4.1 Counting Inversions

**Question:** Let $A$ be an array of $N$ distinct numbers. A pair $(A[i], A[j])$ is an inversion if $i < j$ but $A[i] > A[j]$. Find the total number of inversions in $A$.

**The discovery dialog.** Below is a dialog between a teacher and a student for solving the problem above.

1. **Teacher:** How fast is brute-force?
   
   **Student:** $O(N^2)$ time. For each item $A[i]$, we need to go to $N - i = O(N)$ elements to see if there are $A[j] < A[i]$, with $j > i$.

2. **Teacher:** Let’s try to achieve $O(N \log(N))$ time. Is there a set of items that can be sorted by size or some key?
   
   **Student:** Can’t sort the array, because that destroys the order of elements, and we can’t realize inversions anymore.

3. **Teacher:** Are there certain operations being done repeatedly, such as searching, or finding the largest (or smallest) element? Can you use a data structure to speed up these queries? What about a hash table or a heap?
   
   **Student:** We don’t search for a specific number. Instead, for each item $A[i]$, we have to compare it with every other item $A[j]$ with $j > i$. Unclear how to use data structure on this.

4. **Teacher:** Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?
   
   **Student:** Let’s try divide-and-conquer. Here I have an array, so it’s natural to divide it into two halves. I’d use recursion to find the number of inversions in each half.
   
   For example, if $A = [1, 8, 5, 4, 7, 2, 6, 3]$, I’ll recurse on $[1, 8, 5, 4]$ and $[7, 2, 6, 3]$. The left subarray has 3 inversions: $(8, 5)$, $(8, 4)$, and $(5, 4)$. Likewise, the right subarray has 4 inversions: $(7, 2)$, $(7, 6)$, $(7, 3)$, and $(6, 3)$.

5. **Teacher:** Do you miss any inversion in the example above?
   
   **Student:** Yes, I missed the “crossing” inversions: $(8, 7)$, $(8, 2)$, $(8, 6)$, $(8, 3)$, $(5, 2)$, $(5, 3)$, $(4, 2)$, and $(4, 3)$. For each such inversion, one element is in the left subarray, and the other in the right subarray. We can’t realize them if we only work on each subarray individually.

6. **Teacher:** Is there a brute-force algorithm to calculate those crossing inversions? How fast is it?
   
   **Student:** $O(N^2)$ time. For each item in the left subarray, we have to look at $N/2$ items on the right subarray. Totally the time is $N/2 \cdot N/2 = O(N^2)$. 

4-1
7. **Teacher**: Ideally we want $O(N)$ time for counting the crossing inversions, so that the running time of our algorithm is $O(N \log(N))$. But first, let’s try to count the crossing inversions in $O(N \log(N))$ time. Is there a set of items that can be sorted by size or some key?

**Student**: Of course we can’t sort the entire array, but can we sort the two subarrays? This turns out that sorting the subarrays doesn’t affect the number of crossing inversions.

8. **Teacher**: Now assume that the subarrays are sorted. How fast can you find the crossing inversions?

**Student**: For each $A[i]$ in the left subarray, I need to find every $j$ in the right subarray such that $A[j] > A[i]$. Previously I had to go to every element in the right subarray, but it’s sorted now.

This time, I can use binary search to find the smallest index $j$ in the right half such that $A[j] > A[i]$ in the right subarray within $O(\log(N))$ time. So totally I can solve the problem in $O(N \log(N))$ time.

9. **Teacher**: Let’s find the number of crossing inversions in $O(N)$ time. Where are the bottlenecks?

**Student**: There are two places we use $O(N \log(N))$ time: sorting the two subarrays, and using binary search for $O(N)$ times.

10. **Teacher**: Let’s get rid of the sorting cost first. What’s the total cost of sorting?

**Student**: So the recursion is $T(N) = 2T(N/2) + O(N \log(N))$. Using the recursion tree, the cost is $O(N^2 \log(N))$.

