Question 2.1:

Provide an algorithm to compute the following linear reccurrence in $\mathcal{O}(\log n)$ time:

$$g(n) = \begin{cases} n & n < 4 \\ 2 \cdot g(n-2) - 3 \cdot g(n-3) & n \geq 4 \end{cases}$$

Answer:

We can generalize this to the case where $g(n)$ is any linear recurrence of degree t. That is:

$$g(n) = \sum_{i=1}^{t} \alpha_i \cdot g(n-i)$$

We can construct the matrix A:

$$A := \begin{pmatrix} \alpha_1 & \alpha_2 & \cdots & \alpha_t \\ 1 & & & \\ & \ddots & & \\ & & 1 & 0 \end{pmatrix}$$

Which means that:

$$A \cdot \begin{pmatrix} g(n) \\ \vdots \\ g(n-t+1) \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{t} \alpha_i g(n+1-i) \\ g(n) \\ \vdots \\ g(n-t) \end{pmatrix} = \begin{pmatrix} g(n+1) \\ \vdots \\ g(n-t) \end{pmatrix}$$

So recursively, that means that for every $n \geq t$:

$$\begin{pmatrix} g(n) \\ \vdots \\ g(n-t+1) \end{pmatrix} = A^{n-t} \begin{pmatrix} g(t) \\ \vdots \\ g(1) \end{pmatrix}$$

We can prove this through induction:

Base case: $n = t$

If $n = t$ then $A^{n-t} = A^0 = I$, so essentially what we need to prove is

$$\begin{pmatrix} g(t) \\ \vdots \\ g(1) \end{pmatrix} = \begin{pmatrix} g(t) \\ \vdots \\ g(1) \end{pmatrix}$$

Which is true.

Inductive step:

Suppose this is true for n. We need to show that it is true for $n+1$. So:

$$\begin{pmatrix} g(n) \\ \vdots \\ g(n-t+1) \end{pmatrix} = A^{n-t} \begin{pmatrix} g(t) \\ \vdots \\ g(1) \end{pmatrix}$$

And since

$$A \cdot \begin{pmatrix} g(n) \\ \vdots \\ g(n-t+1) \end{pmatrix} = \begin{pmatrix} g(n+1) \\ \vdots \\ g(n-t) \end{pmatrix}$$

This means:

$$\begin{pmatrix} g(n+1) \\ \vdots \\ g(n-t) \end{pmatrix} = A^{n+1-t} \begin{pmatrix} g(t) \\ \vdots \\ g(1) \end{pmatrix}$$

As required.

So in order to compute $g(n)$, it is sufficient to compute

$$A^{n-t} \cdot \begin{pmatrix} g(t) \\ \vdots \\ g(1) \end{pmatrix}$$

So all we need is to create an algorithm to compute matrix powers in $\mathcal{O}(\log n)$ time.
Notice that:

$$A^{n-t} = \begin{cases} A^{\frac{n-t}{2}} \cdot A^{\frac{n-t}{2}} & 2 \mid n \\ A \cdot A^{\frac{n-t-1}{2}} \cdot A^{\frac{n-t-1}{2}} & 2 \nmid n \end{cases}$$

(The split into cases is necessary so that the powers are all natural.)
I will denote the function that computes A^n as $\mathrm{pow}(\mathrm{A}, \mathrm{n})$. And the function to multiply two matrices will be $\mathrm{matMult}(\mathrm{A}, \mathrm{B})$. So we define our pow function to be:

$$\mathrm{pow}(\mathrm{A}, \mathrm{n}) \to \begin{cases} \mathrm{matMult}\left(\mathrm{pow}\left(\mathrm{A}, \frac{\mathrm{n}}{2}\right), \mathrm{pow}\left(\mathrm{A}, \frac{\mathrm{n}}{2}\right)\right) & 2 \mid n \\ \mathrm{matMult}\left(\mathrm{A}, \mathrm{matMult}\left(\mathrm{pow}\left(\mathrm{A}, \frac{\mathrm{n}-1}{2}\right), \mathrm{pow}\left(\mathrm{A}, \frac{\mathrm{n}-1}{2}\right)\right)\right) & 2 \nmid n \end{cases}$$

We can define matMult to just compute matrix multiplication normally, meaning that:

$$\mathrm{matMult}(\mathrm{A}, \mathrm{B})_{\mathrm{i,j}} \to \sum_{\mathrm{k}=1}^{\mathrm{t}} \mathrm{A}_{\mathrm{i,k}} \cdot \mathrm{B}_{\mathrm{j,k}}$$

