7.5 Merge Sort

We will now look at a second Θ(*n* ln(*n*)) algorithm: merge sort. This is the first divide-and-conquer algorithm: we solve the problem by dividing the problem into smaller sub-problems, we recursively call the algorithm on the sub-problems, and we then recombine the solutions to the sub-problems to create a solution to the overall problem.

We will recursively define merge sort as follows:

1. If the list is of size 1,
2. Otherwise,
    a. Divide the unsorted list into two sub-lists,
    b. Recursively call merge sort on each sub-list, and
    c. Merge the two sorted sub-lists together into a single sorted list.

We will first assume we have two sorted lists: what is the algorithm for merging them?

7.5.1 Merging Sorted Lists

Suppose we have two sorted lists, how can we merge them into a single sorted list? Consider the two lists of size n_1 and n_2:

3	5	18	21	24	27	31

and

2	7	12	16	33	37	42

We must begin with a new list of size $n_1 + n_2$ and we start with three indices all set to 0:

🟥 **3**	5	18	21	24	27	31

🟥 **2**	7	12	16	33	37	42

🟦													

We copy the smaller of the two objects into the new array and increment that index:

🟥 **3**	5	18	21	24	27	31

2	🟥 **7**	12	16	33	37	42

2	🟦												

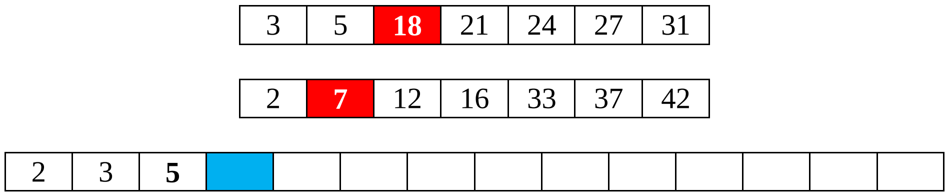
Repeating this process, we compare 3 and 7 and now copy 3 into the new list and increment that index:

3	🟥 **5**	18	21	24	27	31

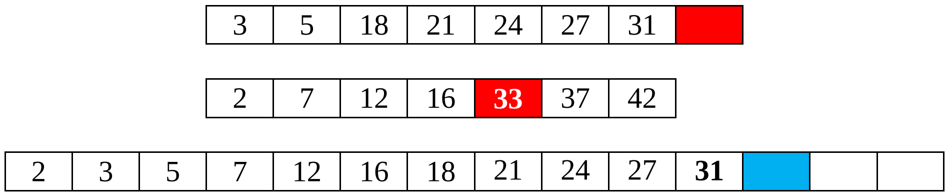
2	🟥 **7**	12	16	33	37	42

2	**3**	🟦											

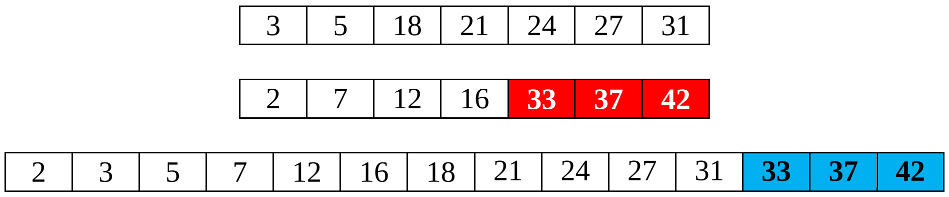
At this point, we would copy from the first array again, incrementing that index:



We could repeat this, however, at some point the index of one of the two arrays will be beyond the end of the array. Having copied 31 into the new array, the index of the first array is beyond the end



At this point, however, all we must do is copy the remaining entries of the second array down:



and we are done.

7.5.1.1 Implementing Merging

Let's assume that the two arrays are `array1` and `array2` with sizes `n1` and `n2` while the output array `arrayout` is of size `n1 + n2`. We begin by defining three indices:

```
int i1 = 0, i2 = 0, k = 0;
```

Next, we iterate through the lists:

```
while ( i1 < n1 && i2 < n2 ) {
    if ( array1[i1] < array2[i2] ) {
        arrayout[k] = array1[i1];
        ++i1;
    } else {
        assert( array1[i1] >= array2[i2] );      // Requires #include <cassert>

        arrayout[k] = array2[i2];
        ++i2;
    }

    ++k;
}
```

Finally, all the entries in one of the two arrays has not yet been copied over, so we finish by copying those over:

```
for ( ; i1 < n1; ++i1, ++k ) {
    arrayout[k] = array1[in1];
}

for ( ; i2 < n2; ++i2, ++k ) {
    arrayout[k] = array2[i2];
}
```

7.5.1.2 Run-time Analysis of Merging

You will note that one of these two loops will never run: the first array ran until one of `i1 < n1` or `i2 < n2` evaluated to false (`0`).

The run-time of merging can be quickly determined by realizing that the statement `++k` will only be executed $n_1 + n_2$ times, and therefore the run time is $\Theta(n_1 + n_2)$. If the sizes of the arrays are comparable, that is, $n = n_1$ and $n_1 \approx n_2$, we can say that the run time is $\Theta(n)$.

7.5.2 The Algorithm

Thus, of the five sorting techniques (insertion, exchange, selection, merging, and distribution), ours falls into the fourth case, merging. Now that we know we can merge two lists in $\Theta(n)$, we will simply apply the algorithm:

1. If the array is of size 1, it is sorted and we are finished;
2. Otherwise,
    a. Split the list into two approximately equal sub-lists,
    b. Recursively call merge sort on those sub-lists, and
    c. Merge the resulting sorted sub-lists together into one sorted list.

