Here is the exact pseudocode from the PDF for each section, followed by the explanation in the **Input** → **Process** → **Output** format.

4.1 Scan

This algorithm calculates the **prefix sums** (running totals) of an array.

The Pseudocode (Figure 2)

```
scanUp(A, L, f) =
  if (|A| = 1) then return A[0]
  else
    n ← |A|;
    m ← n/2;
    l ← scanUp(A[0 : m], L[0 : m - 1], f)
||
    r ← scanUp(A[m : n], L[m : n - 1],
f);
    L[m - 1] ← l;
    return f(l, r)
```

```
scanDown(R, L, f, s) =
  if (|R| = 1) then R[0] = s; return;
  else
    n ← |A|;  // Note: Context implies
this uses the size of the current slice
    m ← |R|/2;
    scanDown(R[0 : m], L[0 : m - 1], s) ||
    scanDown(R[m : n], L[m : n - 1], f(s,
L[m - 1]));
    return

scan(A, f, I) =
  L ← array[|A| - 1];
  R ← array[|A|];
  total ← scanUp(A, L, f);
  scanDown(R, L, f, I);
  return <R, total>;
```

Explanation

1. Input

- `A` : An array of numbers, e.g., $[1, 2, 3, 4]$.
- `f` : A function to combine them (usually addition `+`).

- `I`: An identity element (usually `0` for addition).

2. Process

This approach uses a logical tree structure.

- **Step 1 (`scanUp`):** The "Summing Phase."
  - The algorithm recursively splits the array in half until it hits single numbers.
  - It adds pairs of results together (`f(l, r)`).
  - It saves partial sums in the array `L` (think of `L` as the internal nodes of a tree).
- **Step 2 (`scanDown`):** The "Passing Phase."
  - It takes a value `s` (the sum coming from the left/parent) and pushes it down.
  - **Left Child:** Inherits `s` directly (because it's the start of the sequence).
  - **Right Child:** Inherits `s` + the sum of the Left Child (stored in `L`).

3. Output

- `R`: An array of prefix sums. For input $[1, 2, 3, 4]$, `R` is $[0, 1, 3, 6]$.
- `total`: The sum of the whole array (10).

4.2 Filter

This algorithm removes elements that don't fit a certain rule.

The Pseudocode (Figure 3)

```
FILTER(A, p) =
  n ← |A|;
  F ← array[n];
  parfor i in [0 : n]
    F[i] ← p(A[i])
  <X, m> ← plusScan(F);
  R ← array[m];
  parfor i in [0 : n]
    if (F[i]) then R[X[i]] ← A[i]
  return R
```

Explanation

1. Input

- `A` : A list of items.
- `p` : A "predicate" (a true/false question). Example: "Is the number odd?"

2. Process

- **Line 4-5 (`parfor`):** Every processor checks one item. If `p(A[i])` is true, they write `1` into array `F`. If false, they write `0`.
- **Line 6 (`plusScan`):** We run a scan on `F`. This calculates the *destination index* for every surviving item.
    - Example `F`: $[1, 0, 1]$.
    - Scan `X`: $[0, 1, 1]$. (Item 0 goes to index 0. Item 2 goes to index 1).
- **Line 8-9 (`parfor`):** Processors look at the flags again. If the flag is `1`, they copy the item from `A` into the new array `R` at the position calculated by the scan `X[i]`.

3. Output

- `R`: A condensed array containing only the items that returned `true`.

4.2 Flatten

This algorithm turns a "list of lists" into one simple list.

The Pseudocode (Figure 4)

```
FLATTEN(A) =
  sizes ← array(|A|);
  parfor i in [0 : |A|]
    sizes[i] ← |A[i]|
  <X, m> ← plusScan(sizes);
  R ← array(m);
  parfor i in [0 : |A|]
    o ← X[i];
    parfor j in [0 : |A[i]|]
      R[o + j] ← A[i][j]
  return R
```

Explanation

1. Input

- `A`: A nested array, e.g., `[[10, 20], [30], [40, 50]]`.