So this is an $\mathcal{O}(t)$ operation t^2 times (for every element of the matrix), so $\mathrm{matMult} \in \mathcal{O}\left(\mathrm{t}^3\right) = \mathcal{O}(1)$ since t is a constant.
This means that if $T(n)$ is the time complexity of $\mathrm{pow}(\mathrm{A}, \mathrm{n})$ then:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

Which, by the Master Theorem, has a time complexity of $\mathcal{O}(\log n)$.
So in order to calculate $g(n)$, we compute

$$\mathrm{matMult}\left(\mathrm{pow}(\mathrm{A}, \mathrm{n}-\mathrm{t}), \begin{pmatrix} g(t) \\ \vdots \\ g(1) \end{pmatrix}\right)$$

And then take the first index (which is $\mathcal{O}(1)$). This operation takes

$$\mathcal{O}(\mathrm{pow}(\mathrm{n}-\mathrm{t})) + \mathcal{O}(\mathrm{matMult}) + \mathcal{O}(1) = \mathcal{O}(\log(n-t)) + \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(\log n)$$

Time, as required.

Note 2.1:

Since this algorithm works for any linear recurrence, by extension it works for the specific linear recurrence given in the question.
Or, for an "explicit" algorithm for this specific recurrence, we can define:

$$\mathrm{linearMatMake}(\alpha_1, \ldots, \alpha_{\mathrm{t}}) \to \begin{pmatrix} \alpha_1 & \cdots & \alpha_t \\ 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix}$$

Which takes $\mathcal{O}\left(t^2\right) = \mathcal{O}(1)$ time, so it doesn't affect the time complexity of the algorithm.
So to compute this specific linear reccurrence we'd need to compute:

$$\mathrm{matMult}\left(\mathrm{pow}(\mathrm{linearMatMake}(0, 2, -3), \mathrm{n}), \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}\right)_1$$

■

Question 2.2:

We place a set of points on the line $y = 0$: $\{p_1, \ldots, p_n\}$ and a set of points on $y = 1$: $\{q_1, \ldots, q_n\}$. We connect every point p_i with its respective q_i. Formulate an algorithm to compute the number of intersections between these lines in $\mathcal{O}(n \log n)$ time.

Answer:

We will split this algorithm into two parts: the initialization and the "meat".

Initialization

Before we can get to the meat of the algorithm, we first will order the set $\{p_1, \ldots, p_n\}$[1]. We can do this by mergesorting it (we want to preserve the value of p_i and i itself, so we'll mergesort the values (p_i, i) where the key is p_i), which takes $\mathcal{O}(n \log n)$ time. After mergesorting it, we can create an array P such that $P[i]$ gives the index of p_i in the order (this is why we needed to preserve both p_i and i).
Next, we mergesort $\{q_1, \ldots, q_n\}$ (we must also preserve the indexing, so we store both q_i and i). Now suppose the sorted array is:

$$[(q_{m_1}, m_1), (q_{m_2}, m_2), \ldots, (q_{m_n}, m_n)]$$

(Meaning $q_{m_1} < q_{m_2} < \cdots < q_{m_n}$)
We then iterate over this array, constructing the following array:

$$\sigma = [P[m_1], \ldots, P[m_n]]$$

This corresponds to the permutation created by the sets of points. What this means is that if the ordering is:

$$[p_4, p_3, p_2, p_1] \text{ and } [q_2, q_4, q_1, q_3]$$

Then P will be:

$$[4, 3, 2, 1]$$

So σ will be:

$$\sigma = [3, 1, 4, 2]$$

Essentially, σ represents the permutation of the points q_i relative to the points p_i. That is, if we think of the ordering of $\{p_i\}$ as the numbers 1 until n (which is $P[i]$), then we think of q_i as the same number as q_i (so we think of q_i as $P[i]$), and we are left with the permutation.
The reason why this is good is because every intersection is represented as an inversion in σ ($i < j$ form an inversion if $\sigma(i) > \sigma(j)$). This is because an intersection is just an iversion in the order of q_is relative to the order of p_is. So the line that connects p_i and q_i (ℓ_i) intersects ℓ_j if and only if there is an inversion between $P[i]$ and $P[j]$ in σ.

