idea: a comparison with c eliminates only one quadrant, that is, only a quarter of the remaining entries, and three sub-problems of size $\frac{n}{2} \times \frac{n}{2}$ remain. As a result, it is better than the brute-force algorithm but worse than the n binary searches, which use only the order of the rows.

A $\Theta(n \log n)$ solution: one binary search per row. Run a binary search for x in each of the n sorted rows. Return (i, j) if one of the searches finds x, and report NO otherwise.

Correctness: x is in M if and only if it is in one of the rows, and binary search on a sorted row finds x if and only if it is there.

Complexity: with three-way comparisons, binary search in a sorted array of size n (a power of 2) makes at most $\log_2 n + 1$ comparisons. The worst case is when x is not in M. Then all n searches are performed, for a total of $n(\log_2 n + 1) = \Theta(n \log n)$ comparisons in the worst case. This algorithm uses only the order of the rows.

Remark: n binary searches of the columns, instead of the rows, also work, with the same correctness proof and the same number of comparisons. That algorithm uses only the order of the columns.

Motivation for the linear-time solution. The n binary searches of the rows can be improved by using the order of the columns as well. Search the rows from top to bottom. If the binary search in row i fails, it ends with an index $0 \le p \le n$ such that $M[i, p] < x < M[i, p + 1]$ (where $p = 0$ means that $x < M[i, 1]$ and $p = n$ means that $x > M[i, n]$). For every column $j > p$ and every row $r > i$, $M[r, j] > M[i, j] \ge M[i, p + 1] > x$. Hence the columns $p + 1, \ldots, n$ can be discarded, and the binary search in row $i + 1$ can be restricted to the columns $1, \ldots, p$.

On some inputs this improvement saves many comparisons, but in the worst case it does not help. If $x > M[i, n]$ for every row $i < n$, then every search ends with $p = n$. Each search discards only its own row, the next search range is still all the columns $1, \ldots, n$, and the number of comparisons remains $\Theta(n \log n)$.

The staircase walk below fights this worst case. It also discards columns $p + 1, \ldots, n$ after each row, but instead of a binary search in the columns $1, \ldots, p$ of the next row, it scans the next row from column p to the left. The scan of a row costs one comparison for each column it discards plus one comparison that discards the row itself. In the bad case above, every row costs a single comparison. In general, every comparison discards a row or a column, and so the total number of comparisons is at most $2n - 1$.

A $\Theta(n)$ solution: the staircase walk. Start at the top-right corner and maintain a current position (i, j).

- Initially, $i = 1$ and $j = n$.
- While $i \le n$ and $j \ge 1$, compare x with $M[i, j]$:
  - If $x = M[i, j]$: return (i, j).
  - If $x < M[i, j]$: set $j \leftarrow j - 1$ (discard column j).
  - If $x > M[i, j]$: set $i \leftarrow i + 1$ (discard row i).
- Report NO (x does not appear in M).

Remark: the walk can also start at the bottom-left corner, with $i = n$ and $j = 1$. In that version, after $x < M[i, j]$ the walk discards row i and moves up ($i \leftarrow i - 1$), and after $x > M[i, j]$ it discards column j and moves right ($j \leftarrow j + 1$). The correctness proof and the complexity analysis are symmetric.

Two example runs. The figure shows the staircase walk on the same 8×8 matrix for two values of x. The visited cells are shaded and numbered in the order of the comparisons. After $x < M[i, j]$ the walk moves one cell to the left, and after $x > M[i, j]$ it moves one cell down. The cell in which x is found has a thick border, and the arrow shows where the walk leaves the matrix.

1	4	12	20	31	39	46 (2)	75 (1)
2	8	17	24	38 (5)	56 (4)	61 (3)	76
3	13	18	26	40 (6)	62	65	88
5	14	21	34	43 (7)	63	72	94
11	16	23	42	45 (8)	69	81	95
19	22	44 (11)	55 (10)	59 (9)	77	85	97
27	47 (13)	51 (12)	57	67	78	90	98
48 (15)	50 (14)	54	66	83	89	93	99

$x = 48$ (in the matrix): found at $(8, 1)$ after $15 = 2n - 1$ comparisons.

1	4	12	20	31	39 (3)	46 (2)	75 (1)
2	8	17	24	38 (5)	56 (4)	61	76
3	13	18	26	40 (6)	62	65	88
5	14	21	34 (8)	43 (7)	63	72	94
11	16	23 (10)	42 (9)	45	69	81	95
19	22 (12)	44 (11)	55	59	77	85	97
27 (14)	47 (13)	51	57	67	78	90	98
← 48 (15)	50	54	66	83	89	93	99

$x = 41$ (not in the matrix): NO after $15 = 2n - 1$ comparisons.