11. **Teacher**: Let’s aim to cut the sorting cost to just $O(N \log(N))$ time. You are using a recursive algorithm. Can you visualize the execution of sorting in a small example via a recursion tree? For example, consider $[1, 8, 5, 4, 7, 2, 6, 3]$.

**Student**: Yes, see Figure 4.1.

Figure 4.1: The recursion tree for the sorting operations in the first divide-and-conquer Inversion Counting algorithm. Each internal node is labeled with an array that its parent will need to sort and the corresponding sorted array.

12. **Teacher**: What’s the reason that sorting takes $O(N \log^2(N))$ instead of $O(N \log(N))$? Are there any operations that are done multiple times where they should be done just once?

**Student**: Yes, the array of each subtree keeps being sorted for every ancestor of the root.

13. **Teacher**: How would you avoid sorting the same array multiple times?

**Student**: To avoid that waste, we should keep data sorted. That is, not only we want the number of inversions, we also want to sort the array.
14. **Teacher**: That’s known as the “Inventor’s Paradox”. In induction proofs or recursive algorithms, sometimes your job will be easier if you aim for more, because you’ll have more from the recursive case. Now the recursion will automatically sort the two subarrays, and that gets rid of the sorting problem. Now you still need to figure out how to count the number of crossing inversions on two sorted arrays in linear time. Moreover, you need to somehow combine the two sorted subarrays into a fully sorted one.

If you cannot solve the proposed problem, try to solve first some related problem. Could you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Could you solve a part of the problem? Keep only a part of the condition, drop the other part.

**Student**: I don’t know how to count the crossing inversions in \( O(N) \) time. But I know that if I want to combine the two sorted arrays, I can use the Merge algorithm in Merge Sort.

15. **Teacher**: In the Merge algorithm, we have to compare each element of the left subarray with those in the right subarray. This seems to suggest that we can exploit that to count the crossing inversions as well. Let’s try to turn the Merge algorithm into a Merge-and-Count one.

**Student**: Here is an idea that seems to work. Every time we compare \( A[i] \) on the left with \( A[j] \) on the right during the execution of Merge, if \( A[i] < A[j] \) then we will increment our count of crossing inversions.

16. **Teacher**: Let’s try your algorithm on \([1, 4, 5, 8]\) and \([2, 3, 6, 7]\). Is there a problem?

**Student**: Yes, when we compare 4 and 2, we realize that \((4, 2)\) is an inversion, but then the Merge algorithm moves 2 to the sorted array, and we will miss the inversions \((5, 2)\) and \((8, 2)\). See Figure 4.2 for an illustration.

![Figure 4.2: A snapshot of the naive Merge-and-Count algorithm.](image)

17. **Teacher**: Is there a way that we can realize there are two other inversions \((\cdot, 2)\)?

**Student**: Yes, there are two elements after 4, and these are larger than 4. So any of them will form an inversion with 2. Thus we know that we have two more inversions \((\cdot, 2)\). This gives us a Merge-and-Count algorithm of \( O(N) \) time.

**Reflection**. To solve the problem above, we need two key observations whose realization is made easier thanks to the visualization:
• In the recursion tree of the first divide-and-conquer algorithm, the array of each node keeps getting sorted for every ancestor.

• When we compare two elements $L$ and $R$ in the Merge algorithm from the left and right subarrays respectively, if $(L, R)$ is an inversion then for any subsequent element $X$ of the left subarray, $(X, R)$ is also an inversion.

Also, we learn the Inventor’s Paradox, which is useful if you want to solve a problem recursively.

### 4.2 Maximum Subarray

**Question:** Let $A$ be an array of $N$ (possibly negative) numbers. Find a pair $(i, j)$ with $i \leq j$ that maximizes the sum $A[i] + \cdots + A[j]$.

**The discovery dialog.** Below is a dialog between a teacher and a student for solving the problem above.

1. **Teacher:** How fast is brute-force?
   
   **Student:** $O(N^3)$ time. I’ll need to look at $O(N^2)$ pairs $(i, j)$ with $i < j$, and for each pair, I compute the sum $S[i, j] = A[i] + \cdots + A[j]$ using $O(N)$ time. I then find the maximum among those $S[i, j]$.

