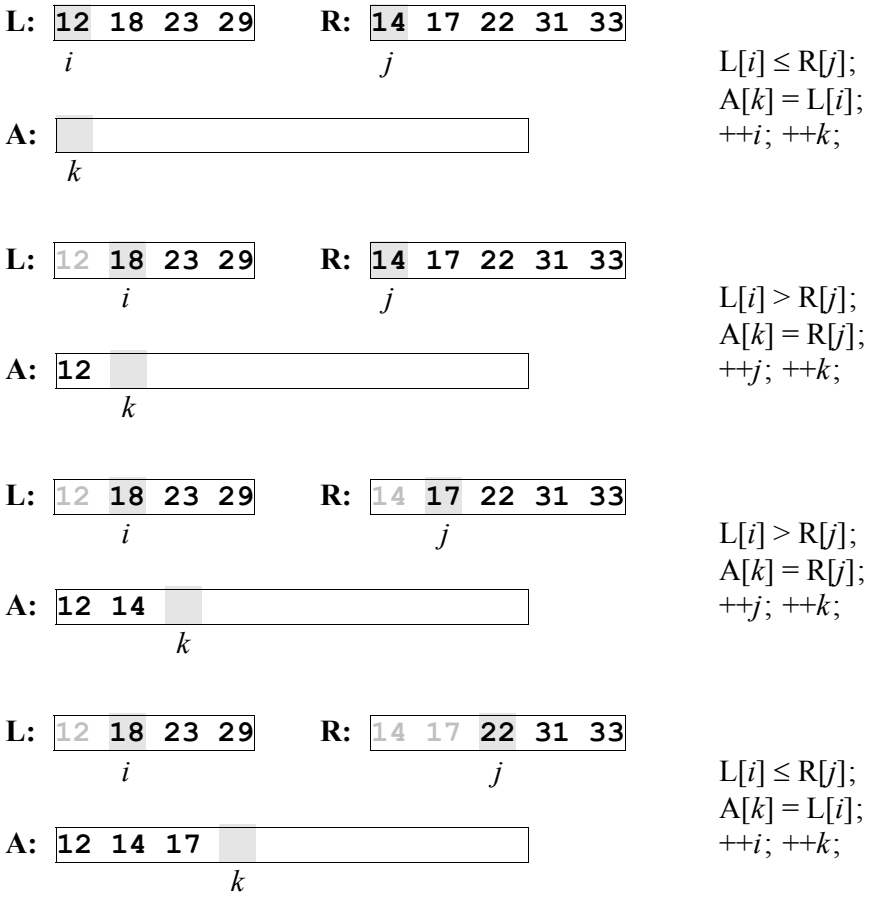
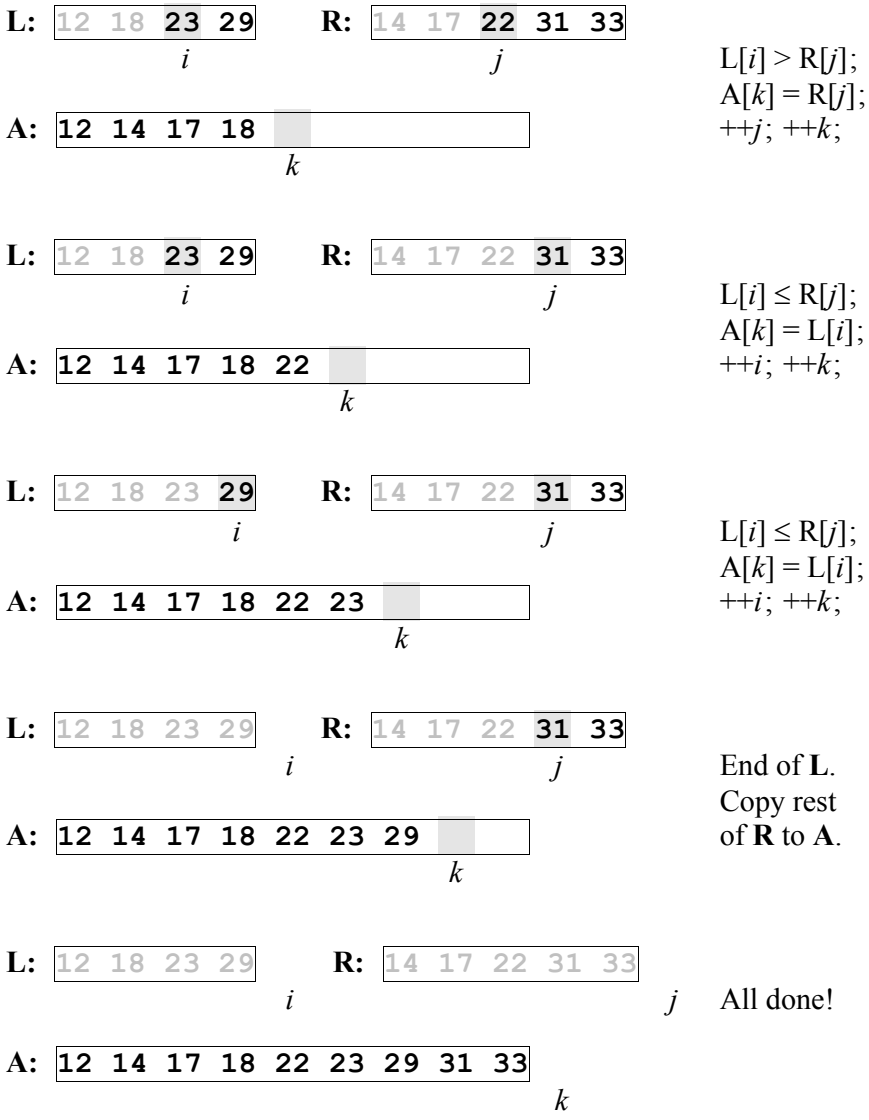
Merging and Mergesort