The three middle anti-diagonals are the cells with $i + j \in \{n, n + 1, n + 2\}$. In the left run, the walk leaves them in both directions: $x = 48$ is compared with $M[2, 5] = 38$, above them ($i + j = 7$), and with $M[6, 5] = 59$, below them ($i + j = 11$). The value 48 is found at $(8, 1)$ after $2n - 1$ comparisons, so the walk may need $2n - 1$ comparisons even when x is in the matrix. In the right run, $x = 41$ is not in the matrix, and the walk leaves the matrix on the left side of the last row after $2n - 1$ comparisons.

Correctness. The algorithm maintains the following invariant: *if x appears in M, then it appears in the sub-matrix with rows $i, \ldots, n$ and columns $1, \ldots, j$.* Initially, this sub-matrix is all of M. Note that $M[i, j]$ is the *largest* entry of row i in the sub-matrix and the *smallest* entry of column j in the sub-matrix.

- If $x < M[i, j]$, then x is smaller than every entry of column j in the sub-matrix (they are all at least $M[i, j]$). Column j can be discarded.
- If $x > M[i, j]$, then x is larger than every entry of row i in the sub-matrix (they are all at most $M[i, j]$). Row i can be discarded.

If the loop ends, then $i = n + 1$ or $j = 0$, the sub-matrix is empty, and by the invariant x is not in M.

Complexity (upper bound). Every comparison that does not find x discards one row or one column. The loop continues as long as at least one row and at least one column remain. Hence before the last comparison at most $n - 1$ rows and at most $n - 1$ columns were discarded, and the total number of comparisons is at most $(n - 1) + (n - 1) + 1 = 2n - 1$.

Complexity (the bound is tight for this algorithm). A worst case is when x is not in M, although, as the left example run shows, the walk may need $2n-1$ comparisons also when x is in M. Define $M[i,j] = 2n(i+j)+i$. Each row and each column is increasing, and all entries are distinct because the value determines $i+j$ (the quotient when dividing by $2n$) and i (the remainder). Let $x = 2n(n+1)+n+1$. Then x is not in M (its remainder when dividing by $2n$ is $n+1$), x is larger than every entry with $i+j = n+1$, and x is smaller than every entry with $i+j = n+2$. The walk visits the cells

$$(1,n),\ (2,n),\ (2,n-1),\ (3,n-1),\ \ldots,\ (n,2),\ (n,1),$$

alternating between the anti-diagonal $i+j = n+1$ (where $x > M[i,j]$, so the walk moves down) and the diagonal just below it $i+j = n+2$ (where $x < M[i,j]$, so the walk moves left). It ends after $n+(n-1) = 2n-1$ comparisons. The figure shows this run for $n = 8$, where $M[i,j] = 16(i+j)+i$ and $x = 153$.

33	49	65	81	97	113	129	145 (1)
50	66	82	98	114	130	146 (3)	162 (2)
67	83	99	115	131	147 (5)	163 (4)	179
84	100	116	132	148 (7)	164 (6)	180	196
101	117	133	149 (9)	165 (8)	181	197	213
118	134	150 (11)	166 (10)	182	198	214	230
135	151 (13)	167 (12)	183	199	215	231	247
152 (15)	168 (14)	184	200	216	232	248	264

↓

$M[i,j] = 16(i+j)+i$ and $x = 153$ (not in the matrix): NO after $15 = 2n-1$ comparisons.

Therefore, the exact worst-case number of comparisons of *this particular algorithm*, the staircase walk, is $2n-1$. This example shows only that the staircase walk cannot do better than $2n-1$ comparisons in the worst case. It does not show that every algorithm needs that many comparisons. Lower bounds for every algorithm are the subject of Problem 2(b) and the bonus.

Problem 2(b): Prove that any algorithm that solves this problem, using only comparisons between x and entries of M, must make at least n comparisons in the worst case.

A first proof (an adversary argument). The adversary restricts the inputs to matrices with the following structure and announces this structure to the algorithm in advance: every entry above the anti-diagonal (the cells with $i + j \leq n$) is smaller than x, every entry below the anti-diagonal (the cells with $i + j \geq n + 2$) is larger than x, and every anti-diagonal entry is larger than all the entries above the anti-diagonal and smaller than all the entries below it. Extra information can only help an algorithm, so a lower bound for an algorithm that knows this structure is also a lower bound for the original problem.

