Merge Sort

Acknowledgement:

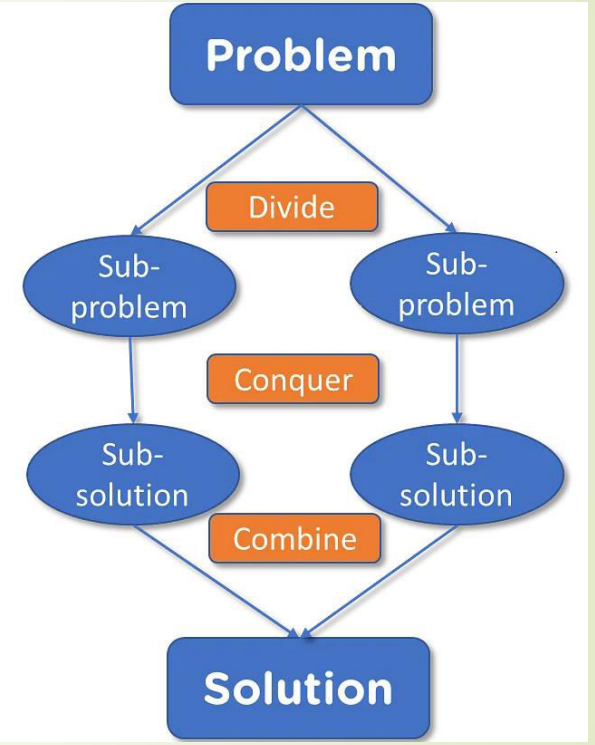
Donald E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 2nd Ed., Addison Wesley, 1998, §5.2.3, p.144-8

Credit: Prof. Douglas Wilhelm Harder, ECE, University of Waterloo, Ontario, Canada

Outline

This topic covers merge sort

- A recursive divide-and-conquer algorithm
- Merging two lists
- The merge sort algorithm
- A run-time analysis
- Applications



Merge Sort

The merge sort algorithm is defined recursively:

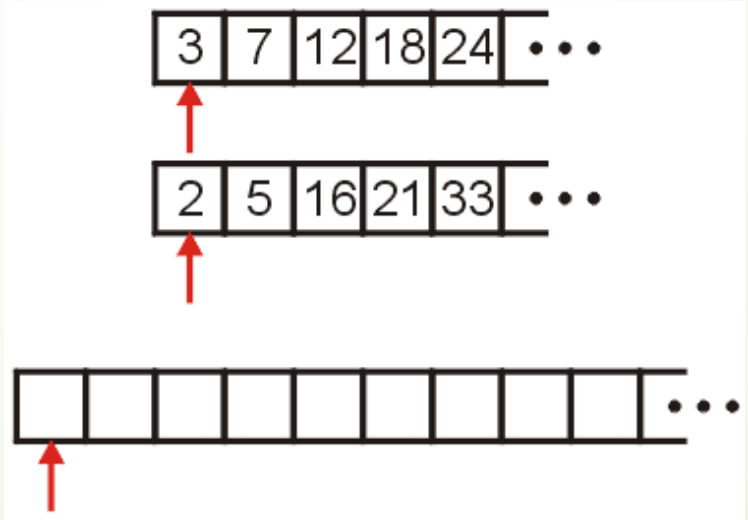
- If the list is of size 1, it is sorted—we are done;
- Otherwise:
  - Divide an unsorted list into two sub-lists,
  - Sort each sub-list recursively using merge sort, and
  - Merge the two sorted sub-lists into a single sorted list

This is the first significant *divide-and-conquer* algorithm we will see

Question: **How quickly can we recombine the two sub-lists into a single sorted list?**

Merging Example

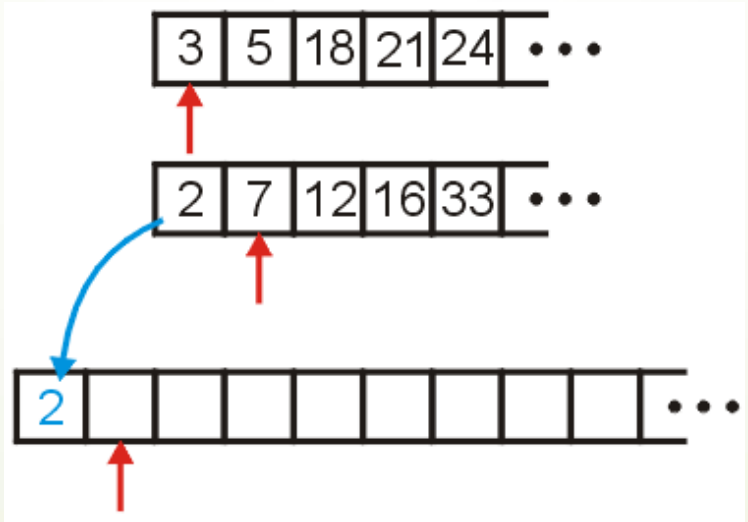
- Consider the two sorted arrays and an empty array
- Define three indices at the start of each array



Merging Example

We compare 2 and 3: 2 < 3

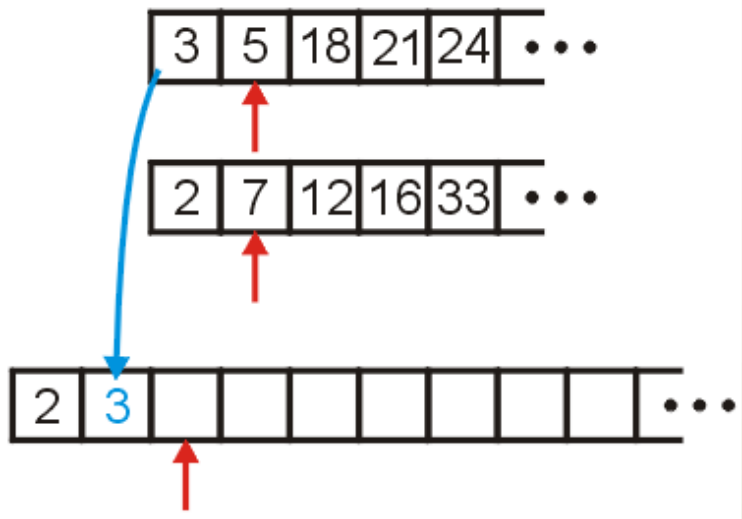
- Copy 2 down
- Increment the corresponding indices



Merging Example

We compare 3 and 7

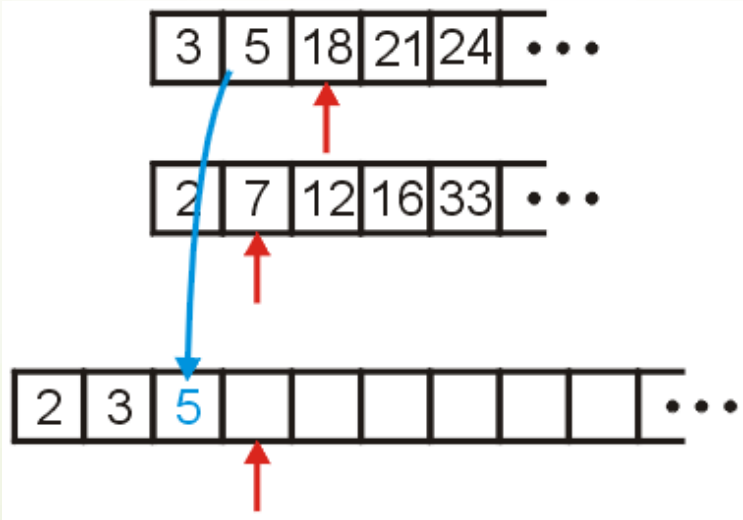
- Copy 3 down
- Increment the corresponding indices



