Sorting algorithms

Waqar Ahmad
Department of Computer Science
Syed Babar Ali School of Science and Engineering
Lahore University of Management Sciences (LUMS)

Outline

Sorting

- Insertion sort
- Merge sort
- Quick sort

Definition

Sorting is the process of:

– Taking a list of objects which could be stored in a linear order

$$(a_0, a_1, ..., a_{n-1})$$

e.*g*., numbers, and returning a reordering

$$(a'_0, a'_1, ..., a'_{n-1})$$

such that

$$a'_0 \le a'_1 \le \cdot\cdot\cdot \le a'_{n-1}$$

The conversion of an Abstract List into an Abstract Sorted List

In-place Sorting

Some sorting algorithms may be performed *in-place*, that is, with the allocation of at most $\Theta(1)$ additional memory (*e.g.*, fixed number of local variables)

Other sorting algorithms require the allocation of second array of equal size

– Requires $\Theta(n)$ additional memory

Run-time

The run time of the sorting algorithms we will look at fall into one of three categories:

$$\Theta(n) \qquad \Theta(n \log(n)) \qquad \mathbf{O}(n^2)$$

Run-time

$\mathbf{O}(n^2)$ sorting algorithms:

- Insertion sort, Bubble sort

Faster $\Theta(n \log(n))$ sorting algorithms:

- Heap sort, Quicksort, and Merge sort

Linear-time sorting algorithms

- Bucket sort and Radix sort
- We must make assumptions about the data

Lower-bound Run-time

Any sorting algorithm must examine each entry in the array at least once

- Consequently, all sorting algorithms must be $\Omega(n)$

We will not be able to achieve $\Theta(n)$ behaviour without additional assumptions

Optimal Sorting Algorithms

- There is no *optimal* sorting algorithm which can be used in all situations.
- Under various circumstances, different sorting algorithms will deliver optimal run-time and memory-allocation requirements

Insertion Sort

Background

Consider the following observations:

- A list with one element is sorted
- In general, if we have a sorted list of k items, we can insert a new item to create a sorted list of size $k + 1$

Background

For example, consider this sorted array containing eight sorted entries

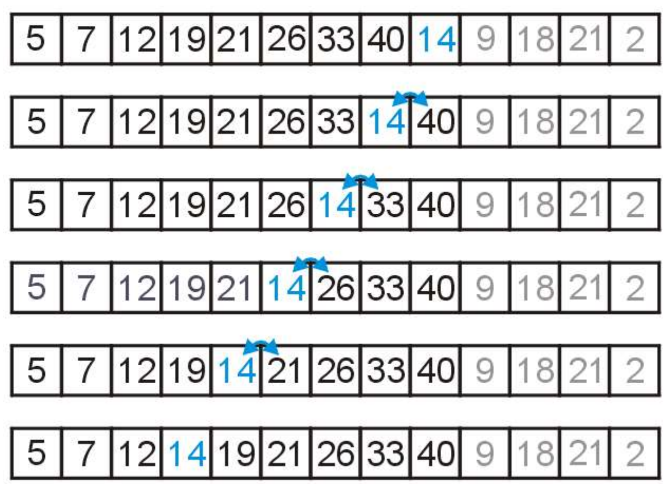
5	**7**	**12**	**19**	**21**	**26**	**33**	**40**	**14**	9	18	21	2

Suppose we want to insert 14 into this array leaving the resulting array sorted

Background

Starting at the end of the sorted list, if the number is greater than 14, copy it to the right

- Once an entry less than 14 is found, insert 14 into the resulting vacancy



The Algorithm

For any unsorted list:

- Treat the first element as a sorted list of size 1

Then, given a sorted list of size $k - 1$

- Insert the k^{th} item in the unsorted list into the sorted list

The Algorithm

```
for ( int j = k; j > 0; --j ) {
    if ( array[j - 1] > array[j] ) {
        std::swap( array[j - 1], array[j] );
    } else {
        // As soon as we don't need to swap, the (k + 1)st
        // is in the correct location
        break;
    }
}
```

Implementation and Analysis

This would be embedded in a function call such as

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Implementation and Analysis

Let's do a run-time analysis of this code

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Implementation and Analysis

The $\Theta(1)$-initialization of the outer for-loop is executed once

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Implementation and Analysis

This $\Theta(1)$- condition will be tested n times at which point it fails

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Implementation and Analysis

Thus, the inner for-loop will be executed a total of $n - 1$ times

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Implementation and Analysis

In the worst case, the inner for-loop is executed a total of k times

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
                // is in the correct location
                break;
            }
        }
    }
}
```

Implementation and Analysis

The body of the inner for-loop runs in $\Theta(1)$ in either case