Merging: Merging two (or more) lists means combining the lists, which must already be sorted, into a single sorted list.

Merging is easier than sorting. There is a very efficient algorithm for merging, illustrated below. Arrays **L** and **R** are merged into **A**.





Unfortunately, this algorithm does not work in place. (L and R cannot occupy the same memory as A.)

Algorithm for Merging: In our merging algorithm, we assume that the lists being merged occupy contiguous ranges within an array A, and the merged list is stored back to the union of these ranges. That is, we merge A[p..q] and A[q+1..r] to A[p..r].

Input: An array A of element type T, where T has a strict weak ordering (denoted by <), and integers p, q, and r with $p \le q \le r$, such that A[p..q] and A[q+1..r] are sorted.

Output: The array A, with A[p..r] sorted and the remaining elements unchanged.

Algorithm:

```
void merge( T[] A, Integer p, Integer q, Integer r)
    L = temp array of size 1..q–p+1;
    L[1..q–p+1] = A[p..q];
    R = temp array of size 1..r–q;
    R[1..r–q] = A[q+1..r];
    i = 1;  j = 1;  k = p;                  // i, j, k = current positions in L, R, A.
    while ( i ≤ q–p+1 and j ≤ r–q )         // Merge from L and R to A until
        if ( L[i] ≤ R[j] )                  //   one of L or R is exhausted.
            A[k] = L[i];
            ++i; ++k;
        else
            A[k] = R[j];
            ++j; ++k;
    while ( i ≤ q–p+1 )                     // Copy remainder of L to A.
        A[k] = L[i];
        ++i; ++k;
    while ( j ≤ r–q )                       // Copy remainder of R to A
        A[k] = R[j];
        ++j; ++k;
```

If it is feasible to place an extra element, with value ∞, at the end of L and R, we can shorten the algorithm a bit.

