1. **Exponentiation Pseudocode**

```
function Exp(x, n):
    # computes x^n for integer n ≥ 0
    if n == 0:
        return 1
    let half =Exp(x, ⌊n/2⌋)
    let sq =half * half
    if n mod 2 == 0:
        return sq
    else:
        return x * sq
```

2. **Inductive Proof for Horner's Rule**
   Base Case (n = 0):

   A degree-0 polynomial is just the constant a_0.

   - The routine returns a_0 when n = 0, so it's correct for this case.

   Inductive Hypothesis:

   - Assume that for any polynomial of degree n - 1, Horner's routine correctly evaluates it at x.

   Inductive Step:

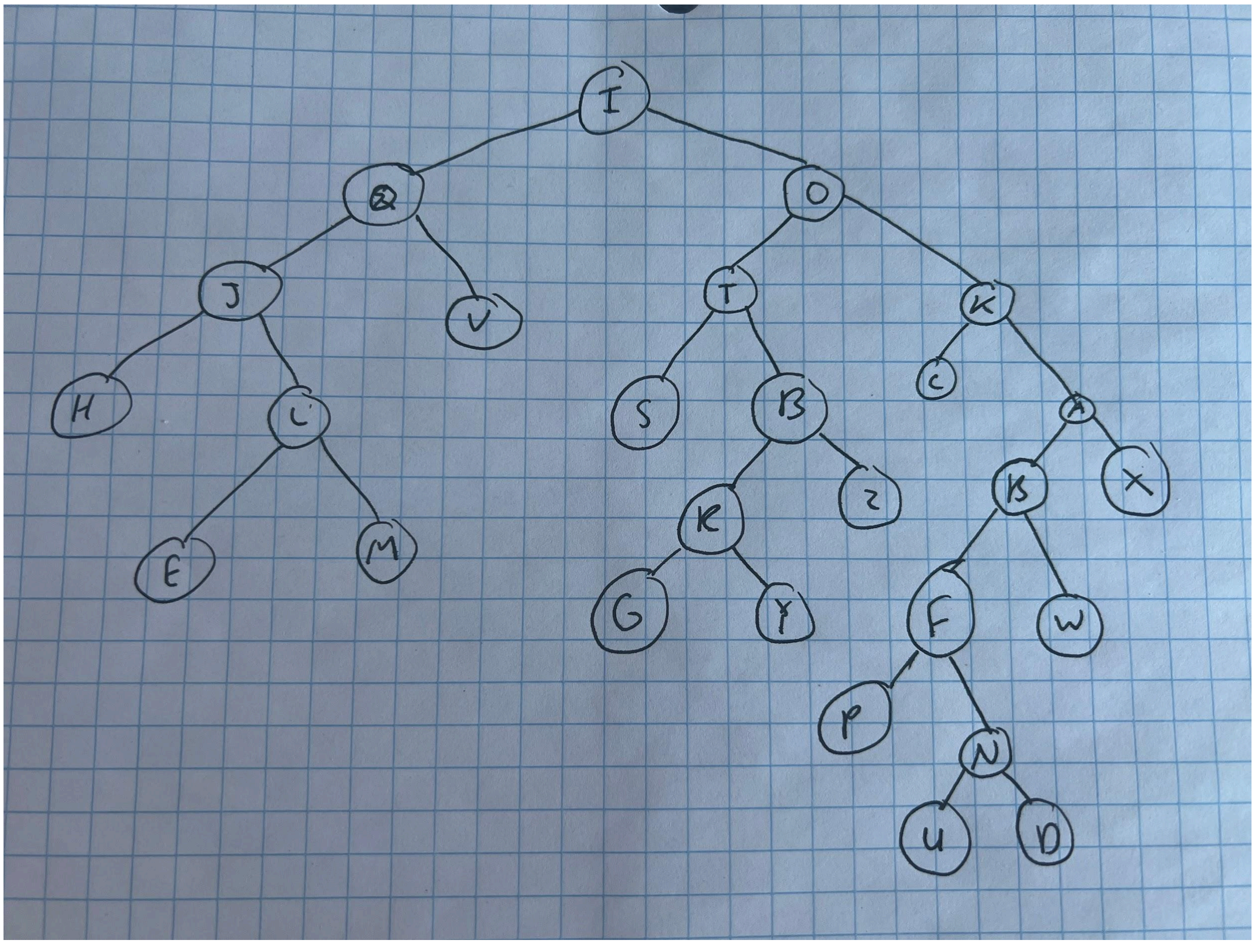
   Write the degree-nnn polynomial as
   $p(x) = a_0 + x(a_1 + a_2x + ... + a_nx^{n-1}) = a_0 + x * q(x),\ where\ q(x)\ has\ degree\ n - 1$

   By the inductive hypothesis, the call
   $horner(x, [a_1, ..., a_n], n - 1)\ correctly\ computes\ q(x)$

   Therefore the routine's final step, returning $a_0 + x * q(x)$, must correctly compute $p(x)$

   By induction, Horner's method evaluates any degree-n polynomial correctly.