2. **Teacher:** Draw a picture for your execution on $A = [3, -2, -1, 5]$.
   
   **Student:** See Figure 4.3.

   ![](4-3: Figure 4.3: Illustration for the brute-force algorithm of the Maximum Subarray on $A = [3, -2, -1, 5]$. For simplicity, we only show the computation of $S[1, j]$, with $j = 1, \ldots, 4$.)

3. **Teacher:** Are there any operations that are done multiple times where they should be done just once?
   
   **Student:** Yes. Each $S[i, j]$ is re-computed several times. For example, when I compute $S[1, 4]$, I implicitly re-compute $S[1, 2]$ and $S[1, 3]$. To avoid this waste, I can compute $S[i, j]$ via $S[i, j - 1] + A[j]$ if $j > i$. This improves the running time of the brute-force algorithm to $O(N^2)$.

4. **Teacher:** Let’s push the running even further, to $O(N \log(N))$. Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?
   
   **Student:** Let’s try divide-and-conquer. Here I have an array, so it’s natural to divide it into two halves. I’d use recursion to find the maximum subarray in each half, and then compare them to find the maximum subarray overall.
5. **Teacher**: Try your algorithm on \( A = [1, 2, 3, 4] \).

**Student**: I’ll recurse on \([1, 2]\) and \([3, 4]\) to realize that the maximum subarray on the left is \([1, 2]\), and that on the right is \([3, 4]\). So the maximum between these two is \([3, 4]\). Wait! This is **not** the overall maximum subarray \([1, 2, 3, 4]\).

6. **Teacher**: What’s the issue? Why did you miss this subarray?

**Student**: The recursion didn’t look at a “crossing” subarray that spans from the left subarray to the right one. So I still need to find the maximum crossing subarray.

7. **Teacher**: What’s the brute-force algorithm for this task?

**Student**: I’ll need to look at \( N/2 \) indices \( i \) from the left and \( N/2 \) indices \( j \) on the right. Totally I need to look at \( N^2/4 = O(N^2) \) pairs \((i, j)\) to evaluate \( S[i, j] \). This costs me \( O(N^2) \) time.

8. **Teacher**: We need \( O(N) \) time for finding the maximum “crossing” subarray, so that our divide-and-conquer can run within \( O(N \log(N)) \) time.

**Student**: Here I have two indices \( i \) and \( j \) to maximize the crossing subarray. I don’t see how to find them within \( O(N) \) time.

9. **Teacher**: If you cannot solve the proposed problem, try to solve first some related problem. Could you imagine a more accessible related problem? A more general problem? A more special problem? An analogous problem? Could you solve a part of the problem? Keep only a part of the condition, drop the other part.

**Student**: If we fix one of the two indices, say \( i \), then I can easily find the best \( j \) within \( O(N) \) time.

10. **Teacher**: Visualize the brute-force algorithm if \( i \) is fixed. For example, consider \( i = 1 \) and \( A = [3, -2, -3, 5, -1, 3] \).

**Student**: See Figure 4.4.

![Figure 4.4: Illustration for the brute-force algorithm for finding the maximum crossing subarray on \( A = [3, -2, -3, 5, -1, 3] \), with \( i = 1 \).](image)

11. **Teacher**: From the picture, the computation of \( S[1, 4] \) is the bottleneck. But do we really need this number? What is the desired output?

**Student**: The maximizing index \( j \) (which is 6 in this case).

12. **Teacher**: Is there a way to know this index without explicitly computing \( S[1, 4] \)?

**Student**: Yes. The three sums \( S[1, 4], S[1, 5], S[1, 6] \) share the common partial sum \( 3 + (-2) + (-3) \). We don’t need to evaluate this to know which one is the max.
13. **Teacher:** So how would you find the maximizing $j$?

**Student:** I’ll compute $P[j] = A[N/2 + 1] + \cdots + A[j]$ for every $j = N/2 + 1, \ldots, N$, and then find the maximizing $j$.