Merging Example

We compare 5 and 7

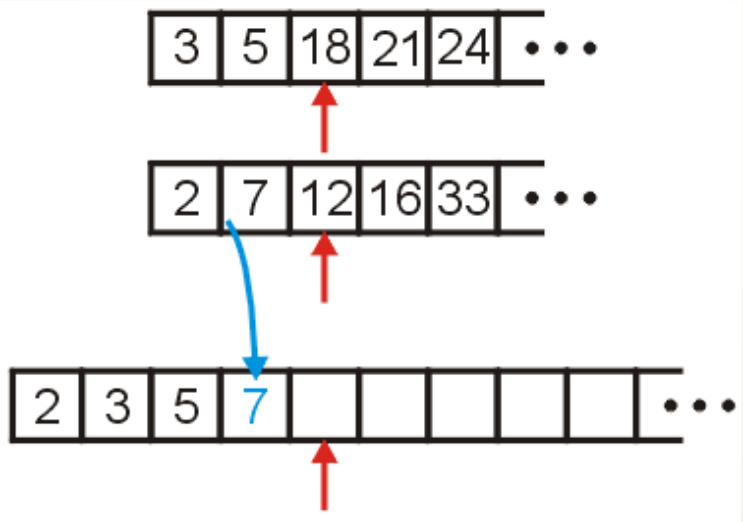
- Copy 5 down
- Increment the appropriate indices



Merging Example

We compare 18 and 7

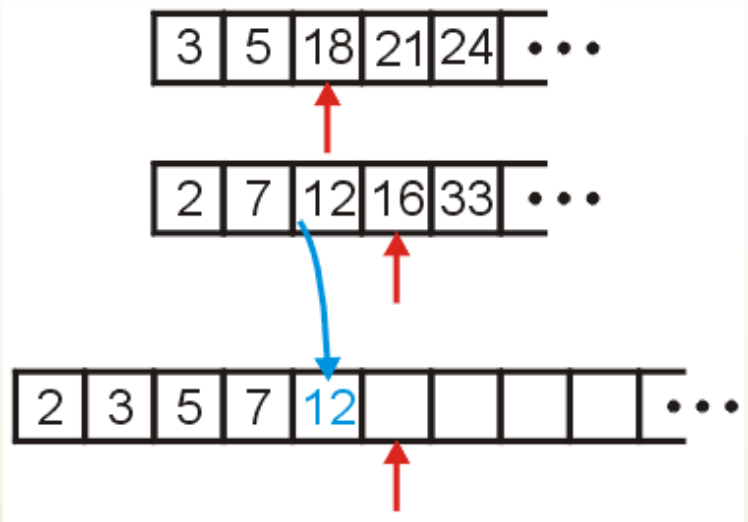
- Copy 7 down
- Increment...



Merging Example

We compare 18 and 12

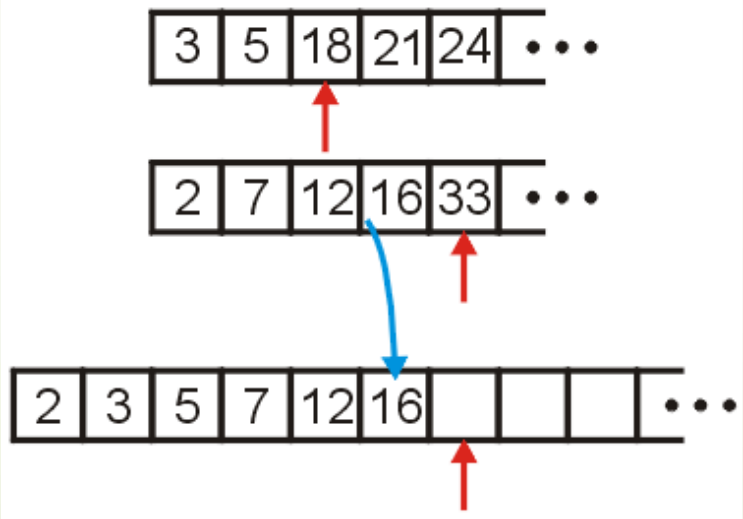
- Copy 12 down
- Increment...



Merging Example

We compare 18 and 16

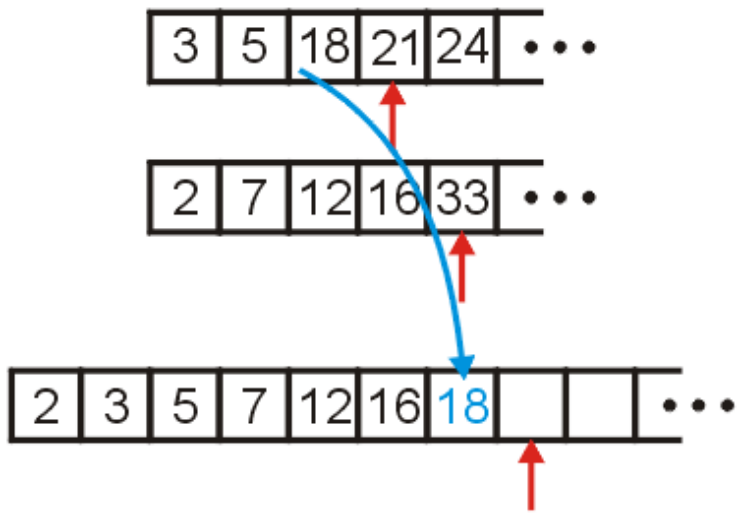
- Copy 16 down
- Increment...



Merging Example

We compare 18 and 33

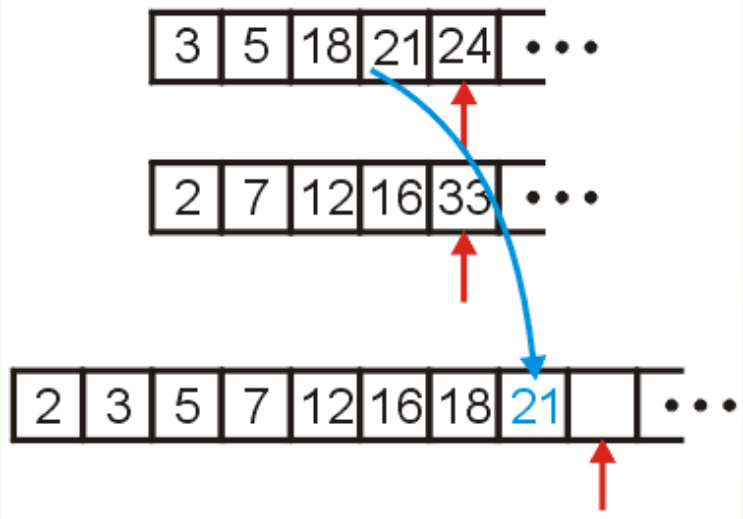
- Copy 18 down
- Increment...



Merging Example

We compare 21 and 33

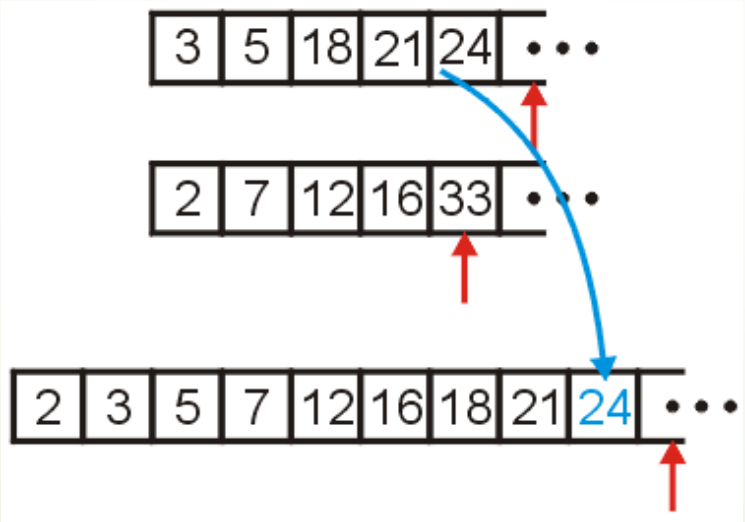
- Copy 21 down
- Increment...



