ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo
Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Notes 8.5.
Assistances and comments will be acknowledged.

8.5 Merge Sort

We will now look at a second $\Theta(n \ln(n))$ 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?

8.5.1 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													

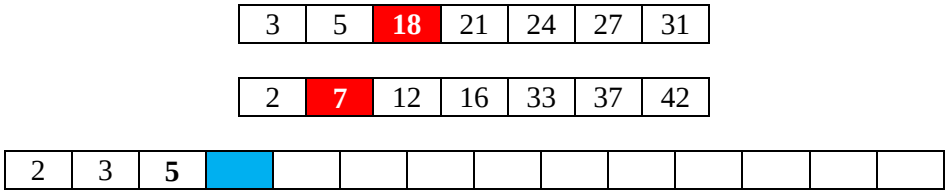
Repeating this process, we compare 3 and 7 and now copy 3 into the new list and increment that index:

3	**5**	18	21	24	27	31

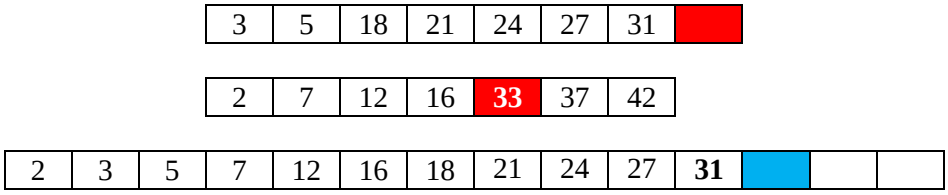
2	**7**	12	16	33	37	42

2	**3**												

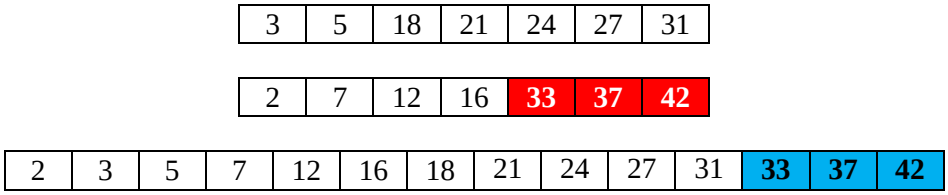
At this point, we would copy from the first array again, incrementing that index:



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.

8.5.1.1 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];
}
```

8.5.1.2 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 $\Theta(n)$.

We do, however, have one significant problem: the merging of two lists—even if they are adjacent—requires the allocation of another array at least equal to the smaller of the two arrays being merged. Thus, if we are merging two lists of size n, the memory requirements will also be $\Theta(n)$.

8.5.2 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.

8.5.3 Implementation

Assume we have a merging function

```
template <typename Type>
void merge( Type *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 - 1` are sorted and returns with the entries from `a` through `c - 1` merged together into a sorted list. Such a function will have to allocation additional memory internally.

For example, consider this array where the entries from 14 to 19 are sorted, and the entries from 20 to 25 are sorted.

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	23	48	73	89	95	17	32	37	57	94	99	28	15	55	7	51	88	97	62

Suppose we wish to merge these two together.

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	23	48	73	89	95	17	32	37	57	94	99	28	15	55	7	51	88	97	62

To do this, we call merge(array, 14, 20, 26), which results in:

9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
13	77	49	35	61	3	17	23	32	37	48	57	73	89	94	95	99	28	15	55	7	51	88	97	62

Note that surrounding entries are not affected.

We will now implement a function

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

which will sort the entries of the argument `array` from indices `first <= i` to `i < 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:

```
template <typename Type>
void merge_sort( Type *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, last );
        merge( array, first, midpoint, last );
    }
}
```

The implementation of insertion sort would also be restricted to soring entries on a sub-range of the array:

```
template <typename Type>
void insertion_sort( Type *array, int first, int last ) {
    for ( int k = first + 1; k < last; ++k ) {
        Type tmp = array[k];
        for ( int j = k; k > first; --j ) {
            if ( array[j - 1] > tmp ) {
                array[j] = array[j - 1];
            } else {
                array[j] = tmp;
                goto finished;
            }
        }
        array[first] = tmp;
        finished: ;
    }
}
```

8.5.4 Example

Consider sorting the following array of size 25:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**77**	**49**	**35**	**61**	**48**	**73**	**23**	**95**	**3**	**89**	**37**	**57**	**99**	**17**	**32**	**94**	**28**	**15**	**55**	**7**	**51**	**88**	**97**	**62**

We will call insertion sort whenever the sub-list has a size less than $N = 6$.