```
template <typename Type>
void insertion_sort( Type *const array, int const n ) {
    for ( int k = 1; k < n; ++k ) {
        for ( int j = k; j > 0; --j ) {
            if ( array[j - 1] > array[j] ) {
                std::swap( array[j - 1], array[j] );
            } else {
                // As soon as we don't need to swap, the (k + 1)st
               // is in the correct location
               break;
            }
        }
    }
}
```

Thus, the worst-case run time is

$$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

Summary

Insertion Sort:

- Insert new entries into growing sorted lists
- Run-time analysis: $\mathbf{O}(n^2)$
  - Insertion sort is used to sort small lists but isn't the most efficient method for handling large lists.
  - Question: what if the list is already sorted?

Merge Sort

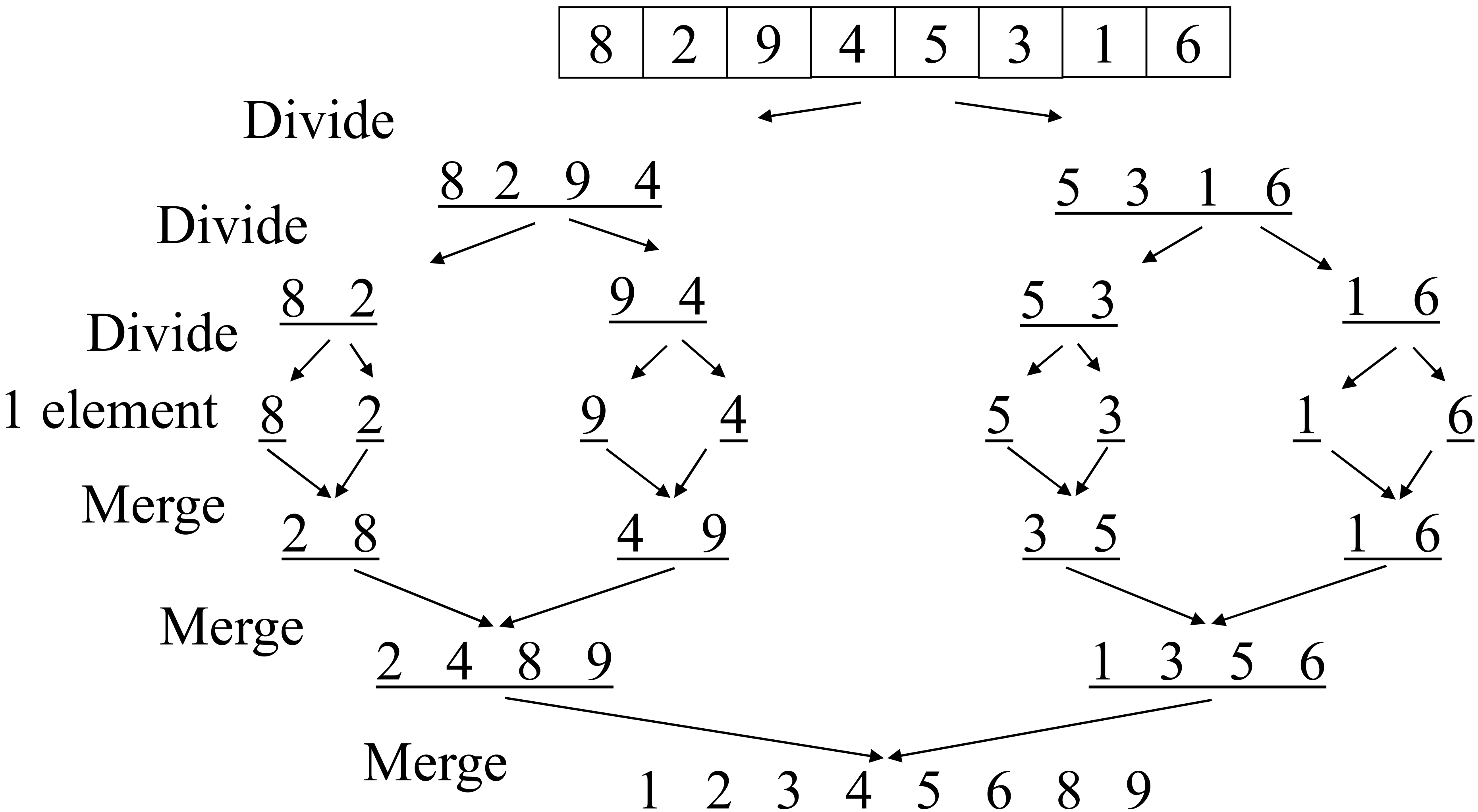
Merge Sort

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

A *divide-and-conquer* algorithm

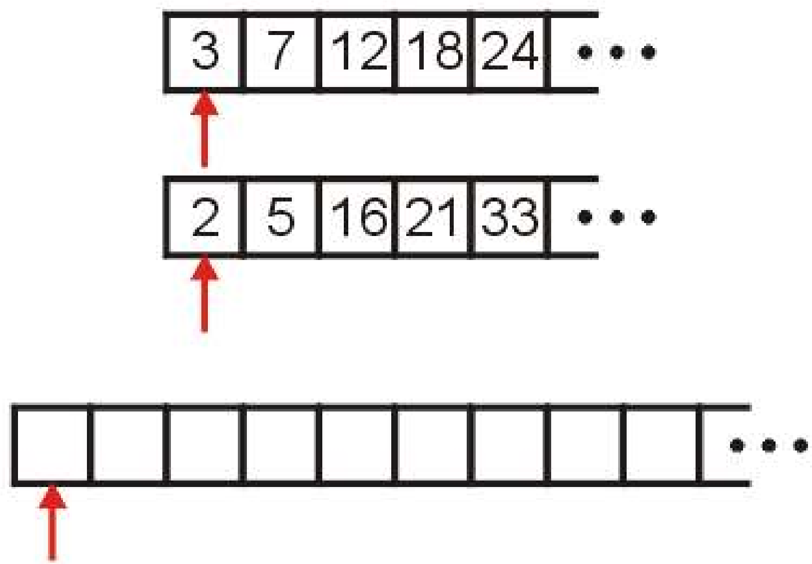
Merge Sort Example



Merging Example

Consider the two sorted arrays and an empty array