3. a) **This is the binary tree that represents the pre-order and post-order sequence**



b) **Recursive Reconstruction Algorithm**

```
function BuildTree(pre[1..n], post[1..n]):

    # pre and post are nonempty, same size, tree is full

    let rootVal =pre[1]

    create node root with value rootVal

    if n == 1:

        return root

    # the root of the left subtree is pre[2]

    let Lroot =pre[2]

    # find Lroot in post to determine left-subtree size
```

```
let i such that post[i] == Lroot   # 1 ≤ i < n

let leftSize =i                    # number of nodes in left subtree

# recursively build left and right subtrees

root.left  <- BuildTree(pre[2 .. 1+leftSize],

                        post[1 .. i])

root.right <- BuildTree(pre[2+leftSize .. n],

                        post[i+1 .. n−1])

return root
```

c) Proof of Correctness

- Existence of split: Since the tree is full and pre[2] is the left-subtree root, it must appear somewhere in post[1..n−1].
- Correct sizes: In a full tree with nnn nodes, both subtrees have odd sizes n_L, n_R with $n_L + n_R + 1 = n$. Locating pre[2] at post[i] ensures exactly i nodes in the left subtree.
- Induction: On a tree of size nnn, assume recursion works for all smaller sizes. Splitting at the correct i gives exactly the original left- and right-subtrees in both traversals, so recursive calls reconstruct them correctly.

4. **D + C Maximum Subarray**
   a) **Pseudocode**

```
function MaxCrossingSubarray(A, low, mid, high):

    leftSum = −∞; sum = 0; maxLeft = mid

    for i from mid down to low:

        sum = sum + A[i]

        if sum > leftSum:

            leftSum = sum; maxLeft = i

    rightSum = −∞; sum = 0; maxRight = mid+1

    for j from mid+1 to high:
```

```
            sum = sum + A[j]
            if sum > rightSum:
                rightSum = sum; maxRight = j
        return (leftSum+rightSum, maxLeft, maxRight)
function MaxSubarray(A, low, high):
        if low == high:
            # base: single element
            return (A[low], low, high)
        mid = ⌊(low + high)/2⌋
        (lSum,  lLow,  lHigh ) = MaxSubarray(A, low, mid)
        (rSum,  rLow,  rHigh ) = MaxSubarray(A, mid+1, high)
        (cSum,  cLow,  cHigh ) = MaxCrossingSubarray(A, low, mid, high)
        # return whichever is largest
        if lSum ≥ rSum and lSum ≥ cSum:
            return (lSum,  lLow,  lHigh)
        else if rSum ≥ lSum and rSum ≥ cSum:
            return (rSum,  rLow,  rHigh)
        else:
            return (cSum,  cLow,  cHigh)
```

b) **Proof of Correctness**

Proceed by induction on n = high - low + 1:

- Base n = 1: MaxSubarray returns the single element, which trivially is the max subarray.
- Inductive step: Assume correct for all sizes < n. Let the maximum subarray of A[low..high] be either entirely in the left half, entirely in the

right half, or crossing the midpoint.

- Recursively we obtain the correct best subarrays in the left and right halves.
- MaxCrossingSubarray correctly finds the best crossing subarray by scanning outward from the center.
- Taking the maximum of these three candidates yields the global maximum.

Thus by induction, MaxSubarray(A,1,n) correctly returns the maximum‑sum subarray.

Generative AI (OpenAI - o4-mini-high):

Input: The following is a recursive formula for exponentiation:

xn = (x⌊n/2⌋2 if n is even

x · x⌊n/2⌋2 if n is odd

Turn this formula into a recursive algorithm by writing pseudocode that takes two numbers x and n as input and returns the value of x^n.

Output:

plaintext Copy Edit

```
Function Power(x, n):
    # Computes xⁿ for integer n ≥ 0 in O(log n) time
    if n == 0 then
        return 1
    end if

    # Recursively compute x^(⌊n/2⌋)
    y ← Power(x, ⌊n / 2⌋)

    # Square the half-power
    if n mod 2 = 0 then
        return y * y
    else
        # If n is odd, multiply by one extra x
        return x * y * y
    end if
End Function
```