Comments for pseudocode requirement in this assignment:

In this assignment, we require you to provide pseudocode or pseudocode-like verbal description for your algorithms. For those who do not have much experience in CS, here I copy the definition from Wikipedia:

Pseudocode is an informal high-level description of the operating principle of a computer program or other algorithm. It uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Pseudocode typically omits details that are essential for machine understanding of the algorithm, such as variable declarations, system-specific code and some subroutines. The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation. The purpose of using pseudocode is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm.

Hence, you are not required to give all details about your algorithm, but you need to make sure that the pseudocodes are human-understandable. For example, you do not need to declare a variable; instead, you should make sure that the reader can understand what this variable represents for (try not to use variables like $a$, $x$, $S$ without any explanation). Since we will judge your solution by both correctness and clearness, some points will be taken off if we cannot figure out what you tried to describe.

Here I provide an example (of a certain kind of sorting algorithm), and this level of details are considered to be sufficient as a pseudocode.

---

**Algorithm 1** ASYMMETRIC-PRAM SORT

**Input:** An array of records $A$ of length $n$

1: Select a sample $S$ from $A$ independently at random with per-record probability $1/\log n$, and sort the sample.

2: Use every $(\log n)$-th element in the sorted $S$ as splitters, and for each of the about $n/\log^2 n$ buckets defined by the splitters allocate an array of size $c \log^2 n$.

3: In parallel locate each record’s bucket using a binary search on the splitters.

4: In parallel insert the records into their buckets by repeatedly trying a random position within the associated array and attempting to insert if empty.

5: Pack out all empty cells in the arrays and concatenate all arrays.

6: For round $r ← 1$ to $2$ do

   for each array $A'$ generated in previous round
   
   Deterministically select $|A'|^{1/3} - 1$ samples as splitters and apply integer sort on the bucket number to partition $A'$ into $|A'|^{1/3}$ sub-arrays.

7: For each subarray apply the asymmetric RAM sort.

8: Return the sorted array.
1 Median

(a)  
\[ W(n) = W(n/5) + W(7n/10) + O(n) \]
\[ S(n) = S(n/5) + S(7n/10) + O(\log n) \]

(b) It is linear work since it is root dominated. The second level of the tree does 9/10th as much work as the top level, and so on.

(c) It is leaf dominated, but hard to calculate the number of leaves. To solve, guess a solution of the form \( k_1 n^\alpha - k_2 \log n - k_3 \). Plug this into the recurrences. The log and constant terms can be made to cancel by picking the \( k \)s right (as done in class notes), leaving \( k_1 n^\alpha = k_1 (n/5)^\alpha + k_1 (7n/10)^\alpha \). By cancelling \( n \) and \( k_1 \) this leaves \( 1 = (1/5)^\alpha + (7/10)^\alpha \), and now solve for \( \alpha \), which gives about \( 0.8397803 \). So the solution is \( O(n^{0.8397803\ldots}) \).

2 Parallel Merging

(a) Call the two arrays \( A \) (of length \( n \)) and \( B \) (of length \( m \)).

First, notice that if the (1-based) index of the cut point in \( A \) is \( i \), then the index in \( B \) must be \( k - i \).

Also, the correct \( A \) cut point \( i \) satisfies \( A_i < B_{k-i+1} \), \( B_{k-i} < A_{i+1} \). Thus, if \( A_i > B_{k-i+1} \), then \( i \) is too large; if \( B_{k-1} > A_{i+1} \), then \( i \) is too small.

Since the arrays are sorted and have distinct elements, exactly one of these three conditions must hold.

Furthermore, \( 0 \leq i \leq k \). So just run binary search over \( i \), which takes \( O(\log n) \) time (really, \( O(\log k) \)).

(b) There are (at least) two solutions to this problem.