Thus, we start with a call to `merge_sort( array, 0, 25 );`

We begin by noting that the first and last entries have indices `first = 0` and `last = 25`, and because $25 - 0 > 6$, we will calculate the midpoint and recursively call merge sort:

```
// Code fragment 1
int midpoint = (0 + 25)/2;  // == 12
merge_sort( array,  0, 12 );
merge_sort( array, 12, 25 );
merge( array, 0, 12, 25 );
```

However, we begin by executing the first of these, so now we call merge sort on the first half:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**77**	**49**	**35**	**61**	**48**	**73**	**23**	**95**	**3**	**89**	**37**	57	99	17	32	94	28	15	55	7	51	88	97	62

Because $12 - 0 > 6$, we will recursively call merge sort:

```
// Code fragment 2
int midpoint = (0 + 12)/2;  // == 6
merge_sort( array,  0, 6 );
merge_sort( array, 6, 12 );
merge( array, 0, 6, 12 );
```

but again, we start by calling the first call to merge sort:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**77**	**49**	**35**	**61**	**48**	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

At this point, $6 - 0 \leq 6$, so we call insertion sort which is performed in-place:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**35**	**48**	**49**	**61**	**77**	73	23	95	3	89	37	57	99	17	32	94	28	15	55	7	51	88	97	62

Insertion sort finishes and returns, and consequently, the call to `merge_sort( array, 0, 6 )` finishes and returns, too. We now go to continue executing the second function call in Code fragment 2. We are now calling merge sort on the second half of the first half the array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	**73**	**23**	**95**	**3**	**89**	**37**	57	99	17	32	94	28	15	55	7	51	88	97	62

At this point, $12 - 6 \leq 6$, so we call insertion sort which is performed in-place:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	35	48	49	61	77	**3**	**23**	**37**	**73**	**89**	**95**	57	99	17	32	94	28	15	55	7	51	88	97	62

Insertion sort finishes and returns, and consequently, the call to `merge_sort( array, 6, 12 )` finishes and returns, too.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
13	**35**	**48**	**49**	**61**	**77**	**3**	**23**	**37**	**73**	**89**	**95**	57	99	17	32	94	28	15	55	7	51	88	97	62

We now go to continue executing the third function call in Code fragment 2. We will now call `merge( array, 0, 6, 12 )` to merge the two sub-arrays. This results in:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	**13**	**23**	**35**	**37**	**48**	**49**	**61**	**73**	**77**	**89**	**95**	57	99	17	32	94	28	15	55	7	51	88	97	62

At this point, `merge_sort( array, 0, 12 )` is finished. It returns back to the function that called it: `merge_sort( array, 0, 25 )`. Thus, we continue to execute the second function in Code Fragment 1: calling merge sort on the entries 12 through 24.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	**57**	**99**	**17**	**32**	**94**	**28**	**15**	**55**	**7**	**51**	**88**	**97**	**62**

Because 25 – 12 > 6, we will recursively call merge sort:

```
// Code fragment 3
int midpoint = (12 + 25)/2;  // == 18
merge_sort( array,  12, 18 );
merge_sort( array, 18, 25 );
merge( array, 12, 18, 25 );
```

but again, we start by calling the first call to merge sort:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	**57**	**99**	**17**	**32**	**94**	**28**	15	55	7	51	88	97	62

Here, 18 – 12 ≤ 6, so we call insertion sort which is performed in-place:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	**17**	**28**	**32**	**57**	**94**	**99**	15	55	7	51	88	97	62

