CS101 Algorithms and Data Structures

Divide and Conquer, Merge/Quick Sort

Textbook Ch 4, 7

Outline

- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort

Outline

This topic covers merge sort

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

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

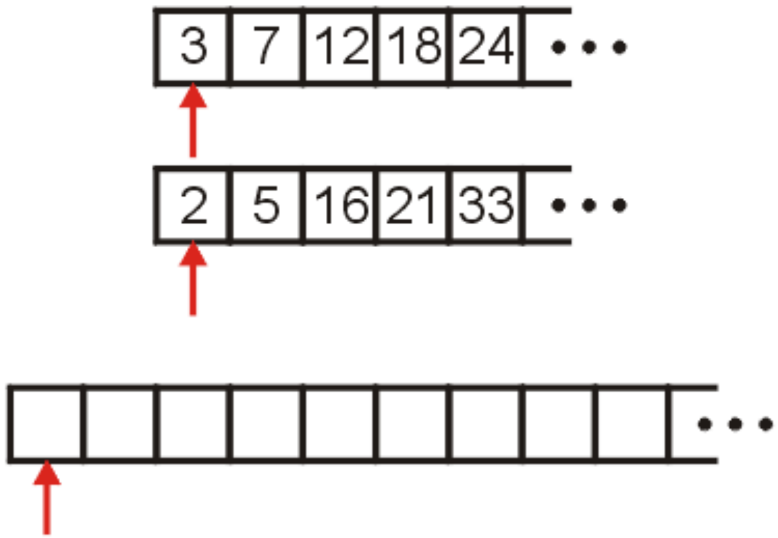
This strategy is called *divide-and-conquer*

Question: How can we merge two sorted sub-lists into a single sorted list?

Merging Example

Consider the two sorted arrays and an empty array

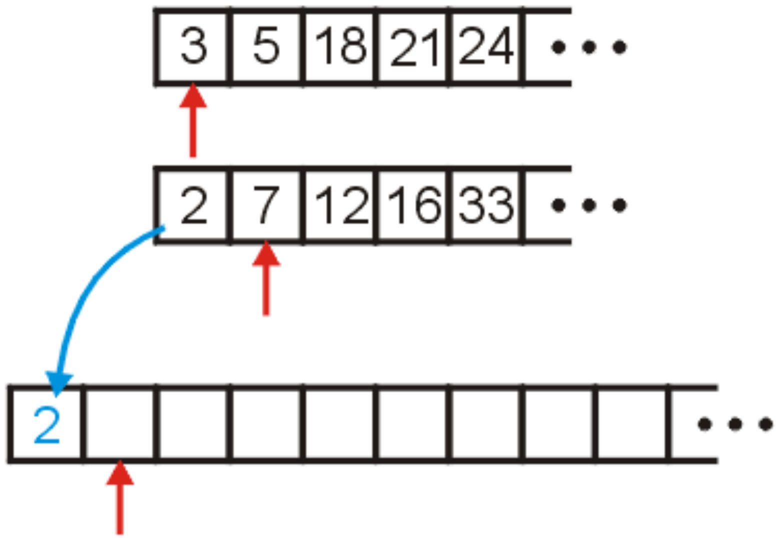
Define three indices at the start of each array



Merging Example

We compare 2 and 3: 2 < 3

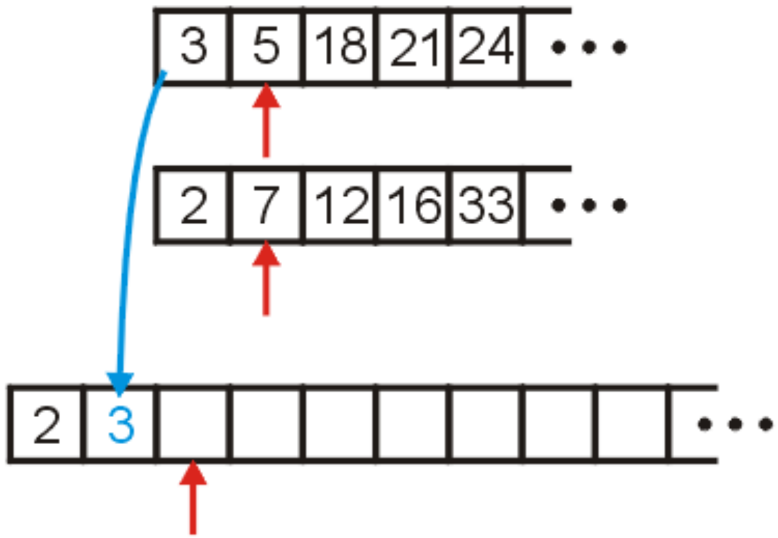
- Copy 2 down
- Increment the corresponding indices