Divide into sequential

First, break the two arrays into \( \frac{n}{\log n} \) pieces by running the \( k \)th-smallest algorithm of part (a) that many times. That is, find the \( \log n \) cut point, the \( 2 \log n \) cut point, the \( 3 \log n \), and so on, in parallel. Because part (a) runs in \( O(\log n) \) work and span, this part has \( O(\log n) \) span and \( O(\log n \frac{n}{\log n}) = O(n) \) work.

Thus we have \( \frac{n}{\log n} \) pairs of arrays, so that the first pair contains the smallest \( 2 \log n \) members of the two arrays, the second pair the next-smallest \( 2 \log n \) members, and so on.

Now, in parallel, merge each pair using the standard sequential merge algorithm, placing it into the appropriate indices in the final array. Each component again takes \( O(\log n) \) work and span, so this step also takes \( O(\log n) \) span and \( O(n) \) work.
Divide and Conquer

Find the $\sqrt{n}$, $2\sqrt{n}$, $3\sqrt{n}$, … cut points in parallel so that the arrays are broken into $\sqrt{n}$ pieces. Then recurse. Once you reduce down to a single element, you also know that element’s position in the final list; simply place it there. Since we do $\sqrt{n}$ binary searches as described in part (a), the work of this algorithm is:

$$W(n) = \sqrt{n} W(\sqrt{n}) + O(\sqrt{n} \log n)$$

To prove that this is linear work assume that $W(n) \leq k_1 n - k_2 n^{1/2}$ and prove by induction (substitution).

$$W(n) \leq n^{1/2} (k_1 n^{1/2} - k_2 n^{1/4}) + n^{1/2} \log n$$

$$= k_1 n - k_2 n^{3/4} + n^{1/2} \log n$$

$$\leq k_1 n - k_2 n^{1/2}$$

The last step holds since we can find a $k_2$ such that $k_2 n^{3/4} \geq k_2 n^{1/2} + n^{1/2} \log n$ for all $n$ greater than some small integer constant. The values of $k_1$ and $k_2$ have to be determined by the base case.

The span of this algorithm is, if we let $\gamma$ be a constant such that the work of part (a) is at most $\gamma \log n$:

$$S(n) \leq \gamma \log n + \gamma \log n^{1/3} + \gamma \log n^{1/4} + \ldots$$

$$= \gamma \log n \left( 1 + \frac{1}{2} + \frac{1}{4} + \ldots \right)$$

$$\leq 2 \gamma \log n,$$

which is $O(\log n)$.

3 Euler Tours

(a) List ranking can also be used to do list prefix sum. In fact the recursive version shown in class after one level of recursion is already doing a sum.

Associate each edge with the node below it. Each down edge grabs the node value and each up edge uses 0. Do a list prefix sum on this. Subtract the down edge from the up edge. This is the desired value since it accounts for all the values added while traversing the subtree, but excluding the value when entering the subtree. It takes linear work and logarithmic span.

This can also be done directly with list ranking by first list ranking, and then moving each link to position $i$ in an array where $i$ is the rank, doing a plus scan on values as above, and again taking the difference.

(b) First consider generating a pointer from each word $i$ to the word that would start the next line if word $i$ were at the beginning of a line. Note that this pointer structure forms a forest—every word points to at most one other word, and every word can be pointed to by many other words. By adding an artificial word of length $L$ at the end, we can make it a tree. Let’s call this the break tree. Now we consider three steps: first making the break tree, then converting it into a Euler tour, and finally using the Euler tour to identify the line starts.
(a) To make the tree, let \( S \) (for Start) be the result of a plus scan on \( W \). Now create the two arrays
\[ E[i] = S[i] + W[i] \] (for End of word) and \( N[i] = S[i] + L \) (for start of Next line). To generate the break tree, for each \( i \) we could binary search the value \( N[i] \) in the array \( E[i] \) to find the parent. However, this would require \( O(n \log n) \) work. Instead, merge \( E \) and \( N \) into \( M \) (for Merged). Ties are broken so elements from \( E \) go first. Let’s call the elements from \( E \) and \( N \) in \( M \) black and white, respectively. Note that for each white element in \( M \) its parent in the break tree is the next black element. We have effectively got the same result as the binary searches but in linear work. We can pass the parent index to its children, by creating an array
\[ A[i] = \begin{cases} i & \text{black} \\ -\infty & \text{white} \end{cases} \]
and doing a backwards min-scan on it. Each white element will get the index of the next black element in \( M \), i.e. its parent. Note that each black (white) element can easily find its matching white (black) element: when doing the merge, keep the original word location as auxiliary data, and rendezvous at that location.

