Merging two adjacent sorted segments

To the left below are two adjacent sorted segments, \( b[h..e] \) and \( b[e+1..k] \). We want an algorithm to merge them in stable fashion into the single sorted segment \( b[h..k] \) shown to the right.

\[
\begin{array}{c|c|c}
  h & e & k \\
  \hline
  2 & 4 & 5 \\
  & 1 & 2 \ 3 \ 6 \ 7
\end{array}
\quad
\begin{array}{c|c|c}
  h & e & k \\
  \hline
  1 & 2 & 3 \ 4 \ 5 \ 6 \ 7 \ 7
\end{array}
\]

To do this, first copy \( b[h..e] \) into another array \( c[0..e-h] \), as shown below. We have written ? for values in \( b[h..e] \) not because values aren’t there but because we don’t care what is in that segment after the copy.

\[
\begin{array}{c|c|c}
  h & e & k \\
  \hline
\end{array}
\quad
\begin{array}{c|c|c}
  0 & e-h \\
  \hline
  2 & 4 & 4 & 5 & 7
\end{array}
\]

The goal now is to merge \( b[e+1..k] \) and \( c[0..e-h] \) in stable fashion into \( b[h..k] \). We show three steps. When an integer is moved, we replace it by ?

Start with this:

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\end{array}
\quad
\begin{array}{c|c|c|c|c|c|c|c|c|c}
  0 & e-h & 2 & 4 & 4 & 5 & 7
\end{array}
\]

Move smaller of \( b[e+1] \) and \( c[0] \):

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\end{array}
\quad
\begin{array}{c|c|c|c|c|c|c|c|c|c}
  0 & e-h & 2 & 4 & 4 & 5 & 7
\end{array}
\]

Move \( c[0] \), not \( b[e+2] \) (stable sort):

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\end{array}
\quad
\begin{array}{c|c|c|c|c|c|c|c|c|c}
  0 & e-h & ? & 4 & 4 & 5 & 7
\end{array}
\]

Move smaller of \( b[e+2] \) and \( c[1] \):

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c}
\end{array}
\quad
\begin{array}{c|c|c|c|c|c|c|c|c|c}
  0 & e-h & ? & 4 & 4 & 5 & 7
\end{array}
\]

That should be enough to give you the idea: At each iteration of a loop, the smallest unmoved (non-?) element in the two segments \( b[e+1..k] \) and \( c[0..e-h] \) is moved to the next available position (the first ?) in \( b[h..] \).

In order to write the loop, we need a loop invariant. We need three variables \( i, j, \) and \( m \) to indicate three positions in the arrays. We define them below; to the right we show them after the last move shown above.

1. The position \( i \) in which to place the next merged integer in \( b[h..] \).
2. The position \( j \) of the first unmoved value in \( b[e+1..k] \).
3. The position \( m \) of the first unmoved value in \( c[0..e-h] \).

The loop invariant has four parts:

- Invariant: \( b[h..i-1] \) contains the moved values, stably sorted,
- \( b[j..k] \) contains the unmoved values in \( b[e+1..k] \),
- \( c[m..e-h] \) contains the unmoved values in \( c[0..e-h] \),
- \( b[h..i-1] \leq b[j..k] \) and \( b[h..i-1] \leq c[m..e-h] \)

The algorithm is shown to the right. After truthifying the invariant by initializing \( i, j, \) and \( m \), a while-loop moves values as long as both segments \( b[j..k] \) and \( c[m..e-h] \) contain a value to move. This makes the code a bit easier to write and to read.

A second loop then moves remaining values in \( c[m..e-h] \). There is no need to move remaining values in \( b[j..k] \) because, if there are any, one can verify that they are already in the correct place at the end of \( b[j..k] \).

Space and time complexity

The time complexity is \( O(k+1-h) \). Extra space is used for array \( c \), so the space is \( O(e+1-h) \).

```c
int i, j, m = 0;
while (j <= k && m <= e-h) {
    if (c[m] <= b[j]) { b[i] = c[m]; m++; i++; }
    else { b[i] = b[j]; j++; i++; }
}
while (m <= e-h) {b[i] = b[m]; m++; i++; }
```

©David Gries, 2018