Merging Example

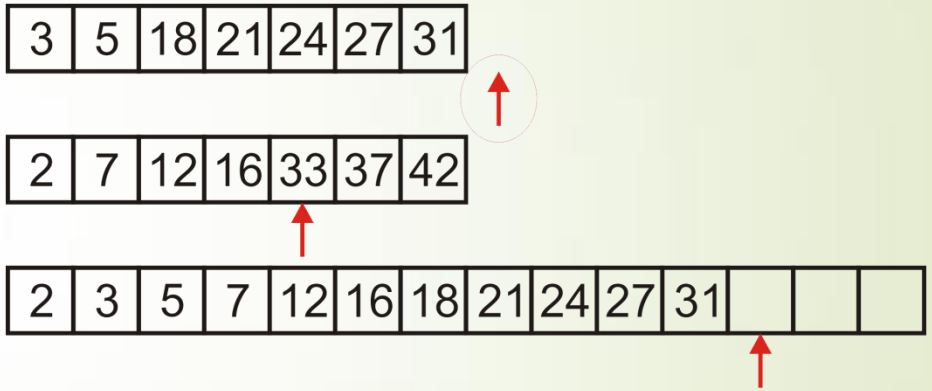
We compare 24 and 33

- Copy 24 down
- Increment...

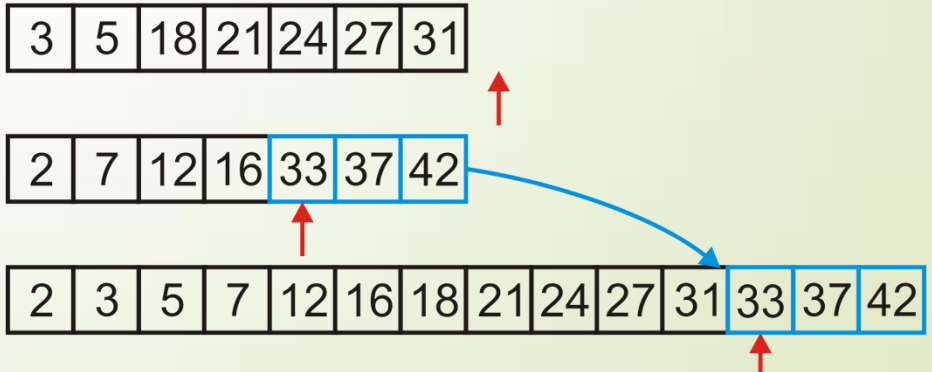


Merging Example

- We would continue until we have passed beyond the limit of one of the two arrays



- After this, we simply copy over all remaining entries in the non-empty array



Merging Two Lists

Programming a merge is straight-forward:

- the sorted arrays, `array1` and `array2`, are of size `n1` and `n2`, respectively, and
- we have an empty array, `arrayout`, of size `n1 + n2`

Define three variables

```
int i1 = 0, i2 = 0, k = 0; // which index into these three arrays
```

Merging Two Lists

We can then run the following loop:

```
#include <cassert>
//...
int i1 = 0, i2 = 0, k = 0;
while ( i1 < n1 && i2 < n2 )
 {
    if ( array1[i1] < array2[i2] )
      {
         arrayout[k] = array1[i1];
         ++i1;
       }
    else
    {
         assert( array1[i1] >= array2[i2] );
         arrayout[k] = array2[i2];
         ++i2;
    }
    ++k;
}
```

Merging Two Lists

We're not finished yet, we have to empty out the remaining array

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

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

Analysis of merging

The statement ++k will only be run at most $n_1 + n_2$ times

- Therefore, the body of the loops run a total of $n_1 + n_2$ times
- Hence, merging may be performed in $\Theta(n_1 + n_2)$ time

If the arrays are approximately the same size, $n = n_1$ and $n_1 \approx n_2$, we can say that the run time is $\Theta(n)$

Problem: We cannot merge two arrays in-place

- This algorithm always required the allocation of a new array
- Therefore, the memory requirements are also $\Theta(n)$ – (not in place)

The Algorithm

The algorithm:

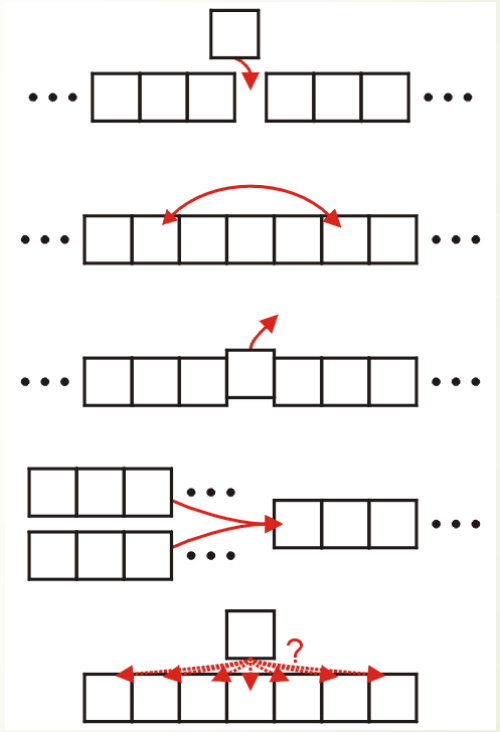
- Split the list into two approximately equal sub-lists
- Recursively call merge sort on both sub lists
- Merge the resulting sorted lists

The Algorithm

Recall the five sorting techniques:

- Insertion
- Exchange
- Selection
- Merging
- Distribution

Clearly merge sort falls into the fourth category



The Algorithm

Question:

- we split the list into two sub-lists and sort them
- how should we sort those lists?

Answer (theoretical):

- if the size of these sub-lists is > 1, use merge sort again
- if the sub-lists are of length 1, do nothing: a list of length one is sorted

The Algorithm

However, just because an algorithm has excellent asymptotic properties, this does not mean that it is practical at all levels

Answer (practical):

- If the sub-lists are less than some threshold length, use an algorithm like insertion sort to sort the lists
- Otherwise, use merge sort, again

Implementation

Suppose we already have a function

```
template <typename Type>
void merge( Type *array, int a, int b, int c );
```

that assumes that the entries

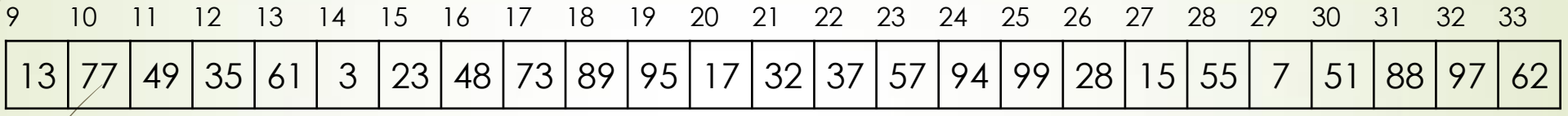
`array[a]` through `array[b - 1]`, and

`array[b]` through `array[c - 1]`

are sorted and merges these two sub-arrays into a single sorted array from index a through index `c - 1`, inclusive

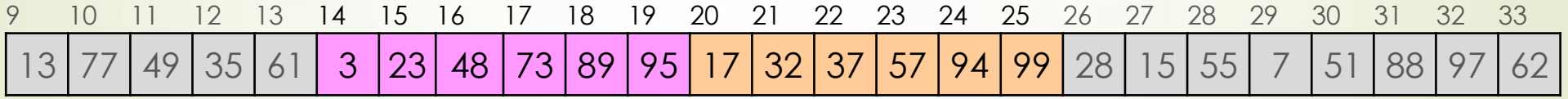
Implementation

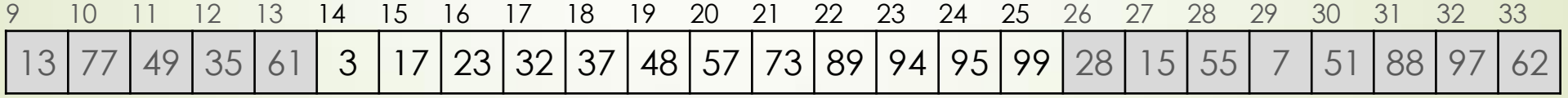
For example, given the array,



a call to `void merge( array, 14, 20, 26 );`

merges the two sub-lists forming





Implementation

We will therefore implement a function

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

that will sort the entries in the positions `first <= i` and `i < last`

- If the number of entries is less than N, call insertion sort
- Otherwise:
  - Find the mid-point,
  - Call merge sort recursively on each of the halves, and
  - Merge the results

