Earlier, we developed method \texttt{merge}, who spec is shown appears to the right. It merges two adjacent sorted segments of an array into a single sorted segment. It does so stably, meaning that two equal values remain in the same relative position. The method takes time $O(k+1-h)$ and uses space $O(e+1-h)$.

We use method \texttt{merge} to write recursive sorting method \texttt{mergeSort}, to the right. It is simple enough that its correctness needs little explanation.

To sort a whole array \texttt{c}, use the call
\begin{verbatim}
mergeSort(c, 0, c.length-1);
\end{verbatim}

Space complexity

Method \texttt{merge} requires an extra array of size $e+1-h$. Here, \texttt{e} is the average of \texttt{h} and \texttt{k}, so \texttt{mergeSort} requires space $O((k+1-h)/2)$ while the call on \texttt{merge} is being executed. That’s actually $O(k+1-h)$. The recursive calls also require space, but half as much, and not at the same time. So the space requirement is $O(k+1-h)$.

When sorting a whole array \texttt{c}, as shown above, \texttt{mergeSort} requires space $O(c.length)$. That is one drawback of \texttt{mergeSort}.

Time complexity

Consider using \texttt{mergeSort} to sort an array of size \texttt{n}. Two recursive calls are made, each to sort an array of size at most $\text{ceil}(n/2)$. Each of these recursive calls makes two further recursive calls on array segments of size at most $\text{ceil}(n/4)$, and so on.

The tree shown below depicts these recursive calls. Since at each level the size the segments being sorted is halved, the maximum depth of recursion is $\log_2 n$.

How much work is required at each level? At the top level, time $O(n)$ is required to merge the two adjacent segments. This is shown in the column on the right. At the next level, $O(n/2)$ time is required to merge two segments of size $n/4$; this is done twice, so the time required at the second level is $O(2*n/2)$, which is also $O(n)$. In the same way, it can be seen that the time required at each level is $O(n)$.

Since there are $\log n$ levels, the total time required by \texttt{mergeSort} is $n \log n$.

\begin{verbatim}
/** b[h..e] and b[e+1..k] are sorted. Stably * swap their values so that b[h..k] is sorted. */
public static void merge(
    int b[], int h, int e, int k)

/** Sort b[h..k]. */
public static void mergeSort(int[] b, int h, int k) {
    if (h >= k) return; // if b[h..k] has size 0 or 1
    int e= (h + k) / 2;
    mergeSort(b, h, e);  // Sort b[h..e]
    mergeSort(b, e + 1, k); // Sort b[e+1..k]
    merge(b, h, e, k);  // Merge the 2 segments
}
\end{verbatim}

See the next page, please.
Remembering merge Sort and quick Sort

Some people have trouble remembering merge Sort and quick Sort. Here’s one way to think about them.

- merge Sort relies on method merge to merge to adjacent sorted segments.
- quick Sort relies on the partition algorithm to place values on one or the other side of the pivot.

```java
/** Sort b[h..k]. */
public static void mergeSort(int[] b, int h, int k) {
    if (h >= k) return; // if b[h..k] has size 0 or 1
    int e = (h + k) / 2;
    mergeSort(b, h, e);   // Sort b[h..e]
    mergeSort(b, e + 1, k); // Sort b[e+1..k]
    merge(b, h, e, k);   // Merge b[h..e] and b[e+1..k]
}
```

```java
/** Sort b[h..k]. */
public static void quickSort(int[] b, int h, int k) {
    if (h >= k) return; // if b[h..k] has size 0 or 1
    int j = partition(b, h, k);
    // b[h..j-1] ≤ b[j] ≤ b[j+1..k]
    quickSort(b, h, j-1);  // Sort b[h..j-1]
    quickSort(b, j+1, k);  // Sort b[j+1..k]
}
```