Here ∞ need merely be greater than the largest possible element of L or R.

Algorithm (alternate version):

```
void merge2( T[] A, Integer p, Integer q, Integer r)
    L = temp array of size 1..q–p+2;
    L[1..q–p+1] = A[p..q];   L[q–p+2] = ∞;
    R = temp array of size 1..r–q+1;
    R[1..r–q] = A[q+1..r];   R[r–q+1] = ∞;
    i = 1;  j = 1;                          // i, j, k = current positions in L, R, A.
    for ( k = p, p+1, ..., r–1, r )         // Merge from L and R to A.
        if ( L[i] ≤ R[j] )
            A[k] = L[i];
            ++i;
        else
            A[k] = R[j];
            ++j;
```

Analysis of merging:

We consider two basic operations:

i) *Comparisons* of array elements. $C(n)$ = no of comparisons.

ii) *Moves* (copying) of array elements. $M(n)$ = no of moves.

Here $n = r–p+1$ (the input size).

merge( ) performs ***n*–1** comparisons in the worst case:

$$\boxed{C_{\max}(n) = n-1}.$$

merge( ) performs **nearly *n*–1** comparisons in the expected case, if the two lists being merged have comparable size, and elements are randomly distributed between the lists:

$$\boxed{C_{\text{ave}}(n) \approx n-1}$$ under appropriate assumptions.

merge( ) always performs $2n$ moves:

$$\boxed{M_{\max}(n) = M_{\text{ave}}(n) = 2n}.$$

merge2( ) performs n comparisons and $2n$ (or $2n+2$) moves in all cases.

Both merging algorithms run in linear time: $T(n) = \Theta(n)$ in all cases.

Other nice features of merging:

i) *merge*() accesses all arrays in forward sequential order.

   a) This means that it can be adapted easily to merge singly or doubly linked lists.

ii) Our merging algorithms are stable, in the following sense:

   If $i < j$ and A[i] ~ A[j], and if *merge*() moves A[i] to position i' and A[j] to position j', then $i' < j'$. In other words, the relative order of equivalent elements is preserved.

   A[i] ~ A[j] might mean that structures A[i] and A[j] agree on their key field (used to sort), but possibly not on other fields.

Mergesort: Mergesort is a divide-and-conquer algorithm for sorting.

To sort an array A[p **..** r] (a problem with input size $n = r-p+1$) using mergesort, we

i) [*Divide*] Divide into two subproblems by setting $q = \lfloor (p+r)/2 \rfloor$ and defining

   Subproblem 1: Sort A[p **..** q] (input size = $\lceil n/2 \rceil$).

   Subproblem 2: Sort A[$q+1$ **..** r] (input size = $\lfloor n/2 \rfloor$).

ii) [*Solve subproblems*] Solve each subproblem by invoking mergesort recursively, unless the subproblem has size 1.

iii) [*Combine*] Merge the (now sorted) subarrays A[p **..** q] and A[$q+1$ **..** r] into a sorted array A[p **..** r]. This may be done by invoking *merge*(A, p, q, r).

However, if $n = 1$, we don't divide into subproblems. In fact, there is nothing to do; the array is already sorted.

Algorithm for Mergesort:

Input: An array A of element type T, where T has a strict weak ordering (denoted by <), and integers p and r with $p \le r$.

Output: The array A, with A[p **..** r] sorted and the remaining elements unchanged.

Algorithm:

```
void mergesort( T[] A, Integer p, Integer r)
    if ( p < r )
        q = ⌊(p+r)/2⌋;
        mergesort( A, p, q);
        mergesort( A, q+1, r);
        merge( A, p, q, r);
```

Analysis of Mergesort:

We count the number $C(n)$ of comparisons of elements of A. Consider the worst case.

If $n = 1$, no comparisons are required.

Otherwise:

Step (i) is trivial and uses no comparisons.