Implementation

The actual body is quite small:

```
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 );
    }
}
```

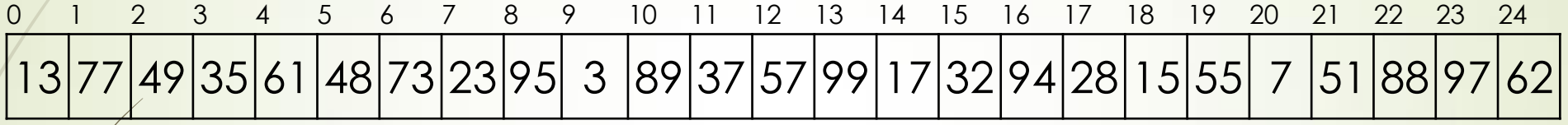
Implementation

Like merge sort, insertion sort will sort a sub-range of the array:

```
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: ;
    }
}
```

Example

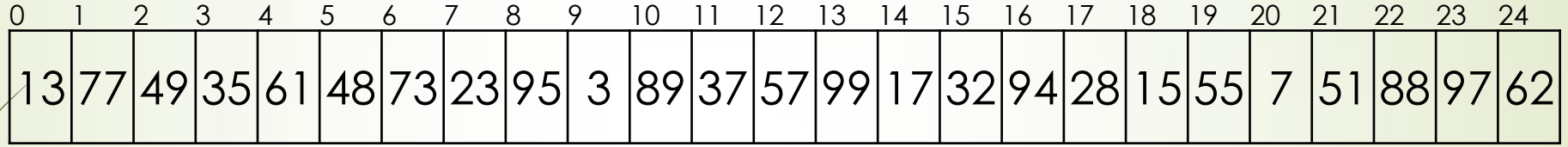
Consider the following is of unsorted array of 25 entries



We will call insertion sort if the list being sorted of size $N = 6$ or less

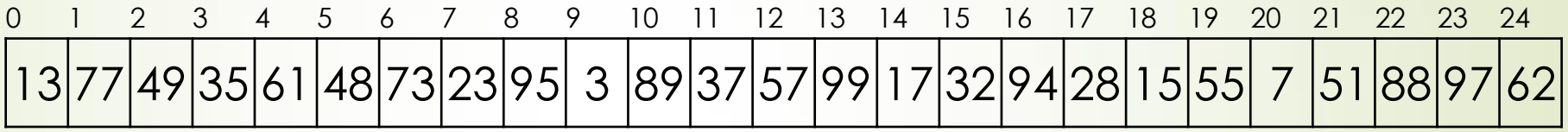
Example

We call `merge_sort( array, 0, 25 )`



Example

We are calling `merge_sort( array, 0, 25 )`



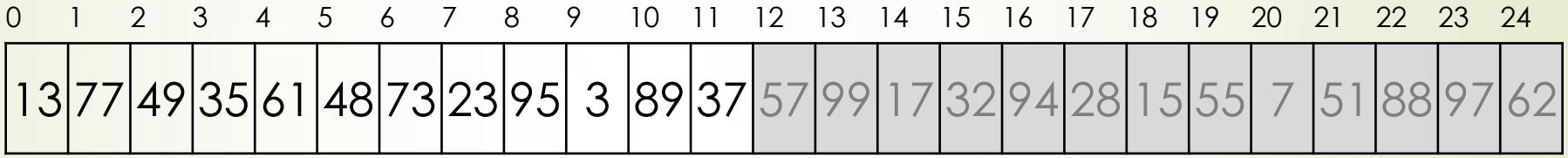
First, $25 - 0 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
```

`merge_sort( array,  0, 25 )`

Example

We are now executing `merge_sort( array, 0, 12 )`



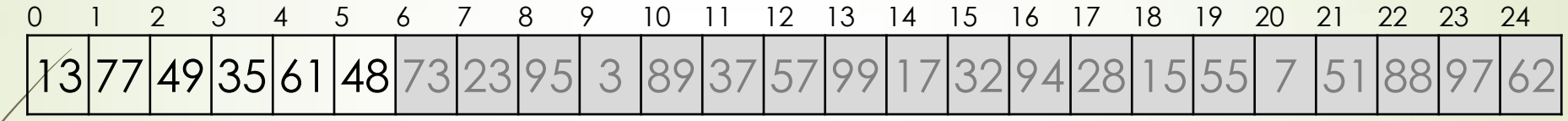
First, $12 - 0 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
```

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 0, 6 )`

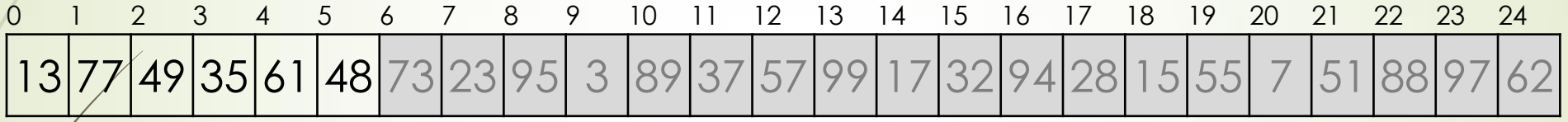


Now, $6 - 0 \leq 6$, so find we call insertion sort

```
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

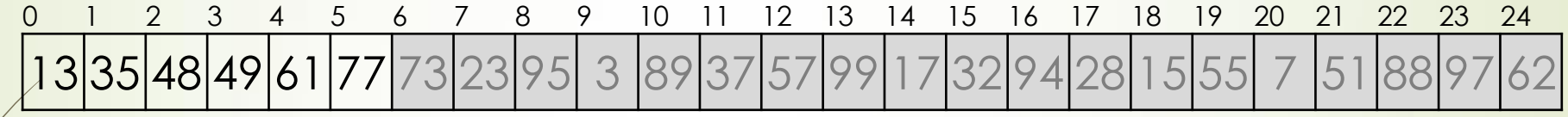
Insertion sort just sorts the entries from 0 to 5




```
insertion_sort( array, 0, 6 )
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from `0` to `5`

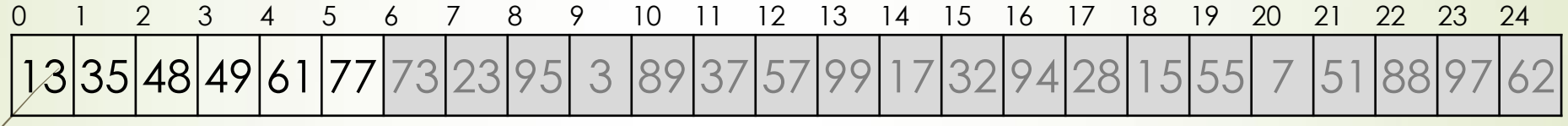


- This function call completes and so we exit

```
insertion_sort( array, 0, 6 )
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

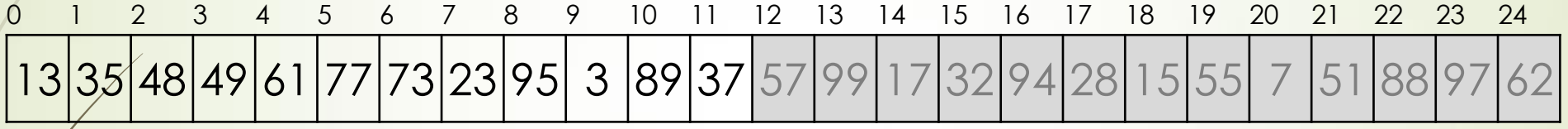
This call to `merge_sort` is now also finished, so it, too, exits