(b) To convert to an Euler tour, we make each black element in \( M \) the down edge for the node below it (the word it corresponds to), and each white element an up edge for the node below it. We need to link these to form the Euler tour. We assume the children of a black element (down edge), i.e., its immediate white predecessors in \( M \), are ordered right-to-left. If a black element (down edge) is a leaf (preceded by another black element) it points to its own white element (its up edge), otherwise it points to the black element corresponding to the previous white element in \( M \) (the down edge of its first child). If a white element (up edge) \( i \) has a preceding white element \( j \), it points to \( j \)’s black element (the down edge of its next sibling), otherwise it is the last child and points to \( i \)’s parent’s white element (the up edge of its parent). The element \( i \) knows its parent from the backwards min-scan. The list is formed.

(c) To use the tree to find the breaks, place a 1 at the very first word and 0 elsewhere. Calculate for each node in the tree the sum of its subtree. If your result is 1 you are the first word on a line, and otherwise you are not.

All steps take linear work and logarithmic span.

4 Buffered B-tree

(a) The nodes split in the same way as in a B-tree or buffer tree, and hence all leaves are at the same height. Since each internal node (except the root) has \( \Theta(B^c) \) children, the height is \( O(\log_{B^c} N) = O(\frac{\log_B N}{\epsilon}) \). Observing that each leaf contains \( \Theta(B) \) objects, and hence there are only \( N/B \) leaves, yields a bound of \( O(\frac{\log_B(N/B)}{\epsilon}) \) on the height. These two bounds are asymptotically equivalent if \( N > B^2 \), and hence either is acceptable.

(b) The cost of inserting into a leaf that is already in memory is \( O(1/B) \). The analysis for this is the same as in a B-tree.

(c) Insertions into the root is free unless the root buffer overflows and things move down the tree. Whenever objects are moved from a parent node down to its child node, which costs \( \Theta(1) \) block transfers to
load both nodes, $\Omega(B^{1-\epsilon})$ objects are moved. Thus, the amortized cost of moving each object is $O(1/B^{1-\epsilon})$. Since an object moves once at each height in the tree, the total cost is $O(\frac{\log_B N}{\epsilon B^{1-\epsilon}})$.

(d) An inserted object first moves through buffers (counted in b) then eventually gets inserted into a leaf (counted in c). The cost of flushing the buffers already counts for the cost of loading the leaf, so the assumption in (b) that the leaf be in memory is fair.

Combining (b) and (c) yields $O(\frac{\log_B N}{\epsilon B^{1-\epsilon}})$.

(e) A search need only look in a single root-to-leaf path and the corresponding buffers. Since each buffer and node fits in a single block, the cost is 1 block transfer per height of the tree, for a total of $O(\frac{\log_B N}{\epsilon})$.

(f) The buffer tree does not allow for efficient searches — each buffer is unordered, so a search may have to load the entire buffer at each node, yielding a search cost of $O(\frac{M}{B} \log_{M/B} N) = \omega(M/B)$ for a buffer tree. The search cost for the buffered B-tree, on the other hand, is logarithmic.

(g) The buffered B-tree is worse than the B-tree for searches by a $1/\epsilon$ factor, but it is better for insertions by a $\epsilon B^{1-\epsilon}$ factor. If insertions are more frequent than searches, the buffered B-tree may be much better.