Serial Quicksort

Quicksort is an important sorting algorithm. Given an array \( A(1:n) \) of items to be sorted, \( \text{Quicksort}(i, j, A) \) sorts the items in positions \( i \ldots j \). To sort the entire array, the main program calls \( \text{Quicksort}(1, n, A) \).

\( \text{Quicksort}(i, j, A) \):

- If \( j \leq i \) then return
- \( \text{pivot} = A(i) \)
- Move items with keys \(<\text{pivot}\) to the front of this subarray of \( A \)
  - Move \( \text{pivot} \) to the position just after the smaller items (its final position). Let \( k \) be this position.
  - Move items with keys \( \geq\text{pivot} \) to positions \( k+1 \ldots j \)
- \( \text{Quicksort}(i, k-1, A) \)
- \( \text{Quicksort}(k+1, j, A) \)

The worst-case time is \( \Theta(n^2) \) since one might end up with, say, \( k = i \) at every step. In fact, that is what happens if the array was already sorted, and any good implementation would use something like \( \text{pivot} = A(\lfloor (i + j)/2 \rfloor) \). However, the expected case time is \( \Theta(n \log n) \), under the assumption that all orderings of the keys are equally likely (and that there are no duplicate keys).

The following implementation is atrocious but, in O-notation, is the same as a good one. In class you’ll see why it is written this way.

To determine where each item is moved to, let \( \text{smaller}(1:n) \) and \( \text{rank}(1:n) \) be integer arrays. For \( i \leq m \leq j \), if \( A(m) < \text{pivot} \) then \( \text{smaller}(m) = 1 \) else \( \text{smaller}(m) = 0 \). We use a \text{scan} operation, also called \text{parallel prefix}, to compute \( \text{rank} \): \( \text{rank}(m) = \sum_{\ell = i}^{m} \text{smaller}(\ell) \). Note that if \( A(m) < \text{pivot} \) then \( \text{rank}(m) \) is the number of items preceding \( m \), plus \( m \) itself, that are smaller than \( \text{pivot} \). For example, suppose \( i = 7, j = 16 \), and the values of \( A \) in these positions are

\[
27 \quad 15 \quad 25 \quad 29 \quad 11 \quad 87 \quad 3 \quad 98 \quad 13 \quad 8
\]

The pivot key is in position 7, with value 27. The values of \( \text{smaller} \) for positions 7 through 16 are

\[
0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1
\]

and the values of \( \text{rank} \) are

\[
0 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 4 \quad 4 \quad 5 \quad 6
\]

For positions where \( A < \text{pivot} \): the value in position 8 will be moved to position 7, the item in position 9 moves to 8, the item in position 11 moves to 9, etc. In general, the item at position \( m \) moves to \( i + \text{rank}(m) - 1 \). Here are the destinations of items where \( A < \text{pivot} \):

\[
\begin{array}{cccccccc}
7 & 8 & . & 9 & . & 10 & . & 11 & 12
\end{array}
\]

For positions where \( A \geq \text{pivot} \): the first one (i.e., the pivot) should go just after the last position where smaller items go, i.e., to \( i + \text{rank}(j) \). The next value \( \geq \text{pivot} \) goes to \( i + \text{rank}(j) + 1 \), etc. Here are the destinations of items where \( A \geq \text{pivot} \):

\[
\begin{array}{cccc}
\end{array}
\]

This moves the item at position \( m \) to \( m + \text{rank}(j) - \text{rank}(m) \). One way to derive this is to note that \( m \) is the \( m-i+1 \) position in the subarray, and \( m-i-\text{rank}(m) \) of the positions ahead of it are \( \geq \text{pivot} \), i.e., are going to positions \( i+\text{rank}(j), i+\text{rank}(j)+1, \ldots, i+\text{rank}(j)+m-i-\text{rank}(m)-1 \), and it will move to the next position.
After the items have been moved to their destinations, the values in \( A(7) \ldots A(16) \) are

\[
15 \quad 25 \quad 11 \quad 3 \quad 13 \quad 8 \quad 27 \quad 29 \quad 87 \quad 98
\]

Then recursive calls are made to \texttt{Quicksort}(7, 12, A) and \texttt{Quicksort}(14, 16, A), i.e., to \texttt{Quicksort}(i, i+rank(j)-1, A) and \texttt{Quicksort}(i+rank(j)+1, j, A)

Using the above, we can rewrite \texttt{Quicksort} as follows. We use an auxiliary array \( B \) so that we don’t change values of \( A \) before we’ve had a chance to move them.

\texttt{Quicksort}(i, j, A):

1. If \( j \leq i \) then return
2. pivot = \( A(i) \)
3. For all \( m \in i..j \) smaller(m) = \( A(m) < \text{pivot} \)
4. For all \( m \in i..j \) rank(m) = \( \sum_{k=i}^{m} \text{smaller(k)} \)
5. For all \( m \in i..j \)
6. if \( A(m) < \text{pivot} \) then \( B(i+\text{rank(m)}-1) = A(m) \)
7. else \( B(m+\text{rank(j)}-\text{rank(m)}) = A(m) \)
8. For all \( m \in i..j \) \( A(m) = B(m) \)
9. \texttt{Quicksort}(i, i+\text{rank(j)}-1, A)
10. \texttt{Quicksort}(i+\text{rank(j)}+1, j, A)

Reminder: this is a very bad way to write serial quicksort and don’t ever do it this way, even though in \( O-\text{notation} \) it is the same as a good implementation. However, it will help us create a parallel version.