Insertion sort finishes and returns, and consequently, the call to `merge_sort( array, 12, 18 )` finishes and returns, too. We now go to continue executing the second function call in Code fragment 3. We are now calling merge sort on the second half of the second half the array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	**15**	**55**	**7**	**51**	**88**	**97**	**62**

Because 25 – 18 > 6, we will recursively call merge sort:

```
// Code fragment 4
int midpoint = (18 + 25)/2;  // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
merge( array, 18, 21, 25 );
```

but again, we start by calling the first call to merge sort:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	**15**	**55**	**7**	51	88	97	62

At this point, 21 – 18 ≤ 6, so we call insertion sort which is performed in-place:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	**7**	**15**	**55**	51	88	97	62

Insertion sort finishes and returns, and consequently, the call to `merge_sort( array, 18, 21 )` finishes and returns, too. It returns back to the function that called it: `merge_sort( array, 18, 25 )`. Thus, we continue to execute the second function in Code Fragment 4: calling merge sort on the entries 21 through 24.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	**51**	**88**	**97**	**62**

Here, also, 25 – 21 ≤ 6, so we call insertion sort which is performed in-place:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	7	15	55	**51**	**62**	**88**	**97**

Insertion sort finishes and returns, and consequently, the call to `merge_sort( array, 18, 21 )` finishes and returns, too. We now call the final function in Code Fragment 4: merging the two sub-arrays. We now more the arrays from 18 to 20 and 21 to 24 to get:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	17	28	32	57	94	99	**7**	**15**	**51**	**55**	**62**	**88**	**97**

The call to `merge_sort( array, 18, 25 )` returns with the entries from 18 to 24 sorted. We now return to execution of the function call `merge_sort( array, 12, 25 )` and execute the last function in Code Fragment 3: merging the sub-arrays from 12 to 17 and 18 to 24. This is performed and results in

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	13	23	35	37	48	49	61	73	77	89	95	**7**	**15**	**17**	**28**	**32**	**51**	**55**	**57**	**62**	**88**	**94**	**97**	**99**

Having completed this merging process, the call to `merge_sort( array, 12, 25 )` returns, and we are back executing the last function call in our original call to `merge_sort( array, 0, 25 )`, namely, merging the two sub-arrays from 0 to 11 and 12 to 24. This yields

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
3	**7**	**13**	**15**	**17**	**23**	**28**	**32**	**35**	**37**	**48**	**49**	**51**	**55**	**57**	**61**	**62**	**73**	**77**	**88**	**89**	**94**	**95**	**97**	**99**

which is a sorted list.

8.5.5 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) );
```

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

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

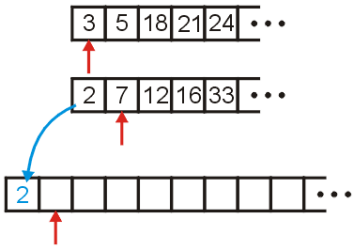
Later on, we will see the *master theorem* which will give us the run-time of most variations of *divide-and-conquer* algorithms.

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 $\Theta(n)$ additional memory. Next we will see quick sort which is, on average, faster than both heap sort and merge sort and usually requires only $\Theta(\ln(n))$ additional memory.

8.5.5 Why is merge sort not O(n^2)?

Recall that merge sort runs in $\Theta(n \ln(n))$ time, even though it removes up to $\binom{n}{2}$ inversions. What operation allows this to occur? Recall that in the merging process, let us suppose that the first list is the first half, and the second list is the second half, and both are of size n. When we compare 2 and 3, we decide to copy 2 into the first location.



This removes n inversions with a single comparison, whereas insertion sort must perform a comparison for every inversion that is removed. Each time during the merge process that an entry is copied from the second array, it removes all inversions associated with the entries in the first array that have not yet been copied. Unfortunately, if the lists are already sorted, we must always still do those comparisons; however, so the run time will be $o(n \ln(n))$ even if the list is initially sorted.