```
merge_sort( array,  0,  6 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 0, 12 )`



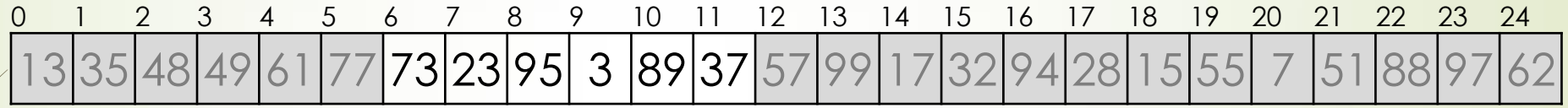
We continue calling

```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
```

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 6,  12 )`

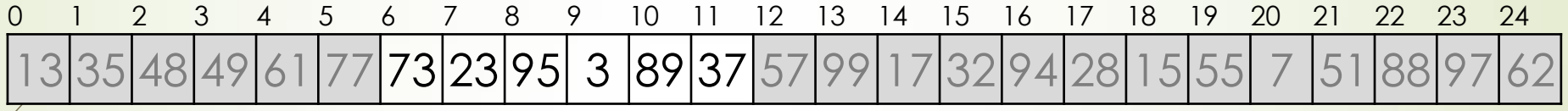


Now, $12 - 6 \leq 6$, so find we call insertion sort

```
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

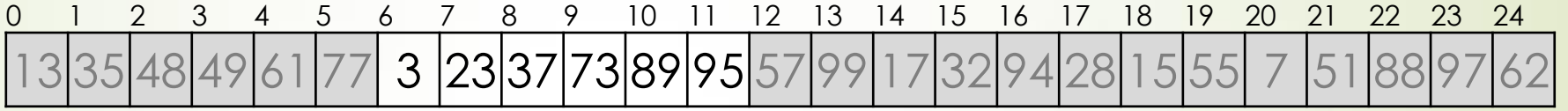
Insertion sort just sorts the entries from 6 to 11




```
insertion_sort( array, 6, 12 )
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 6 to 11

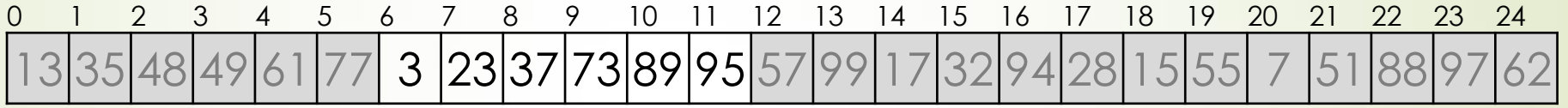


- This function call completes and so we exit

```
insertion_sort( array, 6, 12 )
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

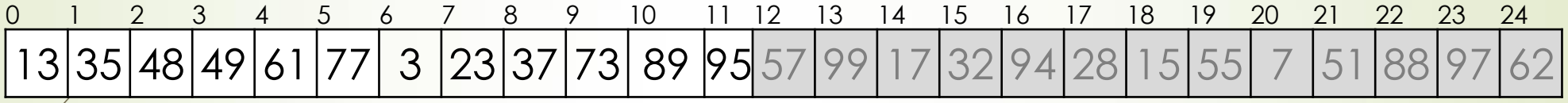
This call to `merge_sort` is now also finished, so it, too, exits




```
merge_sort( array,  6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 0, 12 )`



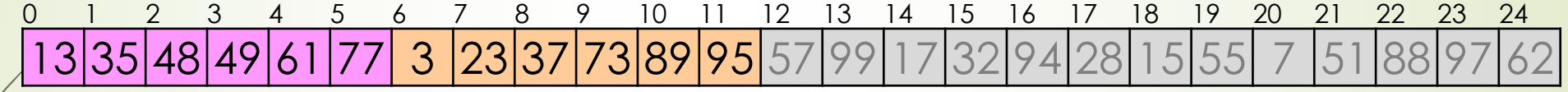
We continue calling

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

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 6, 12 )`

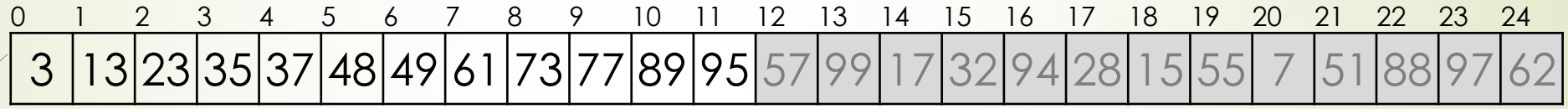


These two sub-arrays are merged together

```
merge( array, 0, 6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 6, 12 )`



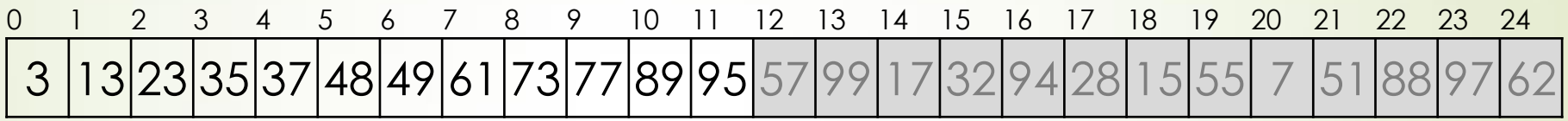
These two sub-arrays are merged together

- This function call exists

```
merge( array, 0, 6, 12 )
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 12 )`



We are finished calling this function as well

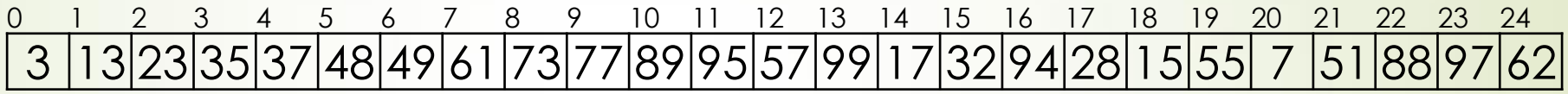
```
midpoint = (0 + 12)/2; // == 6
merge_sort( array, 0, 6 );
merge_sort( array, 6, 12 );
merge( array, 0, 6, 12 );
```

Consequently, we exit

```
merge_sort( array,  0, 12 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 25 )`



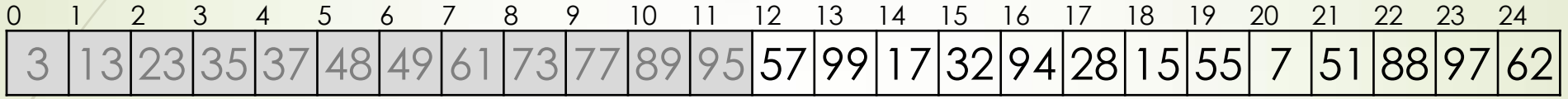
We continue calling