Observation: under this structure, every order of the n anti-diagonal entries is legal. Every cell to the left of or above an anti-diagonal cell lies above the anti-diagonal, and every cell to the right of or below it lies below the anti-diagonal. Hence the sorted order of the rows and the columns only requires each anti-diagonal entry to be larger than the entries above the anti-diagonal and smaller than the entries below it, and it says nothing about how two anti-diagonal entries compare with each other.

Therefore, the outcome of a comparison of x with an entry off the anti-diagonal is already known to the algorithm, and the task is reduced to deciding whether x is one of n arbitrary integers $d_1, \ldots, d_n$ (the anti-diagonal entries), about whose order nothing is known.

The adversary strategy: whenever the algorithm compares x with an anti-diagonal entry d_r, the adversary answers $x < d_r$. Suppose the algorithm stops after fewer than n comparisons. Then some d_r was never compared with x. Both of the following inputs are legal and consistent with all the answers: the input in which all the anti-diagonal entries are larger than x (the correct answer is NO), and the input in which $d_r = x$ and all the other anti-diagonal entries are larger than x (the correct answer is the position of d_r). The algorithm gives the same output on both, so it is wrong on one of them. Hence every algorithm must make at least n comparisons in the worst case. □

A second proof (an explicit construction). This proof replaces the adversary by two explicit inputs that the algorithm cannot tell apart.

The input matrices. Let $M_0[i, j] = 2n(i + j) + i$ and $x = 2n(n + 1) + n + 1$ (the matrix and value from Problem 2(a)). As shown there, M_0 is sorted by rows and by columns, its entries are distinct, and x is not in M_0. Moreover:

- every entry with $i + j \leq n + 1$ (on or above the anti-diagonal) is smaller than x,
- every entry with $i + j \geq n + 2$ (below the anti-diagonal) is larger than x.

For $1 \leq r \leq n$, let M_r be the matrix obtained from M_0 by replacing the anti-diagonal entry in cell $(r, n + 1 - r)$ with x. The matrix M_r is still sorted with distinct entries: the cells to the left of and above $(r, n + 1 - r)$ have $i + j \leq n$, so their entries are smaller than x; the cells to the right of and below it have $i + j \geq n + 2$, so their entries are larger than x. In M_r the correct answer is $(r, n + 1 - r)$, while in M_0 the correct answer is NO.

The argument. Let $\mathcal{A}$ be any (deterministic) algorithm that solves the problem, and run it on M_0 with the value x. Suppose, for contradiction, that $\mathcal{A}$ makes fewer than n comparisons. Since $|D| = n$, there is an anti-diagonal cell $(r, n + 1 - r)$ that $\mathcal{A}$ never compares with x.

Now run $\mathcal{A}$ on M_r with the same x. The matrices M_0 and M_r differ only in cell $(r, n+1-r)$. By induction on the number of comparisons made so far, $\mathcal{A}$ makes the same comparisons on both inputs and receives the same outcomes: as long as the comparisons are the same, none of them involves cell $(r, n + 1 - r)$, so their outcomes are the same in M_0 and in M_r, and therefore $\mathcal{A}$'s next comparison is also the same. Hence $\mathcal{A}$ produces the same output on both inputs. But the correct outputs are different (NO for M_0 and $(r, n + 1 - r)$ for M_r), so $\mathcal{A}$ is wrong on one of them. This contradiction shows that $\mathcal{A}$ makes at least n comparisons on the input M_0.

Therefore any algorithm must make at least n comparisons in the worst case. □

Problem 2(b) – Bonus: Prove that any algorithm that solves this problem, using only comparisons between x and entries of M, must make at least $2n-1$ comparisons in the worst case.

A first proof of the bonus (an adversary argument). For $1 \le r \le n$, let d_r denote the anti-diagonal entry in cell $(r, n+1-r)$, and for $2 \le r \le n$, let e_r denote the entry in cell $(r, n+2-r)$, just below the anti-diagonal. The adversary announces in advance that every entry in a cell with $i+j \le n$ is smaller than x and smaller than all the d's and e's, that every entry in a cell with $i+j \ge n+3$ is larger than x and larger than all the d's and e's, and that x is either not in M or one of the $2n-1$ entries $d_1, \ldots, d_n, e_2, \ldots, e_n$. Again, this information can only help the algorithm.