Question: does it make sense to recursively call merge sort on a list of size 2? Consider the overhead: two function calls, allocating a new array, and merging the two lists together.

In fact, should we even call merge sort recursively on a list of size under 8 or under 16? Certainly the overhead of making two function calls can be expensive.

Consequently, it is reasonable to consider an alternative algorithm:

1. If the size of the array is less than some constant N, use **insertion sort** to sort it,
2. Otherwise,
    a. Split the list into two approximately equal sub-lists,
    b. Recursively call merge sort on those sub-lists, and
    c. Merge the resulting sorted sub-lists together into one sorted list.

Thus, if the list is sufficiently small, insertion sort will be quicker than merge sort. In reality, this constant N can be very large: in one experiment, the author found $N = 64$ as being appropriate.

7.5.3 Implementation

Assume we have a merging function

```
void merge( Object *array, int a, int b, int c );
```

that assumes that the entries in positions `a` through `b - 1` are sorted and the entries in positions `b` through `c` are sorted and returns with the entries from `a` through `c` merged together into a sorted list. We will now implement a function

```
void merge_sort( Object *array, int first, first last );
```

which will sort the entries of the argument `array` from index `first` to index `last`, inclusive. We will start by checking if there are fewer than *N* elements, in which case, we will call `insertion_sort` on those entries. Otherwise, we will find the mid-point, recursively call merge sort on both halves, and then merge the results:

```
void merge_sort( Object *array, int first, int last ) {
    if ( last - first <= N ) {
        insertion_sort( array, first, last );
    } else {
        int midpoint = (first + last)/2;

        merge_sort( array, first, midpoint );
        merge_sort( array, midpoint + 1, last );
        merge( array, first, midpoint + 1, last );
    }
}
```

7.5.4 Example

Consider sorting the following array of size 23:

13	77	49	35	61	48	3	23	95	73	89	37	57	99	94	28	15	55	7	51	88	97	62

We will call insertion sort whenever the sub-list has a size less than $N = 8$.

We begin by noting that the first and last entries have indices `first = 0` and `last = 22`, and therefore we will split the list at `int midpoint = (first + last)/2` and recursively call merge sort on the entries `first` through `midpoint` and `midpoint + 1` through `last`.

Applying merge sort on the entries 0 through 11,

13	77	49	35	61	48	3	23	95	73	89	37	57	99	94	28	15	55	7	51	88	97	62

This sub-array has more than $N = 8$ entries, so we recursively call merge sort separating these list at `midpoint = (0 + 11)/2` which equals `5`, so we recursively call merge sort on the entries `0` through`5`:

13	77	49	35	61	48	3	23	95	73	89	37	57	99	94	28	15	55	7	51	88	97	62

This array is of size less than $N = 8$, so we just sort them using insertion sort:

13	35	48	49	61	77	3	23	95	73	89	37	57	99	94	28	15	55	7	51	88	97	62

We proceed to sort the entries `middle + 1 == 6` through `11`:

13	35	48	49	61	77	3	23	95	73	89	37	57	99	94	28	15	55	7	51	88	97	62

Again, it is of size less than $N = 8$, so we sort it using insertion sort:

13	35	48	49	61	77	3	23	37	73	89	95	57	99	94	28	15	55	7	51	88	97	62

We now merge these two lists together into one sorted sub-list of size 12:

3	13	23	35	37	48	49	61	73	77	89	95	57	99	94	28	15	55	7	51	88	97	62

Next, we proceed to sort the lists from 12 to 23:

3	13	23	35	37	48	49	61	73	77	89	95	57	99	94	28	15	55	7	51	88	97	62

Again, we calculate `midpoint = (12 + 23)/2` which equals `17`, so we call merge sort on entries 12 through 17:

3	13	23	35	37	48	49	61	73	77	89	95	57	99	94	28	15	55	7	51	88	97	62

The size is less than $N = 8$, so we apply insertion sort and continue sorting the entries 18 through 23:

3	13	23	35	37	48	49	61	73	77	89	95	15	28	55	57	94	99	7	51	88	97	62

Again, we apply insertion sort

3	13	23	35	37	48	49	61	73	77	89	95	15	28	55	57	94	99	7	51	62	88	97

and merge the two lists into a single sorted list:

3	13	23	35	37	48	49	61	73	77	89	95	7	15	28	51	55	57	62	88	94	97	99

Finally, we merge the two sub-lists together to form a single sorted list:

3	7	13	15	23	28	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

7.5.5 Run-time Analysis

The time required to perform a merge sort (ignoring our optimization by calling insertion sort) on an array of size *n* is:

1. The time required to sort the left half containing approximately *n*/2 entries,
2. The time required to sort the right half, again with *n*/2 entries, and
3. The time required to merge the results.

This gives us

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$

Solving this in Maple gives us:

```
> rsolve( {T(n) = 2*T(n/2) + n, T(1) = 1}, T(n) );
```

$$\frac{n(\ln(2) + \ln(n))}{\ln(2)}$$

which can be simplified to *n* + *n* lg(*n*); thus, the run-time of merge sort is Θ(*n* lg(*n*)).

Later on, we will see the *master theorem* which will give us the run-time of most variations of *divide-and-conquer* algorithms.

There are no best-case and no worst-case scenarios for merge sort. They will all have the same *n* lg(*n*) run time.

In practice, merge sort is faster than heap sort; however, unlike heap sort, merge sort requires the allocation of an addition array for the merging process: this requires Θ(*n*) additional memory. Next we will see quick sort which is, on average, faster than both heap sort and merge sort and usually requires only Θ(ln(*n*)) additional memory.