14. **Teacher:** Elaborate your algorithm on the array $A = [3, -2, -3, 5, -1, 3]$, for all choices $i = 1, 2, 3$.

**Student:** For $i = 1$, we have $P[4] = 5$, $P[5] = 4$, and $P[6] = 7$. The maximum of them is 7, so the best $j$ is 6. For $i = 2$, wait, the formula for the best $j$ doesn’t depend on $i$, so $j$ is 6 regardless of what $i$ is.

15. **Teacher:** Now, if $i$ is no longer fixed, how would you find the maximizing pair $(i, j)$?

**Student:** I’ll first find the index $j$ that maximizing $P[j]$. This takes $O(N)$ time because $P[j] = P[j-1] + A[j]$ for every $j > N/2 + 1$. Once $j$ is fixed, I can easily find the best $i$ within $O(N)$ time.

**Reflection.** In this example, we first try to improve the brute-force algorithm from $O(N^3)$ time to $O(N^2)$ time, before switching to a different design paradigm (divide-and-conquer) to have an $O(N \log(N))$ time. At the first glance, working on the improved version of brute-force is a waste of time, but it actually gives us an idea for computing the $P[i]$ terms using $O(N)$ time instead of $O(N^2)$ time. Next, when we have to optimize with many parameters ($i$ and $j$ in this case), it’s usually a good strategy to fix all but one of those to have a more accessible related problem.

Visualization again proves its usefulness, since it helps us to realize the unnecessary repetition in the brute-force algorithm, and the bottleneck in computing the maximizing $j$ for the crossing subarrays.

It seems rather surprising to us that somehow the best $j$ doesn’t depend on the choice of $i$. Let’s try to find an intuitive explanation so that we can see it at a glance. Note that


If we can find $i$ that maximizes $A[i] + A[i+1] + \cdots + A[N/2]$, and $j$ that maximizes $A[N/2 + 1] + \cdots + A[j]$, then that $(i, j)$ will be the maximizing pair. That’s why we can choose the best $i$ and $j$ independently.

### 4.3 Local Minimum in Grid

**Question:** You are given an $n$-by-$n$ grid of distinct numbers. A number is a local minimum if it is smaller than all its neighbors. (A neighbor of a number is one immediately above, below, to the left, or to the right. Most numbers have four neighbors; numbers on the side have three; the four corners have two.) Find a local minimum.

**The Discovery Dialog.** Below is a dialog between a teacher and a student for solving the problem above.

1. **Teacher:** What exactly does the input consist of?

   **Student:** A 2D array of size $n \times n$ whose elements are distinct numbers.

2. **Teacher:** What is the output?

   **Student:** A local minimum, namely an element that is smaller than the neighboring entries.

3. **Teacher:** Does that element even exist?

   **Student:** Yes, the minimum element of the array is a local minimum.
4. **Teacher**: How fast is brute-force?

   **Student**: I can look into each of the $n^2$ element of the array, and for each element, I need to check $O(1)$ adjacent entries to see if it’s a local minimum. Overall I need $O(n^2)$ time.

5. **Teacher**: Now let’s test the specialization-generalization paradigm on this problem. Could you find a special version of it?

   **Student**: This is a 2D problem, so maybe we should start with the 1D version? That is, find a local minimum in an array $A[1 : n]$ of $n$ distinct elements.

6. **Teacher**: What’s the running time of brute-force on this new problem?

   **Student**: A linear scan will take $O(n)$ time.

7. **Teacher**: Let’s aim to reduce it to $O(\log(n))$ time. Do you have any plan to attack it?

   **Student**: Sounds like another binary-search variant. So I’ll first look at the middle element to see if it’s a local minimum . . .

**Exercise**: Solve this 1D problem.

8. **Teacher**: Now let’s try to find a generalization of our original problem. Generalize “2D array”, but do not go to 3D or higher dimensions, because it’s complex.

   **Student**: In programming, the interpretation of 2D array is actually broader. It actually means a **ragged array**. So what if we work on an array of $n$ columns, each column of at most $k$ entries?