Merging Example

We compare 3 and 7

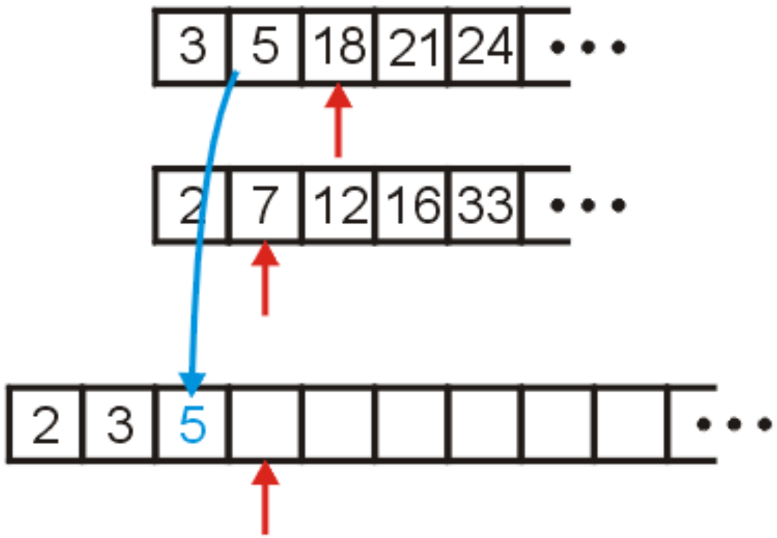
- Copy 3 down
- Increment the corresponding indices



Merging Example

We compare 5 and 7

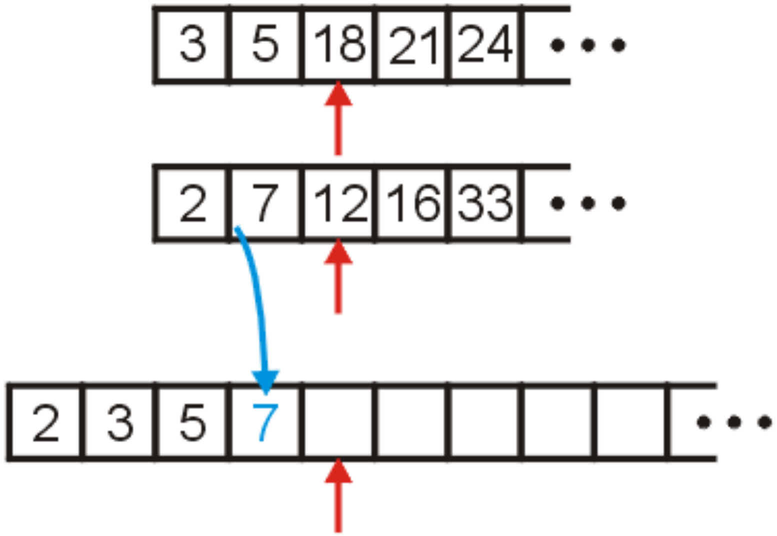
- Copy 5 down
- Increment the appropriate indices



Merging Example

We compare 18 and 7

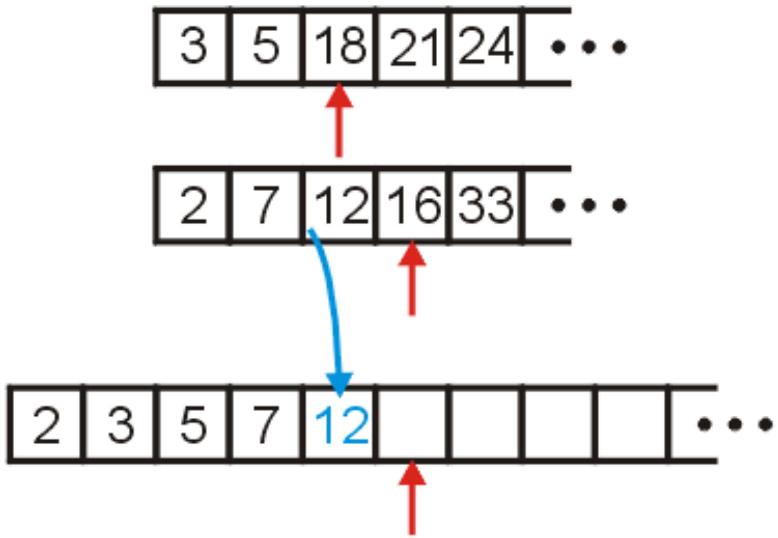
- Copy 7 down
- Increment...



Merging Example

We compare 18 and 12

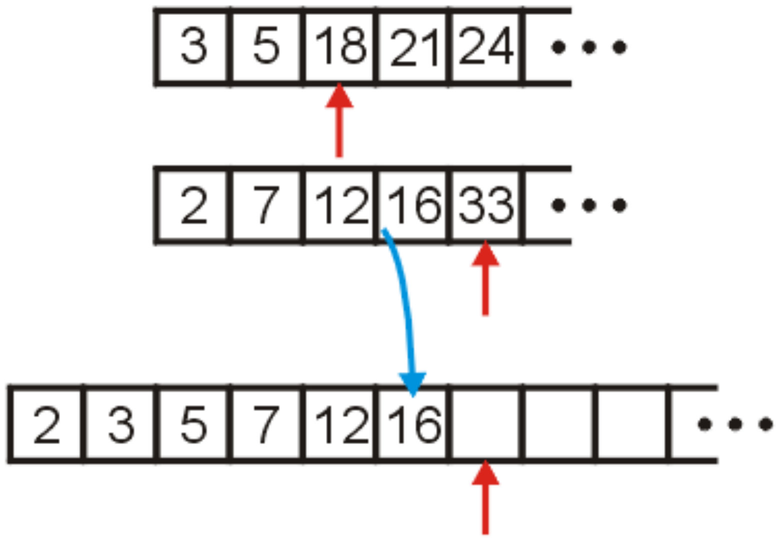
- Copy 12 down
- Increment...