2. Process

- **Lines 2-4:** Calculate how big each sub-list is. Result: `[2, 1, 2]`.
- **Line 5 (`plusScan`):** Calculate starting positions (offsets). Result `X`: `[0, 2, 3]`.
    - Sub-list 1 starts at 0.
    - Sub-list 2 starts at 2.
- **Lines 7-10:** A double loop. The outer loop `i` runs for each sub-list. The inner loop `j` runs for each item in that sub-list.
    - It copies the item `A[i][j]` to `R[o + j]` (Offset + current position).

3. Output

- `R`: One single array `[10, 20, 30, 40, 50]`.

4.3 Search

This finds where a value belongs in a sorted list.

The Pseudocode (Figure 5)

```
findBlock(A, v, k) =
  s ← |A|/k;
```

```
  r ← k;
  parfor i in [0 : k]
    if (A[i * s] < v and A[(i + 1) * s] >
v)
    then r ← i
  return (A[r * s, (r + 1) * s], i * s)

search(A, v, k) =
  (B, o) = findBlock(A, v, min(|A|, k));
  if (|A| ⩽ k) then return o
  else return o + search(B, v, k);
```

Explanation

1. Input

- `A` : A sorted array.
- `v` : The value we are looking for.
- `k` : The number of splits (how many processors we can use).

2. Process

- `findBlock` **:** Instead of checking just the middle (like Binary Search), we check `k` positions evenly spaced out.

    - The `parfor` checks all `k` spots at the same time.
    - It identifies which "block" (slice) of the array holds the value `v`.
- `search`**:** This is the main loop.
    - It calls `findBlock` to narrow down the array to a smaller slice `B`.
    - It adds the offset `o` to keep track of the total position.
    - It recurses (calls itself) on the smaller slice `B`.

3. Output

- An integer index representing where `v` is (or should be) in `A`.

4.4 Merge

This combines two sorted arrays into one.

The Pseudocode (Figure 7)

```
MERGE(A, B, R) =
  case (|A|, |B|) of
  (0, _) ⇒ copy B to R; return;
  (_, 0) ⇒ copy A to R; return;
  otherwise ⇒
    m ← |R|/2;
    (ma, mb) = KTH(A, B, m);
    MERGE(A[0 : ma], B[0 : mb], R[0 : m])
||
    MERGE(A[ma : |A|], B[mb : |B|], R[m :
|R|]);
    return
```

Explanation

1. Input

- `A` and `B`: Two arrays that are already sorted.
- `R`: An empty result array.

2. Process

- **Line 6 (`m`):** We pick the midpoint of the final result array.
- **Line 7 (`KTH`):** We calculate how many elements from `A` (`ma`) and how many from `B` (`mb`) are

needed to fill the first half of `R`. (This `KTH` helper function is complex, but essentially it finds split points).

- **Lines 8-9 (`||`):**
  - **Left Processor:** Merges the small halves of A and B into the left half of R.
  - **Right Processor:** Merges the big halves of A and B into the right half of R.
  - These run at the exact same time.

3. Output

- `R`: The final array containing all elements from A and B, perfectly sorted.

4.5 K-th Smallest

This finds the element that would be at index k if the array were sorted.

The Pseudocode (Figure 9)

```
kthSmallest(A, k) =
  p ← selectPivot(A);
```

```
  L ← filter(A, \x.(x < p));
  G ← filter(A, \x.(x > p));
  if (k < |L|) then
    return kthSmallest(L, k)
  else if (k > |A| - |G|) then
    return kthSmallest(G, k - (|A| - |G|))
  else return p
```

Explanation

1. Input

- `A`: A scrambled (unsorted) array.
- `k`: The rank we want (e.g., 0 for min, $|A|/2$ for median).

2. Process

- **Line 2:** Pick a "Pivot" number `p` (usually random).
- **Lines 3-4:** Use the `filter` function (from 4.2) to create two new arrays:
  - `L`: All numbers smaller than `p`.
  - `G`: All numbers bigger than `p`.
- **Lines 5-9:** Logic check:

- If `k` is smaller than the size of `L`, the answer must be inside `L`. We recurse on `L`.
- If `k` is very large (larger than everything except `G`), the answer is in `G`. We recurse on `G` (subtracting the count of the numbers we skipped).
- Otherwise, the answer is the pivot `p` itself.

3. Output

- The single value that represents the k-th smallest number.