Observation: under this structure, the only constraints among the d's and e's are the $n-1$ triplets

$$d_r < e_{r+1} > d_{r+1} \qquad \text{for } 1 \le r \le n-1,$$

since e_{r+1} is below d_r (in column $n+1-r$) and to the right of d_{r+1} (in row $r+1$). No other two of these cells are in a common row or column, and no path that only moves to the right and down leads from one of them to another, except the pairs in these triplets. Therefore any assignment of values that satisfies all the $n-1$ triplets is legal.

The adversary strategy: whenever the algorithm compares x with some d_r, the adversary answers $x > d_r$, and whenever it compares x with some e_r, the adversary answers $x < e_r$. These answers are consistent with the input in which all the d's are smaller than x and all the e's are larger than x, so x is not in M and the correct answer is NO. Suppose the algorithm stops before comparing x with all the $2n-1$ entries. Then some entry was never compared with x, and setting this entry to x keeps the input legal and consistent with all the answers:

- If $d_r = x$, then d_r is an outer entry of at most two triplets, and the constraints $d_r < e_r$ and $d_r < e_{r+1}$ (for those of e_r, e_{r+1} that exist) hold because all the e's are larger than x.
- If $e_{r+1} = x$, then e_{r+1} is the middle entry of the triplet $d_r < e_{r+1} > d_{r+1}$, and this triplet holds because all the d's are smaller than x.

In the new input the correct answer is the position of that entry, and not NO. The algorithm gives the same output on both inputs, so it is wrong on one of them. Hence every algorithm must make at least $2n-1$ comparisons in the worst case. □

A second proof of the bonus (an explicit construction). The explicit construction of the second proof of part (b) extends to $2n-1$ cells.

Let $D' = \{(r, n+2-r) : 2 \le r \le n\}$ be the $n-1$ cells just below the anti-diagonal (the cells with $i+j = n+2$). For a cell $c = (r, n+2-r) \in D'$, let M'_c be the matrix obtained from M_0 by replacing the entry in cell c with x. The matrix M'_c is still sorted with distinct entries: the cells to the left of and above c have $i+j \le n+1$, so their entries are smaller than x; the cells to the right of and below c have $i+j \ge n+3$, so their entries are larger than x. In M'_c the correct answer is c.

Hence x can be "hidden" in any one of the $|D| + |D'| = n + (n-1) = 2n-1$ cells of $D \cup D'$ without changing the entry in any other cell. The argument of the second proof of part (b) applies, word for word, to every cell of $D \cup D'$: if $\mathcal{A}$, running on M_0, never compares x with some cell $c \in D \cup D'$, then $\mathcal{A}$ produces the same output on M_0 and on the matrix in which x is hidden in c, and it is wrong on one of them. Therefore, on the input M_0, $\mathcal{A}$ must compare x with every cell of $D \cup D'$, that is, $\mathcal{A}$ makes at least $2n-1$ comparisons.

Note that these are exactly the $2n-1$ cells that the staircase walk visits on the input M_0 in Problem 2(a).

The outcomes of the comparisons on M_0 are exactly the answers of the adversary in the first proof of the bonus: x is larger than every entry on the anti-diagonal and smaller than every entry just below it.

Remarks.

- By the bonus, the staircase walk of Problem 2(a), which makes at most $2n-1$ comparisons in the worst case, is optimal.
- The lower bound rules out algorithms that make $O(1)$ or $O(\log n)$ comparisons. In particular, there is no "two-dimensional binary search" with $O(\log n)$ comparisons.
- If the order of all the n^2 entries were known, for example if $M[i,n] < M[i+1,1]$ for every $1 \le i \le n-1$, then reading the matrix row by row would give a sorted array of n^2 entries. In that case, binary search would find x, or report that x is not in M, with at most $\log_2(n^2)+1 = 2\log_2 n+1$ comparisons. The sorted order of the rows and the columns is not enough for this: it does not determine how $M[i,j]$ and $M[k,\ell]$ compare when $i > k$ and $j < \ell$, that is, when the cell (i,j) is below and to the left of the cell (k,ℓ). Every two anti-diagonal cells are in this position, and this freedom is exactly what the lower-bound proofs exploit.
- Allowing x to be absent is not essential for the lower bounds, but it makes them exactly n and $2n-1$. If x were guaranteed to be in M, the same adversaries would prove lower bounds of $n-1$ and $2n-2$: the adversary answers "not equal" as long as at least two candidate cells remain uncompared, since x can be hidden in either of them. Once only one candidate cell remains, the algorithm knows that x is there without a final comparison. These bounds are tight: when x is guaranteed to be in M, the staircase walk can skip its last comparison, since once a single row and a single column remain, x must be in their common cell. Hence the staircase walk makes at most $2n-2$ comparisons in that version.