9. **Teacher**: Contrast this setting with the 1D one to see if we can somehow tweak the old solution for the new problem.

   **Student**: In the 1D problem, each element is simply a number. Here each “element” is an entire column. In the 1D problem, we compare numbers with numbers. If we carry this analogy to our new setting, we have to “compare” columns with columns somehow. But what does it mean?

10. **Teacher**: So we need to somehow find a corresponding number for each column. Ideally, this should be an element of the column. Moreover, when we terminate the binary-search process and pick a column, the corresponding element should be a local minimum of the ragged array.

    **Student**: Could we use a local minimum within each column for this purpose?

11. **Teacher**: Let’s try to find a counter-example for this approach. Start with a small example, say two columns, and each column has three entries.

    **Student**: I have six numbers here, so maybe I should pick them from $\{1, 2, 3, 4, 5, 6\}$. There are two plausible approaches here:

    - **Big gap**: I’ll pick from $\{1, 2, 3\}$ for the first column, and $\{4, 5, 6\}$ for the second one. For the first column, I’ll let it be $(1, 3, 2)$ in that order. My target is 2, which is a local minimum, but is not the global minimum. Likewise, my second column should be either $(5, 6, 4)$ or $(4, 6, 5)$, and the target is 4.

    - **Small gap**: I’ll pick from $\{1, 3, 5\}$ for the first column, and $\{2, 4, 6\}$ for the second one. For the first column, I’ll let it be $(1, 5, 3)$ in that order. My target is 3, which is a local minimum, but is not the global minimum. Likewise, my second column should be either $(2, 6, 4)$ or $(4, 6, 2)$, and the target is 4.

    Upon inspecting four cases above, I find the following counter-example:
\[
\begin{pmatrix}
1 & 4 \\
5 & 6 \\
3 & 2
\end{pmatrix}
\]

Here the two numbers are 3 and 4, but none of them is a local minimum in the entire 2D array.

12. **Teacher**: So one should not simply pick a local minimum in each column. Which element should you choose?

**Student**: From the counter-example, it looks like I should pick the *global* minimum in each column.

13. **Teacher**: How will we know that the algorithm is correct? Let’s say you get the number \( M_i \) in the \( i \)-th column. How small is it, compared to the top and bottom neighbors?

**Student**: It’s smaller than those neighbors, because \( M_i \) is the minimum entry in the \( i \)-th column.

14. **Teacher**: What about the left and right neighbors? For symmetry, we only need to argue that \( M_i \) is smaller than the left neighbor \( L \).

**Student**: The algorithm might never compare \( M_i \) and \( L \) at all, so I’m not sure how to argue that \( M_i < L \).

15. **Teacher**: Here we compare each column with the adjacent ones. The number \( M_i \) is in column \( i \), and \( L \) is in column \( i - 1 \). Those columns are indeed adjacent. How would the algorithm compare them?

**Student**: It compares \( M_i \) with the minimum entry \( M_{i-1} \) in column \( i - 1 \).

16. **Teacher**: Draw an illustration with \( M_{i-1}, M_i \) and \( L \).

**Student**: See Figure 4.5.

![Figure 4.5: The three elements \( M_{i-1}, M_i \), and \( L \) with their relative positions.](image)

17. **Teacher**: Given that \( M_i \) is chosen, what can you tell about the relationship between \( M_i \) and \( M_{i-1} \)?

**Student**: Then \( M_i < M_{i-1} \). But then \( M_{i-1} \leq L \), because \( M_{i-1} \) is the minimum entry in column \( i - 1 \). So indeed, \( M_i < M_{i-1} \leq L \).

18. **Teacher**: So the algorithm is correct. How fast is it?

**Student**: The binary-search has \( O(\log(n)) \) main steps. In each step, we have to find the minimum of some columns, which is \( O(n) \). Hence totally our algorithm is \( O(n \log(n)) \) time.

19. **Teacher**: Let’s push the running time to \( O(n) \). Is there a way to split the problem in two smaller problems, perhaps by doing a binary search? How about partitioning the elements into big and small, or left and right? Does this suggest a divide-and-conquer algorithm?