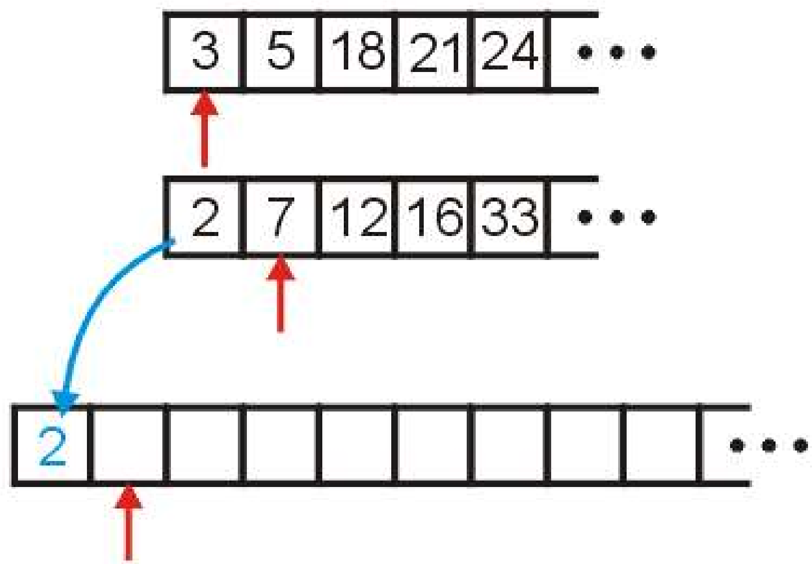
Define three indices at the start of each array



Merging Example

We compare 2 and 3: 2 < 3

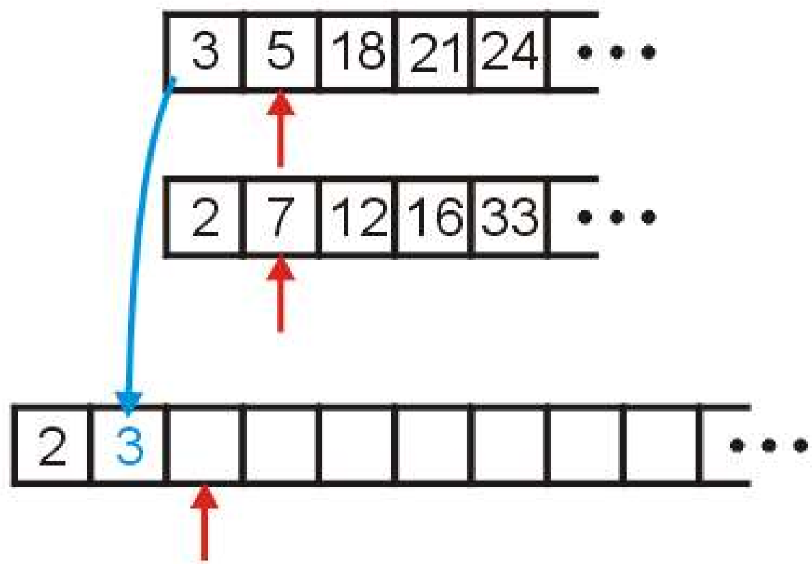
- Copy 2 down
- Increment the corresponding indices



Merging Example

We compare 3 and 7

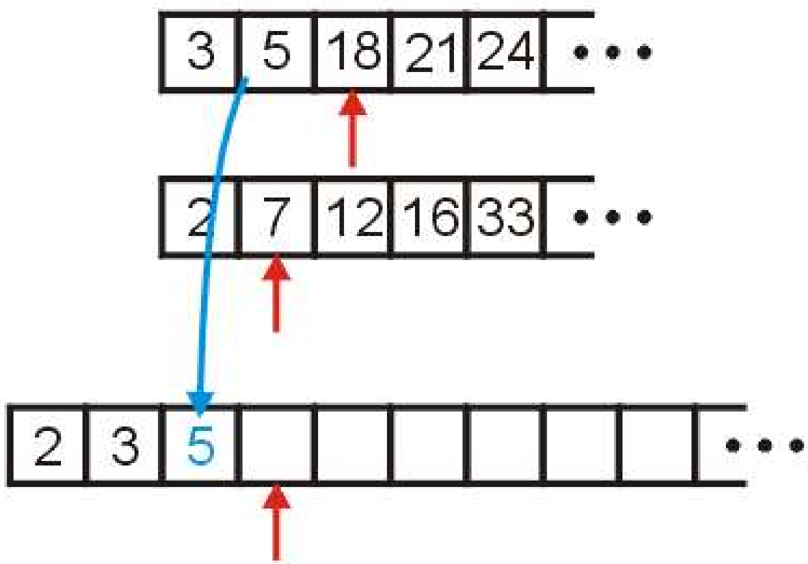
- Copy 3 down
- Increment the corresponding indices



Merging Example

We compare 5 and 7

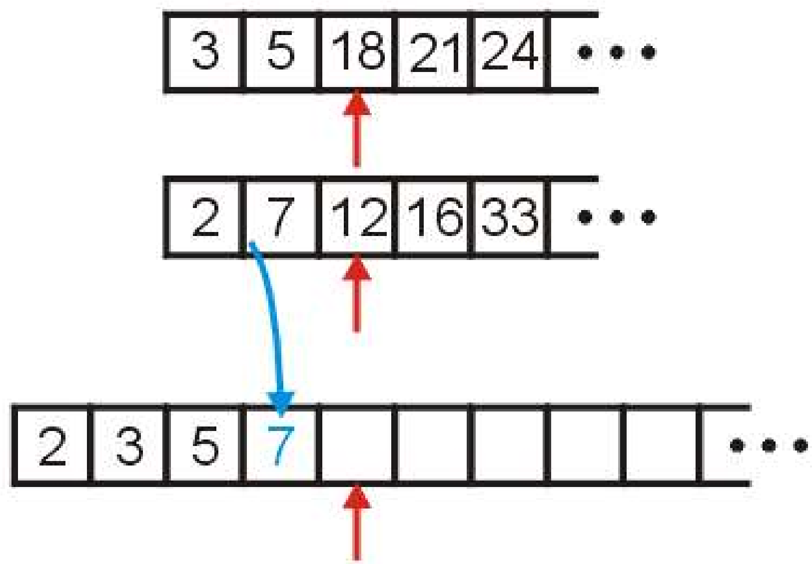
- Copy 5 down
- Increment the appropriate indices



Merging Example

We compare 18 and 7

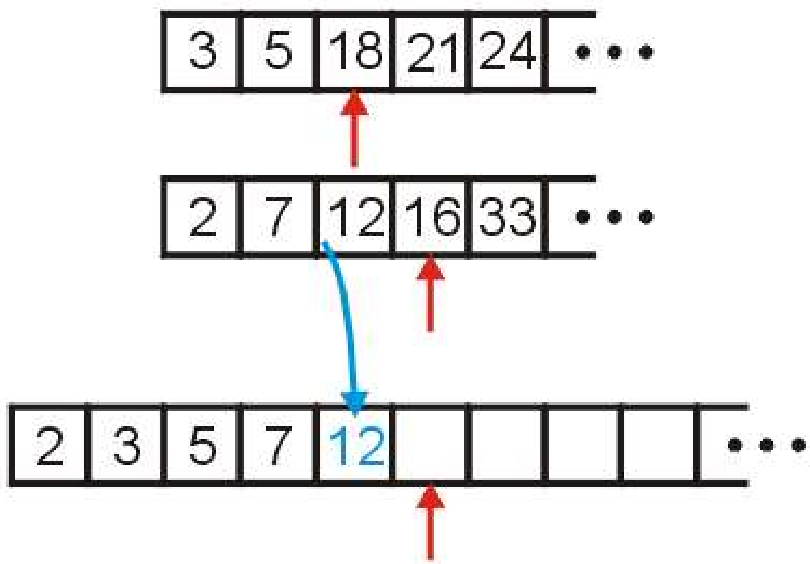
- Copy 7 down
- Increment...



Merging Example

We compare 18 and 12

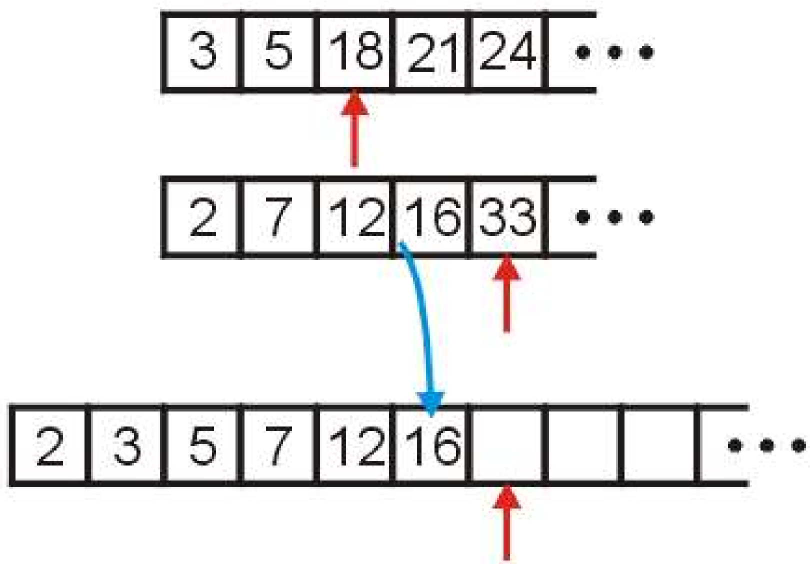
- Copy 12 down
- Increment...



Merging Example

We compare 18 and 16

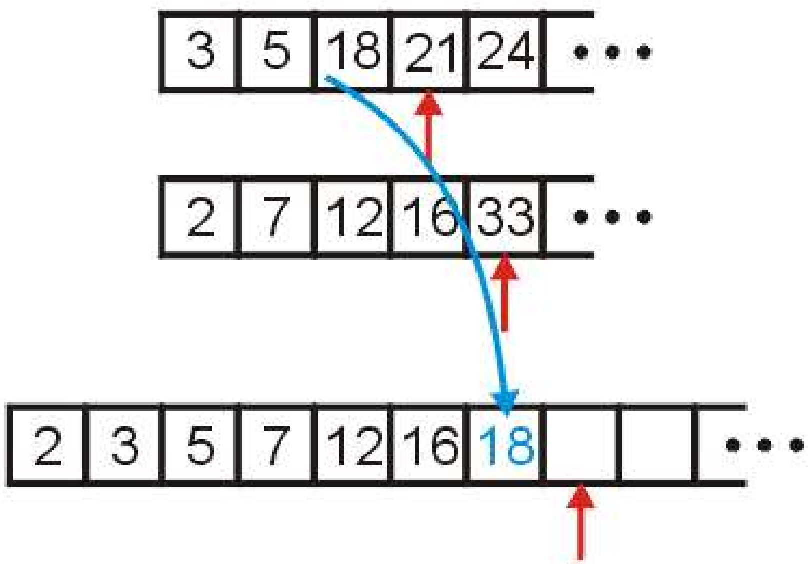
- Copy 16 down
- Increment...



Merging Example

We compare 18 and 33

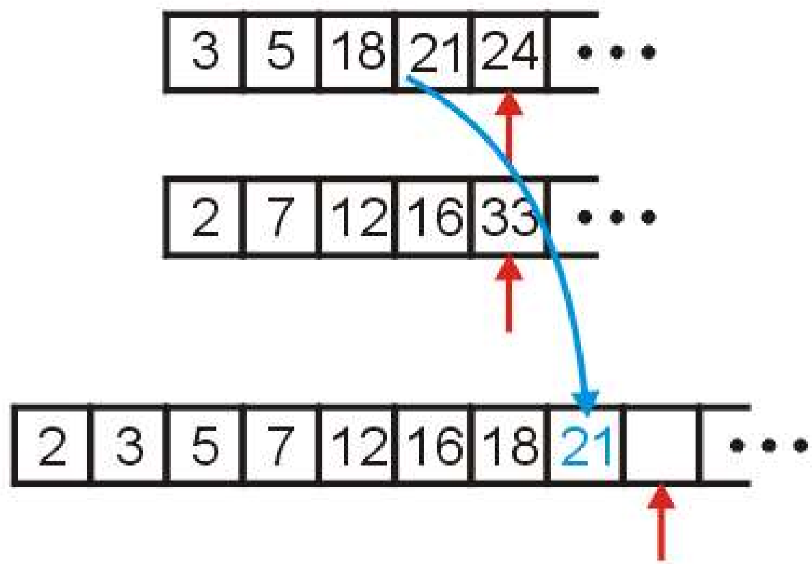
- Copy 18 down
- Increment...



Merging Example

We compare 21 and 33

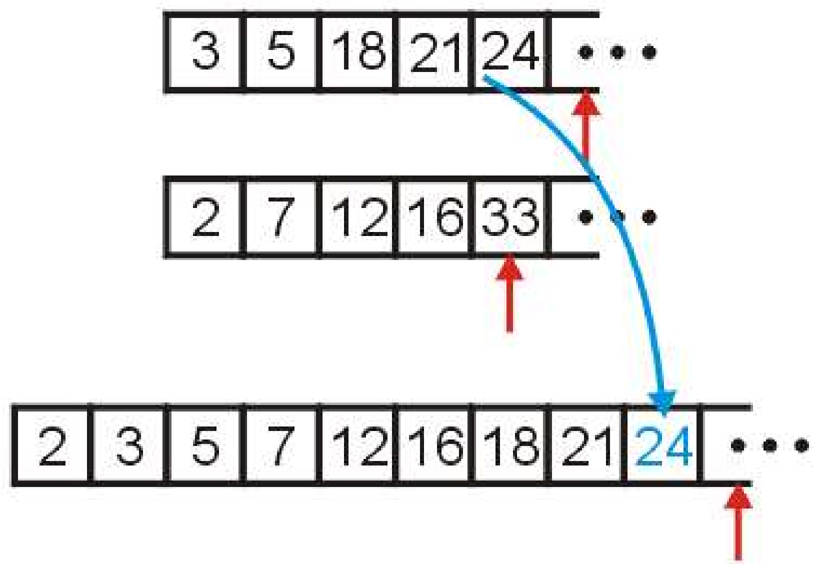
- Copy 21 down
- Increment...



Merging Example