The time complexity of the initialization is $\mathcal{O}(n \log n)$ since it mergesorts twice ($\mathcal{O}(n \log n)$) then iterates over the arrays a constant amount of times ($\mathcal{O}(n)$), so all in all the time complexity is

$$\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n)$$

The meat

This step is more straightforward, we just need to calculate the number of inversions in σ. The meat function will return this and sort σ.
We can do this by splitting the array σ into two subarray σ_1 and σ_2. We know that the number of inversions of σ is equal to the number of inversions in σ_1 and σ_2, plus the number of inversions across them (elements in σ_1 that are greater than elements in σ_2).
In order to compute this, we will perform meat on σ_1 and σ_2. This will compute the inversions in σ_1 and σ_2 and sort them. Then we iterate over σ_1 and σ_2 like so:

- We start two pointers, i and j at the beginning of σ_1 and σ_2 respectively.
- We increment j until it reaches an index such that $\sigma_2[j] > \sigma_1[i]$ or until j reaches the end of σ_2, all the while incrementing the number of inversions. (All the while append $\sigma_2(j)$ into the auxillary array for mergesort).

[1] We must first convert this to a list of some sort which takes $\mathcal{O}(n)$ time.

- Once we reach a point where $\sigma_2[j] > \sigma_1[i]$, append $\sigma_1(i)$ to the auxillary array for mergesort, increment i, and double the inversion counter if i is not yet pointing at the end of σ_1 (as any inversion found for $i = 1 \dots x$ is also an inversion for $i = x + 1$).
- Once i reaches the end of σ_1, stop.

This process just iterates over σ, so it takes $\mathcal{O}(n)$ time.
Iterate over whatever is left of σ_1 and σ_2 and append it to the auxillary array. Then copy the auxillary array to σ. This will order σ.
We then add $\text{meat}(\sigma_1)$ and $\text{meat}(\sigma_2)$ to the increment counter and return that. All in all the time complexity for this process is:

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

Which, by the master theorem, has a time complexity of $\mathcal{O}(n \log n)$.

So all in all, this algorithm has a time complexity of $\mathcal{O}(n \log n)$, as required.

■

Question 2.3:

$A[1 \dots n]$ is an array of distinct integers. We know that it is cyclicly-sorted, that is it is sorted and then shifted k positions right cyclicly. Formulate an algorithm to compute k.

Answer:

Notice that if k is the cyclic degree, then shifting A left by k positions will sort A. That means $A[k]$ will become the last element ($k \mapsto 0$ during the shift, and since it is cyclic, $0 \mapsto n$), and the last element in A shifted is the maximum element since A shifted is sorted.
So this problem is equivalent to computing the index of the maximum element in A. Furthermore, since A is shifted, we know:

$$A[k+1] < \dots < A[n] < A[1] < \dots < A[k]$$

We can prove this quite simply. Let A_k be A shifted left k elements, which means $A_k[i] = A[i + k \bmod n]$. We know:

$$A_k[1] < \dots < A_k[n]$$

So:

$$A[k+1] < \dots < A[n] < A[1] < \dots A[k]$$

Notice that if we split A into two subarrays, the maximum of A is the maximum of the maximums of both subarrays. That is to say that if A_1 and A_2 are two subarrays:

$$\max(A) = \max\left(\max(A_1), \max(A_2)\right)$$

So naively, we can just compute the maximum of both subarrays and take the maximum of these. This would take:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

Time, which is just $\mathcal{O}(n)$ time. This isn't optimal, as a simpler solution would have been to just to iterate over the entire array which is also $\mathcal{O}(n)$ time (and is in fact actually quicker).
The trick is to recall that A is actually sorted, albeit shifted. So what we can do is compare the first element of A_1 and A_2, which are the first and second halves of A respectedly. We know that

- If A_1 contains the index k, then it has all indexes up to k ($[1, k]$), and $A_1[1] = A[1]$, which means that $A_2[1]$ is equal to $A[i]$ for some $i \notin [1, k]$, which we know is less than $A[1]$.
- If A_2 contains the index k, let its first index be $A[i]$ (that is, $A_2[1] = A[i]$), we know that $i \in [1, k]$, so $A_1[1] = A[1] < A[i] = A_2[i]$.

So, A_i contains the index k if and only if $A_i[1] > A_j[1]$.
So if $A_i[1] > A_j[1]$, then we just need to compute the index of the maximum of A_i, and return that (the maximum index must be relative to A).

```
Function ComputePivot(A, l, h)
    if h == l then
        return h;
    end
    if A[l] > A[⌊(l+h)/2⌋ + 1] then
        return ComputePivot(A, l, ⌊(l+h)/2⌋);
    else
        return ComputePivot(A, ⌊(l+h)/2⌋, h);
    end
end
```

We can then compute k by `ComputePivot(A, 1, A.len)`.
This algorithm has a time complexity of

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1)$$

Which, by the master theorem, is in $\mathcal{O}(\log n)$.