**Student**: We may try divide-and-conquer, but how should I divide? This is no longer a 1D array where I can divide it to the left and right subarrays.
20. **Teacher**: In divide-and-conquer, we usually go from a problem of size \( n \) to subproblems of size \( n/2 \). Restate the problem for size \( n/2 \).

**Student**: Find a local minimum within a 2D array of size \( n/2 \times n/2 \).

21. **Teacher**: Given an \( n \times n \) array, can you divide it to arrays of size \( n/2 \times n/2 \)?

**Student**: Yes, I can divide it to 4 quadrants.

22. **Teacher**: Now we aim for \( O(n) \) time. Let’s say that you will do some \( \Theta(n^k) \) time to reduce the search within \( a \leq 4 \) quadrants. So \( T(n) = aT(n/2) + \Theta(n^k) \). What should \( a \) and \( k \) be?

**Student**: From the Master theorem, I think there are two cases: (1) \( a = 1 \) and \( k = 1 \), or (2) \( a = 2 \) and \( k = 0 \). The first case is like Binary Search but we are allowed \( O(n) \) cost for processing the middle element. The second case is like Merge Sort but we are allowed just \( O(1) \) cost for the Merge operation.

23. **Teacher**: Both directions are plausible. But what would you think which is likely to be the correct route?

**Student**: Given that the 1D version follows a binary search, I think the first approach is likely to be correct here.

24. **Teacher**: Any idea which quadrant to pick?

**Student**: Could I simply pick an arbitrary quadrant to search? It takes \( O(1) \) cost.

25. **Teacher**: Let’s find a counter-example for your algorithm. Try to search exhaustively on small arrays, say \( 2 \times 2 \) ones, with elements drawn from \( \{1,2,3,4\} \).

**Student**: Consider the following array:

\[
\begin{pmatrix}
1 & 2 \\
3 & 4
\end{pmatrix}
\]

Here if we go to the bottom left quadrant then we’ll end up with 3, but this is not a local minimum.

26. **Teacher**: So what’s the issue with your approach?

**Student**: The local minimum within a quadrant may happen to be at the boundary of the quadrant. In that case, it may not be a local minimum in the global context.

27. **Teacher**: Look back at the 1D version. If we pick an arbitrary subarray from the two possible choices, we’ll end up with the same fate: a local minimum in a subarray may not be a local minimum in the global context. How did we resolve it?

**Student**: Initially we look at the middle element with its two adjacent elements. If the middle element is not a local minimum, one of its two neighbors \( Y \) must be smaller than \( X \). We’ll then recurse on the subarray that contains \( Y \).

28. **Teacher**: Carry this analogy to the 2D case.

**Student**: If the center \( X \) of the grid is not a local minimum then one of its four neighbors \( Y \) is smaller than \( X \). We’ll then recurse on a quadrant that contains \( Y \).

29. **Teacher**: Let’s try to find a counter-example for this approach. Let’s try a \( 3 \times 3 \) grid. Look for a tie, meaning that any of the four neighbors is smaller than the center, so the choice of the quadrant will be arbitrary.

**Student**: I’ll pick the elements of the grid from \( \{1,2,\ldots,9\} \). Also, let the center be the maximum element, namely 9. The other elements can be drawn in an arbitrary fashion from \( \{1,2,\ldots,8\} \).
Given that the choice of quadrant is arbitrary, here I’ll pick the bottom right quadrant, namely

\[
\begin{pmatrix}
9 & 5 \\
7 & 8
\end{pmatrix}
\]

This quadrant doesn’t contain a local minimum in the global context.

30. **Teacher:** In the 1D version, we have only \(O(1)\) time, so we only look at a single element (namely the middle one). But here we have \(O(n)\) time, so we can look at a bigger area. What is a plausible candidate for this “middle area”?

**Student:** This means I should look for an area of \(O(n)\) elements that covers the center of the grid. But there are many ways to draw such an area!