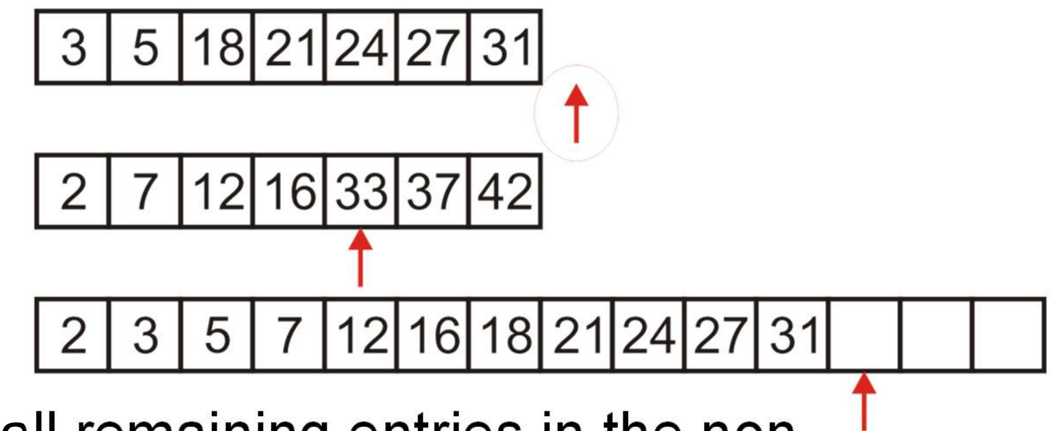
We compare 24 and 33

- Copy 24 down
- Increment...

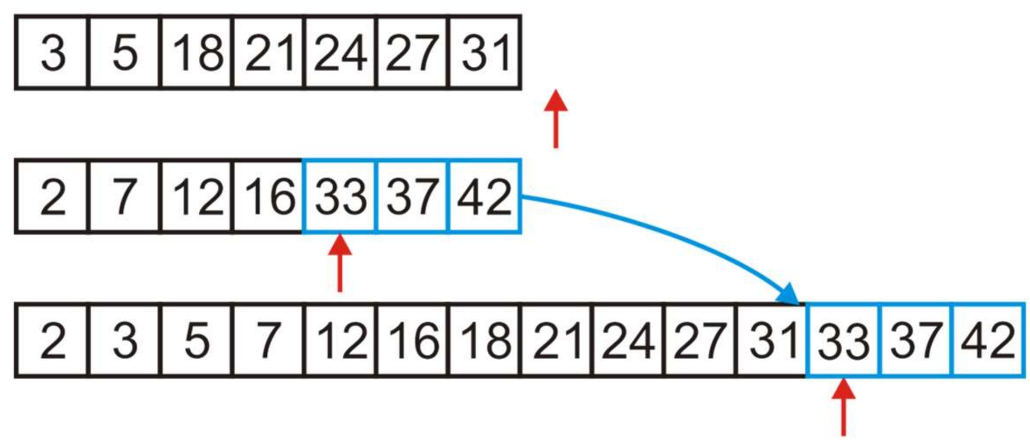


Merging Example

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



Analysis of merging

- The sorted arrays, `array1` and `array2`, are of size `n1` and `n2`, respectively, and,
- We have an empty array, `arrayout`, of size `n1 + n2`
- Merging may be performed in $\Theta(n_1 + n_2)$ time

We can say that the run time is $\Theta(n)$

Problem: We cannot merge two arrays in-place

- This algorithm require the allocation of a new array
- Therefore, the memory requirements are also $\Theta(n)$

Merge Sort 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

Question:

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

Answer:

- 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 already sorted

Implementation : Merge function

We need to implement a merge 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: Merge Sort function

We also need to implement a merge_sort function

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

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

- Find the mid-point,
- Call merge sort recursively on each of the halves, and
- Merge the sorted lists

Example

Consider the following unsorted array of 25 entries

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

Example

We call `merge_sort( array, 0, 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

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

Example

We are calling `merge_sort( array, 0, 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

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 )`

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

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 )`

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

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

Example

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

The elements from 0 to 5 are recursively sorted. Not all steps are shown here.

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

Example

The call to `merge_sort` (0, 6) is finished.

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