■

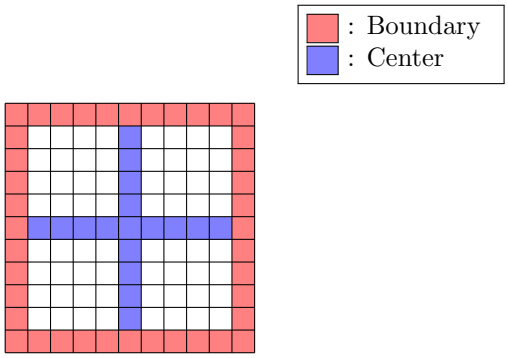
Question 2.4:

A 2-dimensional array $A[1 \dots n, 1 \dots n]$ represents a topological map. When it rains, the rain makes it way to local minima: points $A[i, j]$ which are less than all its neighbors (not including daigonals). Write an algorithm that finds a minima like this.

Answer:

The naive approach is to first check if the center is a local minima, if it is, return it. Otherwise, split the array into four quarters, and find recursively find the local minima of that array.
While this seems like it would work, it is flawed. Why is that? The issue is with the borders of the quarters. If the local minimum found is on the border of the subarray, then we still don't know if it is in fact a local minimum: it has adjacent elements which aren't in its subarray.
So we need to come up with some idea that involves the borders of the subarrays. The idea is to somehow choose the best subarray to find the local minimum in.
The algorithm is as follows:

- Iterate over the middle row, middle column, and boundaries of the array, and compute its minimum.
- If the minimum is a local minimum in the subarray, return it.
- Otherwise, the minimum of the boundaries/centers, has a minimum neighbor. Recurse over the subarray which contains the minimum neighbor (and also the minimum element on the boundaries/center).



Let A_i be the ith subarray, so A_0 is the initial array, and then A_1 is the subarray which contains the minimum of the boundary/center.
Let m_i be the minimum point on the boundary/center of A_i.
In order to prove that this algorithm works, we need to prove the following:

Proposition 2.1:

The algorithm terminates.

Proof:

At each recursive step of the algorithm, we are restricting our view to a quarter of the previous array. So at some point we will reach an array that will automatically give us a local minimum without recursion ($n \leq 3$). So the algorithm terminates.

■

Remember that by the definition of the algorithm, $m_i \in A_{i+1}$, in fact it is on the boundary of A_{i+1} (this isn't something new, this is the definiton of the algorithm).

Proposition 2.2:

$\{m_i\}_{i=0}^{n}$ is a decreasing series.

Proof:

m_{i+1} is the minimum of the boundary/center of A_{i+1}, and we know that m_i is on the boundary of A_{i+1}, so $m_{i+1} \leq m_i$.

■

Proposition 2.3:

The algorithm returns a local minimum. (The algorithm works).

Proof:

Suppose the algorithm terminates, returning m_i.
Suppose for the sake of a contradiction that m_i is not a local minimum of A_0. This can only happen if m_i is on the boundary of A_i. Since if m_i was on the center of A_i, then all of its neighbors would also be in A_i, and m_i is a local minimum of A_i, so it is then less than all of its neighbors, and is therefore a local minimum of A_0.
So m_i is on the boundary of A_i. And we know that the boundary of A_i is part of the boundary/center of A_{i-1}. And we know that the minimum of the boundary/center of A_{i-1} is m_{i-1}, so $m_{i-1} \leq m_i$, but $\{m_i\}$ is decreasing, so $m_{i-1} = m_i$ (so they're at the same index, as all elements are distinct).
Because m_i isn't a local minimum, it is larger than one of its neighbors. But we know that the minimum neighbor of m_{i-1}, which we proved is m_i, is in A_i, and m_i is a local minimum of A_i, so it is smaller than its smallest neighbor. This means it is smaller than all of its neighbors, and thus is a local minimum, in contradiction.

■

The complexity of the algorithm is:

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

As in order to compute the minimum of the boundary/center, it takes $\mathcal{O}(n)$ operations (since there are $2n + 4(n-2) = 6n - 8$ elements on the boundaries/center), and every subarray we look at is half the size of the previous array.
By the master theorem, this has a complexity of:

$$T(n) \in \mathcal{O}(n)$$

■