```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
```

`merge_sort( array,  0, 25 )`

Example

We are now executing `merge_sort( array, 12, 25 )`



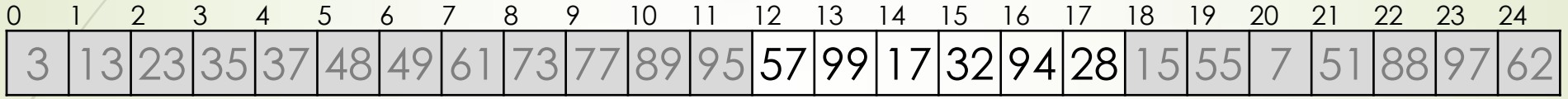
First, $25 - 12 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
```

```
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 12, 18 )`

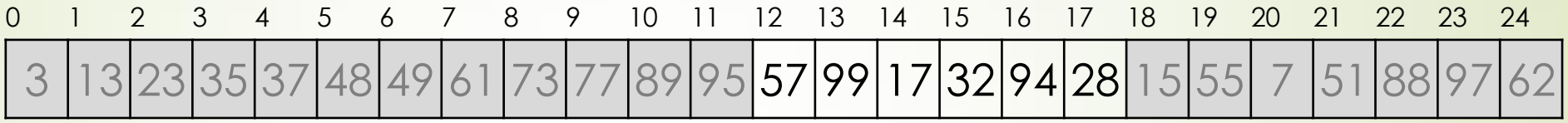


Now, $18 - 12 \leq 6$, so find we call insertion sort

```
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

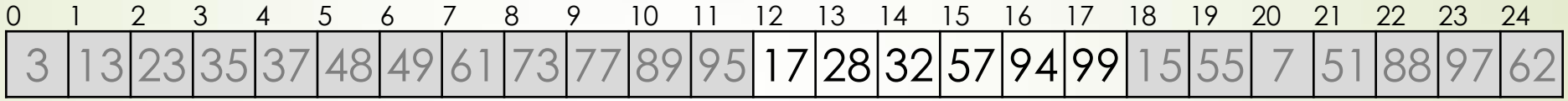
Insertion sort just sorts the entries from 12 to 17




```
insertion_sort( array, 12, 18 )
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 12 to 17

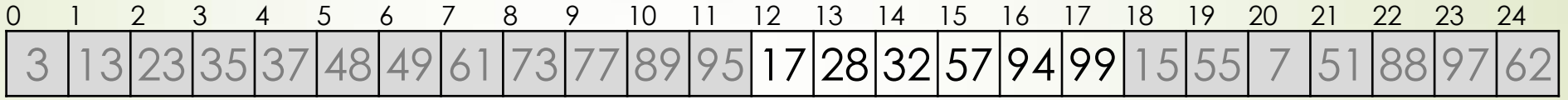


- This function call completes and so we exit

```
insertion_sort( array, 12, 18 )
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

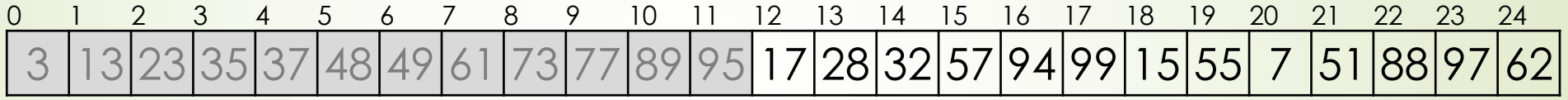
This call to `merge_sort` is now also finished, so it, too, exits




```
merge_sort( array, 12, 18 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 12, 25 )`



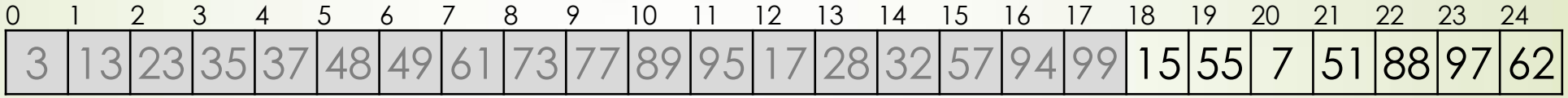
We continue calling

```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
merge_sort( array, 18, 25 );
```

```
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 18, 25 )`



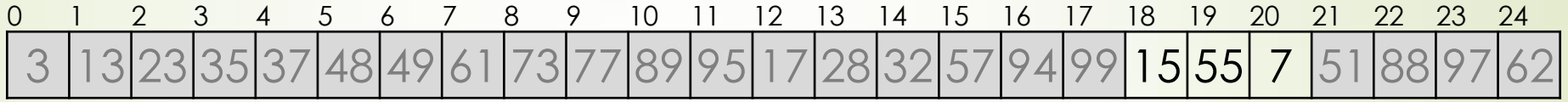
First, $25 - 18 > 6$, so find the midpoint and call `merge_sort` recursively

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
```

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 18, 21 )`

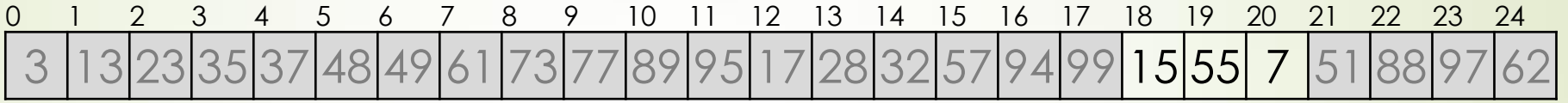


Now, $21 - 18 \leq 6$, so find we call insertion sort

```
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

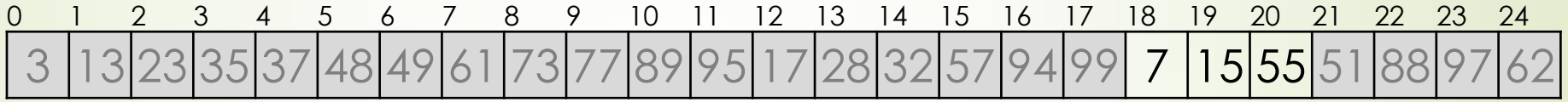
Insertion sort just sorts the entries from 18 to 20




```
insertion_sort( array, 18, 21 )
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 18 to 20

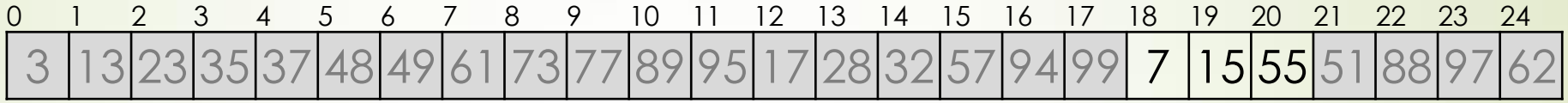


- This function call completes and so we exit

```
insertion_sort( array, 18, 21 )
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

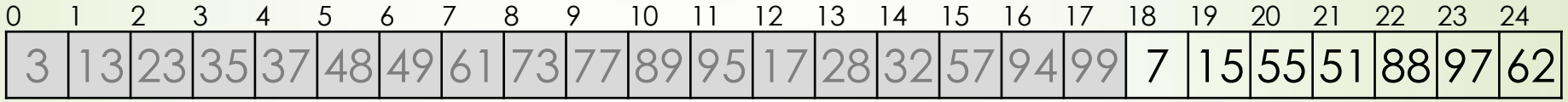
This call to `merge_sort` is now also finished, so it, too, exits




```
merge_sort( array, 18, 21 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 18, 25 )`



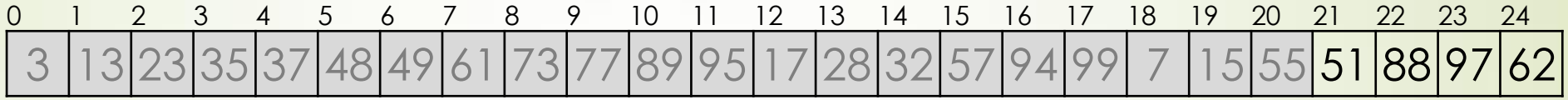
We continue calling

```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
```

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are now executing `merge_sort( array, 21, 25 )`

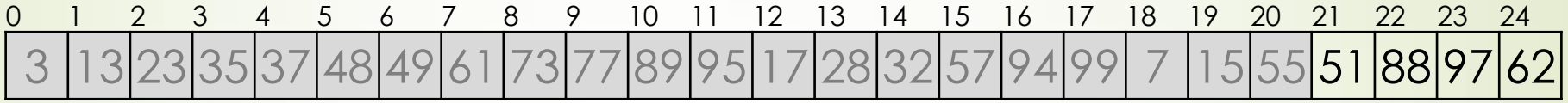


Now, $25 - 21 \leq 6$, so find we call insertion sort