```
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 )`

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

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 )`

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

```
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

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

The elements from 6 to 11 are recursively sorted. Not all steps are shown here.

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

Example

This call to `merge_sort (6, 12)` is now also finished, so it, too, exits

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

```
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 )`

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 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 )`

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

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 )`

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

These two sub-arrays are merged together

- The function call `merge( array, 0, 6, 12 )` 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 )`

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

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 )`

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

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 )`

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

The process continues until this part is also sorted.

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

Example

We return to executing `merge_sort( array, 12, 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
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

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 )`

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

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 )`

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

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 )`

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

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 )`

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

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

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

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

T(n) = $\Theta(n \log(n))$

Quick Sort

Quicksort

- Merge sort splits the array into sub-lists and sorts them
  - The larger problem is split into two sub-problems based on *location* in the array

- Quick sort:
  - Chose an object (pivot) in the array and partition the remaining objects into two groups relative to the chosen object

Quicksort

For example, given

80	38	95	84	66	10	79	**44**	26	87	96	12	43	81	3

we can select the middle entry, 44, and sort the remaining entries into two groups, those less than 44 and those greater than 44:

38	10	26	12	43	3	**44**	80	95	84	66	79	87	96	81

Notice that 44 is now in the correct location if the list was sorted

– Proceed by applying the algorithm recursively to the first six and last eight entries

Run-time analysis

In the best case, the list will be split into two approximately equal sub-lists, and thus, the run time could be very similar to that of merge sort: $\Theta(n \log(n))$

What happens if we don’t get that lucky?

Worst-case scenario

Suppose we choose the first element as our pivot.

Using 2, we partition into

2	80	38	95	84	66	10	79	26	87	96	12	43	81	3

We still have to sort a list of size $n - 1$

The run time is $\mathrm{T}(n) = \mathrm{T}(n - 1) + \Theta(n) = \Theta(n^2)$

– Thus, the run time goes from $n \log(n)$ to n^2

Worst-case scenario

Our goal is to choose the median element in the list as our pivot:

80	38	95	84	**66**	10	79	2	26	87	96	12	43	81	3

Unfortunately, it's difficult to find

Median-of-three

It is difficult to find the median so consider another strategy:

– Choose the median of the first, middle, and last entries in the list

This will usually give a better approximation of the actual median

80	38	95	84	99	10	79	44	26	87	96	12	43	81	3

Median-of-three

Sorting the elements based on 44 (pivot), results in two sub-lists, each of which must be sorted (again, using quicksort)

Select 26 to partition the first sub-list:

38	10	26	12	43	3	44	80	95	84	99	79	87	96	81

Select 81 to partition the second sub-list:

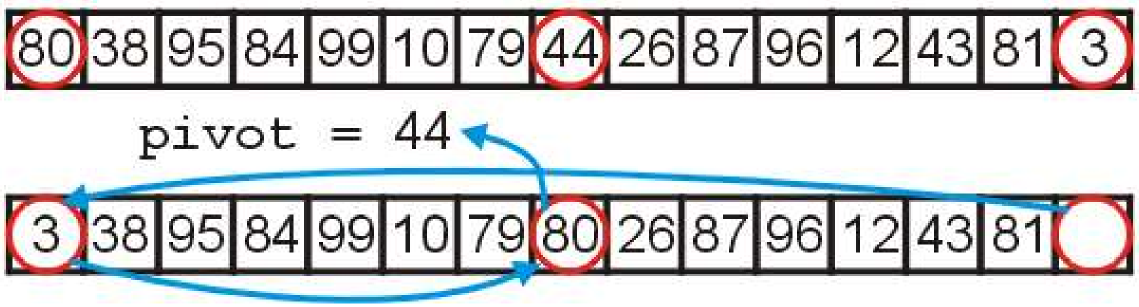
38	10	26	12	43	3	44	80	95	84	99	79	87	96	81

Implementation

First, we have already examined the first, middle, and last entries and chosen the median of these to be the pivot

In addition, we can:

- move the smallest entry to the first entry
- move the largest entry to the middle entry



Implementation

Next, recall that our goal is to partition all remaining elements based on whether they are smaller than or greater than the pivot

We will find two entries:

- One larger than the pivot (staring from the front)
- One smaller than the pivot (starting from the back)

which are out of order and then swap them

Implementation

Continue doing so until the appropriate entries you find are actually in order

The index to the larger entry we found would be the first large entry in the list (as seen from the left)

Therefore, we could move this entry into the last entry of the list

We can fill this spot with the pivot

Quicksort example

Consider the following unsorted array of 25 entries

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

Quicksort example

We call `quicksort( array, 0, 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

`quicksort( array,  0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 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**

Find the midpoint and the pivot

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

`quicksort( array,  0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 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	**62**	99	17	32	94	28	15	55	7	51	88	97	

Find the midpoint and the pivot

```
midpoint = (0 + 25)/2; // == 12
pivot = 57;
```

`quicksort( array, 0, 25 )`

Quicksort example

We are calling `quicksort( array, 0, 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	62	99	17	32	94	28	15	55	7	51	88	97	

Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	62	99	17	32	94	28	15	55	7	**51**	88	97	

Searching forward and backward:

```
low = 1;
high = 21;
```

```
pivot = 57;
```

```
quicksort( array, 0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	61	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	7	77	88	97	

Searching forward and backward:

```
low = 1;
high = 21;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	**61**	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	**7**	77	88	97	

Continue searching

```
low = 4;
high = 20;
```

```
pivot = 57;
```

```
quicksort( array, 0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	61	77	88	97	

Continue searching

```
low = 4;
high = 20;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	73	23	95	3	89	37	62	99	17	32	94	28	15	55	61	77	88	97	

Continue searching

```
low = 6;
high = 19;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	95	3	89	37	62	99	17	32	94	28	15	73	61	77	88	97	

Continue searching

```
low = 6;
high = 19;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	**95**	3	89	37	62	99	17	32	94	28	**15**	73	61	77	88	97	

Continue searching

```
low = 8;
high = 18;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	89	37	62	99	17	32	94	28	95	73	61	77	88	97	

Continue searching

```
low = 8;
high = 18;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	**89**	37	62	99	17	32	94	**28**	95	73	61	77	88	97	

Continue searching

```
low = 10;
high = 17;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	28	37	62	99	17	32	94	89	95	73	61	77	88	97	

Continue searching

```
low = 10;
high = 17;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	28	37	62	99	17	32	94	89	95	73	61	77	88	97	

Continue searching

```
low = 12;
high = 15;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	28	37	32	99	17	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 12;
high = 15;
```

Swap them

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	28	37	32	**99**	**17**	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 13;
high = 14;
```

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	28	37	32	17	99	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 13;
high = 14;
```

Swap them

```
pivot = 57;
```

```
quicksort( array, 0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	28	37	32	**17**	**99**	62	94	89	95	73	61	77	88	97	

Continue searching

```
low = 14;
high = 13;
```

Now, `low > high`, so we stop

```
pivot = 57;
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	28	37	32	17	**57**	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 14;
high = 13;
```

Now, `low > high`, so we stop

```
pivot = 57;
```

```
quicksort( array, 0, 25 )
```

Quicksort example

We are calling `quicksort( array, 0, 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	51	49	35	7	48	55	23	15	3	28	37	32	17	**57**	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively on the first half

```
quicksort( array, 0, 14 );
```

```
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	51	49	35	7	48	55	23	15	3	28	37	32	17	57	62	94	89	95	73	61	77	88	97	99

Find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	51	49	35	7	48	55	**23**	15	3	28	37	32	**17**	57	62	94	89	95	73	61	77	88	97	99