31. **Teacher:** Look back at your first failed approach (meaning picking an arbitrary quadrant). In that case a local minimum within a quadrant may happen to be at the boundary, causing troubles. So we need to somehow deal with the boundary of the quadrants.

**Student:** So here are two plausible candidates for the middle area, illustrated in Figure 4.6.

![Figure 4.6: Two candidates for the middle area before we decide which quadrant to recurse.](image)

32. **Teacher:** We’ll decide which candidate is the correct one later. Initially, we’ll look at the middle area to see if there’s a local minimum. If not, we need to recurse on a quadrant. Carry the analogy of the 1D case in picking the correct quadrant?

**Student:** So we need to identify an element \(X\) in the middle area. If \(X\) is not a local minimum then one of its four neighbors \(Y\) must be smaller than \(X\). We’ll look at the quadrant that contains \(Y\). But I don’t know how to pick \(X\).

33. **Teacher:** In your failed approach, you pick \(X\) as the center. But it doesn’t have to be the case. After all, you have \(O(n)\) time to determine what is \(X\) in the middle area. Here we have an area and we have to pick a single representative element. Have you seen a similar problem?

**Student:** Yes. In the \(O(n \log(n))\) solution, an area is a column, and its representative is the minimum entry. So maybe here \(X\) should be the minimum entry of the area.
34. **Teacher:** Let’s say we pick a quadrant as prescribed above, and recurse. Assume that it returns a point Z at the boundary of the quadrant. Will Z be a local minimum in the global context?

**Student:** I have no clue how to prove that, since there’s very little information from the recursion.

35. **Teacher:** This is again the “Inventor’s Paradox”. The recursion should give us more information about Z than merely saying that it’s a local minimum. What property do we want from Z?

**Student:** Z should be smaller than the neighbor that is outside the quadrant, so that it’s a truly local minimum in the global context.

36. **Teacher:** But the recursion doesn’t have information about this neighbor. The property should only involve elements within the quadrant. To refine the property, we need to see if we have used all the data. What do we know about this quadrant?

**Student:** It contains Y which is smaller than the minimum X of the given area.

37. **Teacher:** Given this data, refine the property about Z.

**Student:** Maybe Z ≤ Y? Since Y is smaller than the minimum of the given area, if Z ≤ Y then Z is smaller than any entry in the area.

38. **Teacher:** Better. But the recursion is given a quadrant; it doesn’t know which is Y among the elements in the boundary. Refine the property of Z even further.

**Student:** So maybe Z is at most the minimum on the boundary. It guarantees that Z ≤ Y.

39. **Teacher:** OK. Let’s look at the global context.

- **Requirement:** Our algorithm needs to get a local minimum V from an n × n grid, and this V needs to be at most the minimum of the boundary.
- **How we handle:** At first, we look at the minimum element X of the middle area. If X is not a local minimum then one of the four neighbors Y must be smaller than X. We’ll recurse on the quadrant that contains Y.

Now, if X is a local minimum then we’ll return X. So X needs to be at most the minimum of the boundary of the entire grid. Which candidate of the middle area satisfies this?

**Student:** The second one, since it contains the boundary of the entire grid.

40. **Teacher:** In the case that X is not a local minimum, we’ll recurse and get a value Z. The recursion guarantees that Z ≤ Y (meaning that it’s truly a local minimum in the global context). But is Z smaller than the minimum of the boundary of the entire grid so that we can meet the requirement?

**Student:** We have Z ≤ Y < X, and X is at most the minimum of the boundary. So yes, Z is smaller than the minimum of the boundary of the entire grid. This algorithm indeed works!

**Reflection.** In this problem we again appreciate the power of the generalization-specialization paradigm. It takes us from solving the 1D problem in \(O(\log(n))\) time to dealing with the 2D ragged array in \(O(n \log(n))\) time. These two cases provide ideas for solving the 2D array in \(O(n)\) time. We also learn how to construct counter-examples to invalidate wrong approaches: look at small cases, exhaust all cases, and look for a tie. This problem also showcases the power of the “Inventor’s Paradox”. This is a very useful technique when you design recursive algorithms.