```
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

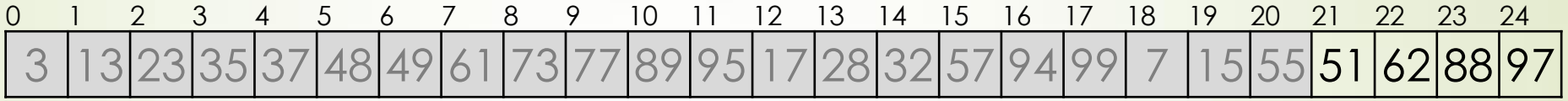
Insertion sort just sorts the entries from 21 to 24




```
insertion_sort( array, 21, 25 )
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

Insertion sort just sorts the entries from 21 to 24

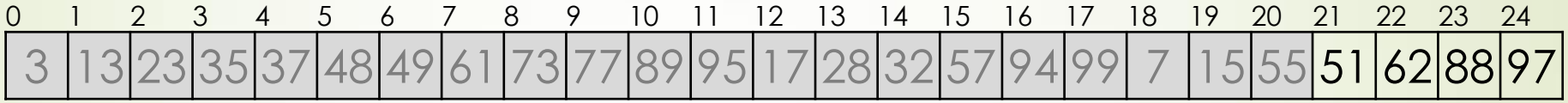


- This function call completes and so we exit

```
insertion_sort( array, 21, 25 )
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

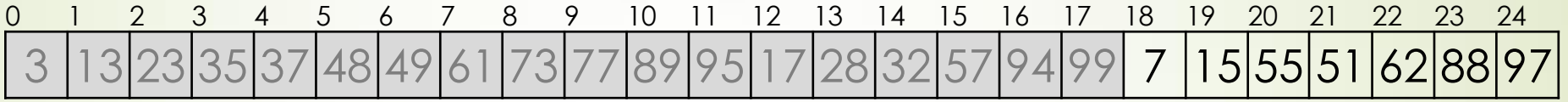
This call to `merge_sort` is now also finished, so it, too, exits




```
merge_sort( array, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 18, 25 )`



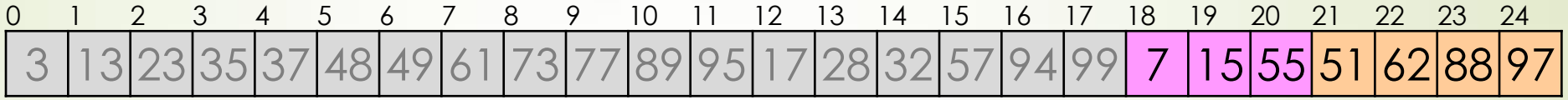
We continue calling

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

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 18, 21, 25 )`

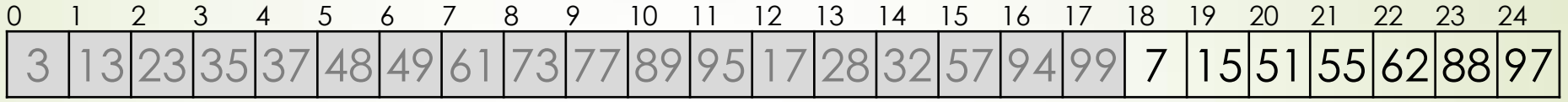


These two sub-arrays are merged together

```
merge( array, 18, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 18, 21, 25 )`



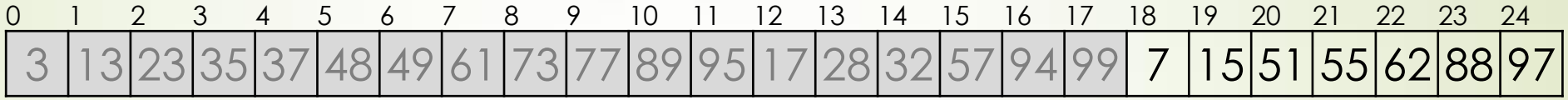
These two sub-arrays are merged together

- This function call exists

```
merge( array, 18, 21, 25 )
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 18, 25 )`



We are finished calling this function as well

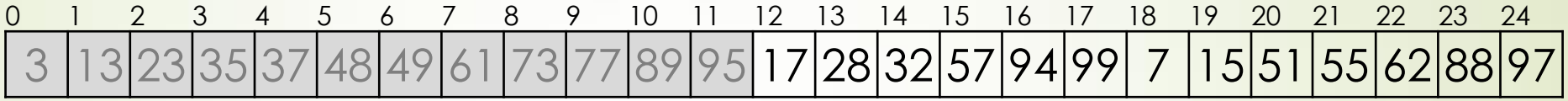
```
midpoint = (18 + 25)/2; // == 21
merge_sort( array, 18, 21 );
merge_sort( array, 21, 25 );
merge( array, 18, 21, 25 );
```

Consequently, we exit

```
merge_sort( array, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 12, 25 )`



We continue calling

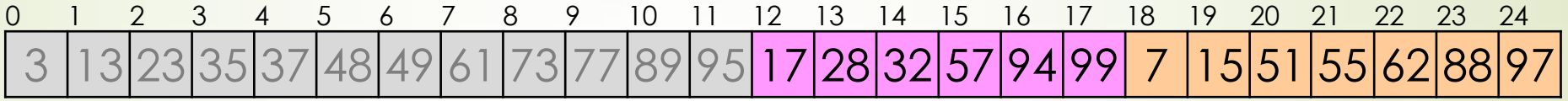
```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
merge_sort( array, 18, 25 );
merge( array, 12, 18, 25 );
```

`merge_sort( array, 12, 25 )`

`merge_sort( array,  0, 25 )`

Example

We are executing `merge( array, 12, 18, 25 )`

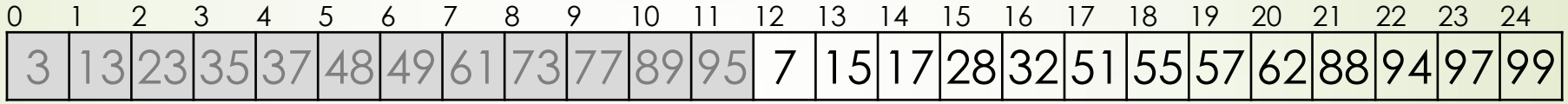


These two sub-arrays are merged together

```
merge( array, 12, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 12, 18, 25 )`



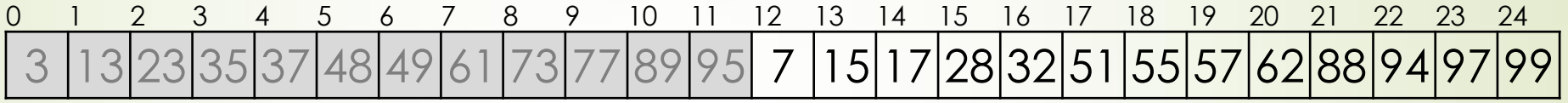
These two sub-arrays are merged together

- This function call exists

```
merge( array, 12, 18, 25 )
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 12, 25 )`



We are finished calling this function as well

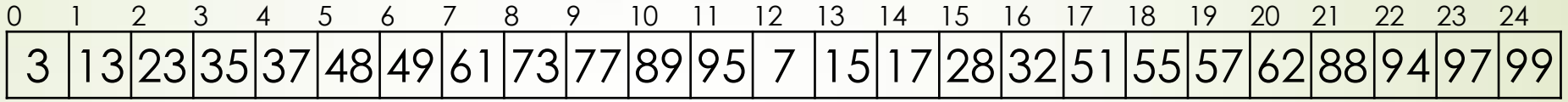
```
midpoint = (12 + 25)/2; // == 18
merge_sort( array, 12, 18 );
merge_sort( array, 18, 25 );
merge( array, 12, 18, 25 );
```