Find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
pivot = 17
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Find the midpoint and the pivot

```
midpoint = (0 + 14)/2; // == 7
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	51	49	35	7	48	55	23	15	3	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 1;
high = 9;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	3	49	35	7	48	55	23	15	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 1;
high = 9;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	3	49	35	7	48	55	23	15	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 2;
high = 8;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	3	15	35	7	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 2;
high = 8;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	3	15	**35**	**7**	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 3;
high = 4;
```

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	3	15	7	35	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

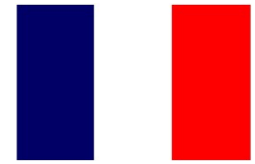
```
low = 3;
high = 4;
```

Swap them

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example



We are executing `quicksort( array, 0, 14 )`

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	3	15	7	35	48	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 4;
high = 3;
```

Now, `low > high`, so we stop

```
pivot = 17;
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 14 )`

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	3	15	7	**17**	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively

```
quicksort( array, 0, 4 );
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 0, 4 )`

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	3	15	7	17	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

Now, when the array size as become quite small, e.g., ≤ 6, we can call insertion sort to sort the array.

```
quicksort( array,  0,  4 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 0, 14 )`

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**	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

```
quicksort( array, 0,  4 );
quicksort( array, 5, 14 );
```

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

```
midpoint = (5 + 14)/2; // == 9
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	**48**	55	23	49	**51**	28	37	32	**35**	57	62	94	89	95	73	61	77	88	97	99

```
midpoint = (5 + 14)/2; // == 9
pivot = 48
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	**35**	55	23	49	**51**	28	37	32		57	62	94	89	95	73	61	77	88	97	99

```
midpoint = (5 + 14)/2; // == 9
pivot = 48
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Starting from the front and back:

- Find the next element greater than the pivot
- The last element less than the pivot

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	55	23	49	51	28	37	32		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 6;
high = 12;
```

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	49	51	28	37	55		57	62	94	89	95	73	61	77	88	97	99

Searching forward and backward:

```
low = 6;
high = 12;
```

Swap them

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	49	51	28	37	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	37	51	28	49	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

Swap them

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	37	**51**	**28**	49	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	37	28	51	49	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

Swap them

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	37	**28**	**51**	49	55		57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

Now, `low > high`, so we stop

```
pivot = 48;
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	37	28	**48**	49	55	51	57	62	94	89	95	73	61	77	88	97	99

Continue searching

```
low = 8;
high = 11;
```

Now, `low > high`, so we stop

`pivot = 48;`

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	37	28	**48**	49	55	51	57	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively on the first half

```
quicksort( array, 5, 10 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We now are calling `quicksort( array, 5, 14 )`

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	35	32	23	37	28	**48**	49	55	51	57	62	94	89	95	73	61	77	88	97	99

We now begin calling quicksort recursively

```
quicksort( array, 5, 10 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 5, 10 )`

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	35	32	23	37	28	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

Now, when the array size as become quite small, e.g., ≤ 6, we can call insertion sort to sort the array.

```
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 5 to 9

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	35	32	23	37	28	48	49	55	51	57	62	94	89	95	73	61	77	88	97	99

```
insertion_sort( array, 5, 10 )
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 5 to 9

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	55	51	57	62	94	89	95	73	61	77	88	97	99

– This function call completes and so we exit

```
insertion_sort( array, 5, 10 )
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

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

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	55	51	57	62	94	89	95	73	61	77	88	97	99

```
quicksort( array,  5, 10 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 5, 14 )`

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	55	51	57	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half

```
quicksort( array, 5, 10 );
quicksort( array, 6, 14 );
```

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 11, 15 )`

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	55	51	57	62	94	89	95	73	61	77	88	97	99

Now, when the array size as become quite small, e.g., ≤ 6, we can call insertion sort to sort the array.

```
quicksort( array,  6, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 11 to 14

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	62	94	89	95	73	61	77	88	97	99

– This function call completes and so we exit

```
insertion_sort( array, 11, 14 )
quicksort( array, 11, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

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

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	62	94	89	95	73	61	77	88	97	99

```
quicksort( array, 11, 14 )
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

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

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	62	94	89	95	73	61	77	88	97	99

```
quicksort( array,  5, 14 )
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

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

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	62	94	89	95	73	61	77	88	97	99

```
quicksort( array,  0, 14 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 0, 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
3	7	13	15	17	23	28	32	35	37	48	49	51	55	**57**	62	94	89	95	73	61	77	88	97	99

We continue calling quicksort recursively on the second half in a similar fashion until the whole array is sorted.

```
quicksort( array, 0, 14 );
quicksort( array, 15, 25 );
```

```
quicksort( array,  0, 25 )
```

Run-time Summary

	Average Run Time	**Worst-case Run Time**
Heap Sort	$\Theta(n \log(n))$	$\Theta(n \log(n))$
Merge Sort	$\Theta(n \log(n))$	$\Theta(n \log(n))$
Quicksort	$\Theta(n \log(n))$	$O(n^2)$

Homework

- Study the following sorting algorithms yourself
  - Bubble sort
  - Selection sort

References and Acknowledgements

- The content provided in the slides are borrowed from different sources including Goodrich's book on Data Structures and Algorithms in C++, Cormen's book on Introduction to Algorithms, Weiss's book , Data Structures and Algorithm Analysis in C++, 3rd Ed., Donald E. Knuth's book, *The Art of Computer Programming,* Algorithms and Data Structures at University of Waterloo (https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/) and https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html.
- The primary source of slides is https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials, courtesy of Douglas Wilhelm Harder.