Step (ii) requires $C_{\max}(\lceil n/2 \rceil) + C_{\max}(\lfloor n/2 \rfloor)$ comparisons, in the worse case

Step (iii) requires $n-1$ comparisons, in the worst case.

$$C_{\max}(1) = 0,$$
$$C_{\max}(n) = n-1 + C_{\max}(\lceil n/2 \rceil) + C_{\max}(\lfloor n/2 \rfloor), \text{ if } n > 1.$$

We have derived a recurrence equation for $C_{\max}(n)$.

For any specific n, we can compute $C_{\max}(n)$ explicitly.

n	$C_{\max}(n)$
1	0
2	2–1+C(1)+C(1) = **1**
3	3–1+C(2)+C(1) = **3**
4	4–1+C(2)+C(2) = **5**
5	5–1+C(3)+C(2) = **8**
6	6–1+C(3)+C(3) = **11**
7	7–1+C(4)+C(3) = **14**
8	8–1+C(4)+C(4) = **17**
9	9–1+C(5)+C(4) = **21**
10	10–1+C(5)+C(5) = **25**
11	11–1+C(6)+C(5) = **29**
12	12–1+C(6)+C(6) = **33**
13	13–1+C(7)+C(6) = **37**
14	14–1+C(7)+C(7) = **41**
15	15–1+C(8)+C(7) = **45**
16	16–1+C(8)+C(8) = **49**

This isn't too useful! It would be much nicer to obtain $C_{\max}(n)$ as a simple function.

Unfortunately, we can't do this because of the floors and ceilings in the recurrence $C_{\max}(n) = n-1 + C_{\max}(\lceil n/2 \rceil) + C_{\max}(\lfloor n/2 \rfloor)$.

Even if n is even, at on some recursive call we are likely to have a list of odd size.

But if n is a power of 2, we can split the list exactly in half on every recursive call to *mergesort*(), and the recurrence simplifies to

$$C_{\max}(n) = n-1 + 2C_{\max}(n/2).$$

We start by assuming that n is a power of 2, say $n = 2^k$. Then $k = \lg(n)$.

$$\begin{aligned}
C_{\max}(n) &= n - 1 + 2C_{\max}(n/2) \\
&= n - 1 + 2\left(n/2 - 1 + 2C_{\max}(n/2^2)\right) \\
&= 2n - (1+2) + 2^2 C_{\max}(n/2^2) \\
&= 2n - (1+2) + 2^2\left(n/2^2 - 1 + 2C_{\max}(n/2^3)\right) \\
&= 3n - (1+2+2^2) + 2^3 C_{\max}(n/2^3) \\
&\vdots \\
&= kn - (1+2+2^2+\ldots+2^{k-1}) + 2^k C_{\max}(n/2^k) \\
&= n\lg(n) - (n-1) + 0 \qquad \text{(since } k = \lg(n) \text{ and } n/2^k = 1\text{)} \\
&= n\lg(n) - n + 1
\end{aligned}$$

So $C_{\max}(n) = n\lg(n) - n + 1$ <u>exactly</u>, when n is a power of 2.

What if n is not a power of 2? Then, at some step, we cannot divide the list exactly in half. But except for very small n, we divide it into two lists of nearly equal size (ratio of sizes very close to 1).

We might expect the ratio of $C_{\max}(n)$ to $n\lg(n) - n + 1$ to be only slightly greater than 1.

The handout *Recurrences: Approximating $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ by n/2* shows that this is correct, at least for $n < 10^6$.

Note, for example, that for $10^3 \le n < 10^4$,

$$n\lg(n) - n + 1 \;\le\; C_{\max}(n) \;<\; 1.0091(n\lg(n) - n + 1)$$

and that for larger n the approximation is even better.

So we will use $\boxed{C_{\max}(n) \approx n\lg(n) - n + 1}$ as our solution, in general.

$C_{\text{ave}}(n)$ is only slightly less than $C_{\max}(n)$.