Consequently, we exit

```
merge_sort( array, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to continue executing `merge_sort( array, 0, 25 )`



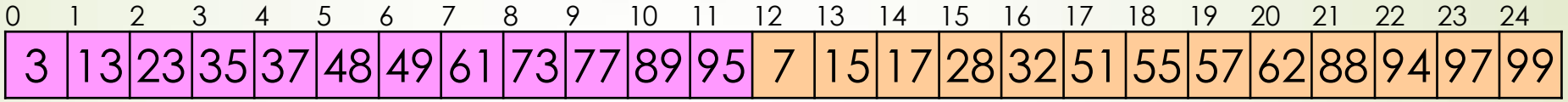
We continue calling

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

`merge_sort( array,  0, 25 )`

Example

We are executing `merge( array, 0, 12, 25 )`

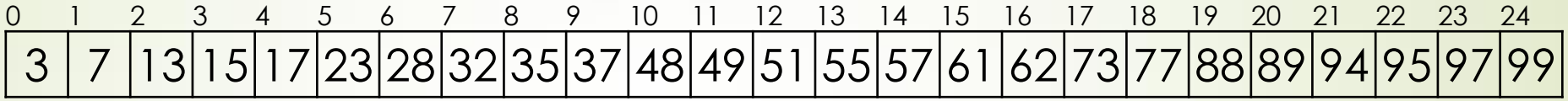


These two sub-arrays are merged together

```
merge( array, 0, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We are executing `merge( array, 0, 12, 25 )`



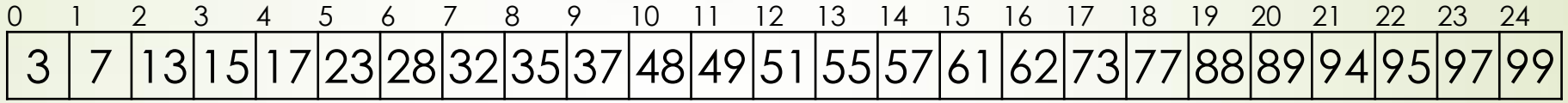
These two sub-arrays are merged together

- This function call exists

```
merge( array, 0, 12, 25 )
merge_sort( array,  0, 25 )
```

Example

We return to executing `merge_sort( array, 0, 25 )`



We are finished calling this function as well

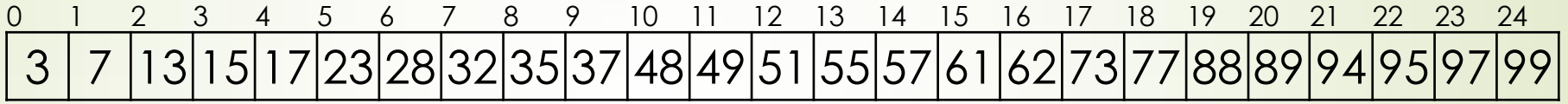
```
midpoint = (0 + 25)/2; // == 12
merge_sort( array, 0, 12 );
merge_sort( array, 12, 25 );
merge( array, 0, 12, 25 );
```

Consequently, we exit

`merge_sort( array,  0, 25 )`

Example

The array is now sorted



- Question: What is the run-time of this algorithm?

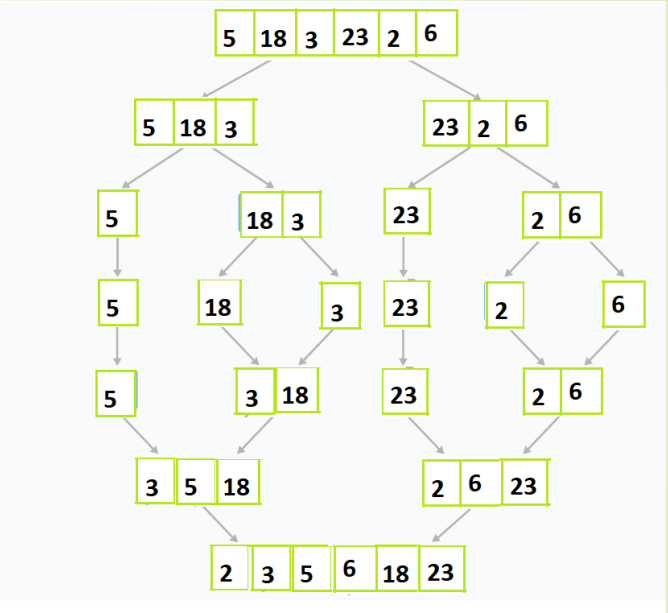
Run-time Analysis of Merge Sort

Thus, the time required to sort an array of size $n > 1$ is:

- the time required to sort the first half,
- the time required to sort the second half, and
- the time required to merge the two lists

That is:

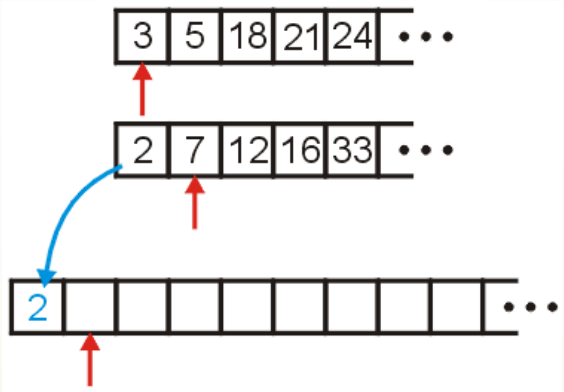
$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & n > 1 \end{cases}$$



Why is it not $O(n^2)$

When we are merging, we are comparing values

- What operation prevents us from performing $O(n^2)$ comparisons?
- During the merging process, if 2 came from the second half, it was only compared to 3 and it was not compared to any other of the other $n - 1$ entries in the first array



- In this case, we remove n inversions with one comparison

Comments

In practice, merge sort is faster than heap sort, though they both have the same asymptotic run times

Merge sort requires an additional array

- Heap sort does not require

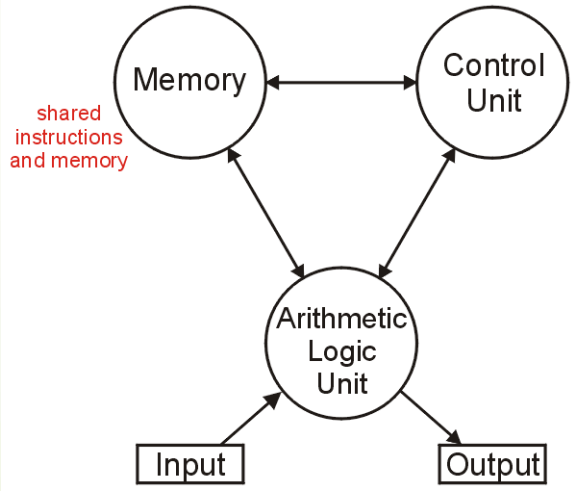
Next we see quick sort

- Faster, on average, than either heap or quick sort
- Requires $\mathbf{O}(n)$ additional memory

Merge Sort Use-cases

The (likely) first implementation of merge sort was on the ENIAC in 1945 by John von Neumann

- The creator of the *von Neumann architecture* used by all modern computers:



http://en.wikipedia.org/wiki/Von_Neumann

Merge Sort Use-cases

- Useful in **distributed** applications
- Used to design Tim sort (sorting algorithm runs in **Python** sort() API)
- Can be applied when data is in different locations like cache, main memory, external memory etc.
- A multi-way merge sort variant is used in **GNU** sorting utility
- Default sorting algorithm of arrays of object type in **Java** since version 7 onward.

Limitations

- Space inefficiency (not in-place)
- Lots of overhead in copying data between arrays and making new arrays

Summary

This topic covered merge sort:

- Divide an unsorted list into two equal or nearly equal sub lists,
- Sorts each of the sub lists by calling itself recursively, and then
- Merges the two sub lists together to form a sorted list
- Run-time analysis
- Applications