Merging Example

We compare 18 and 16

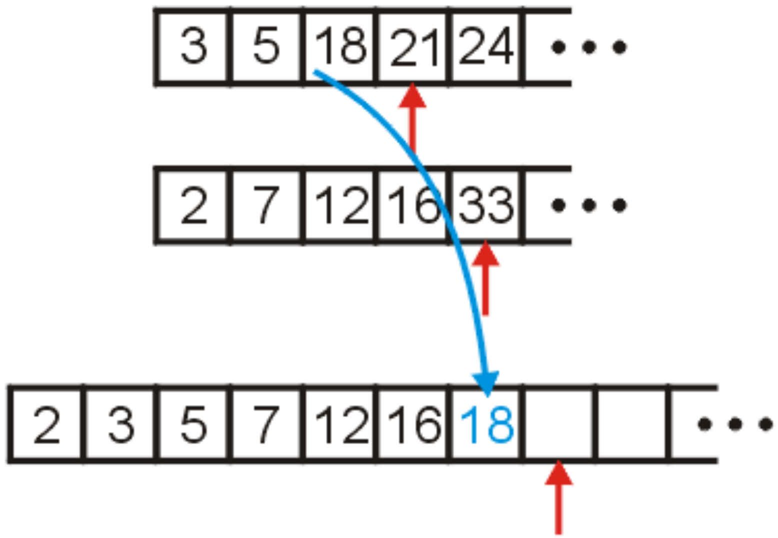
- Copy 16 down
- Increment...



Merging Example

We compare 18 and 33

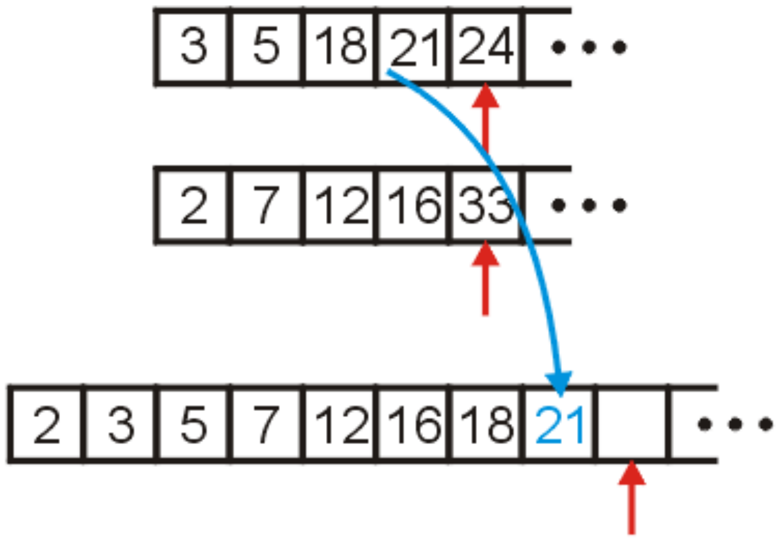
- Copy 18 down
- Increment...



Merging Example

We compare 21 and 33

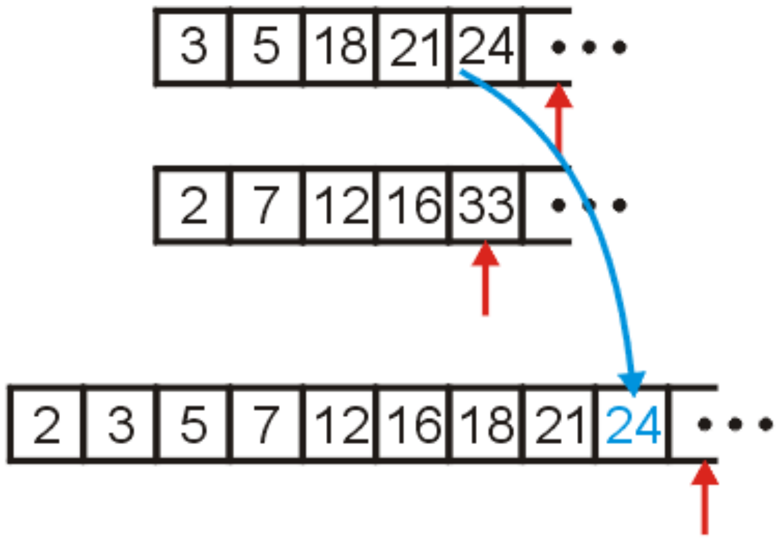
- Copy 21 down
- Increment...



Merging Example

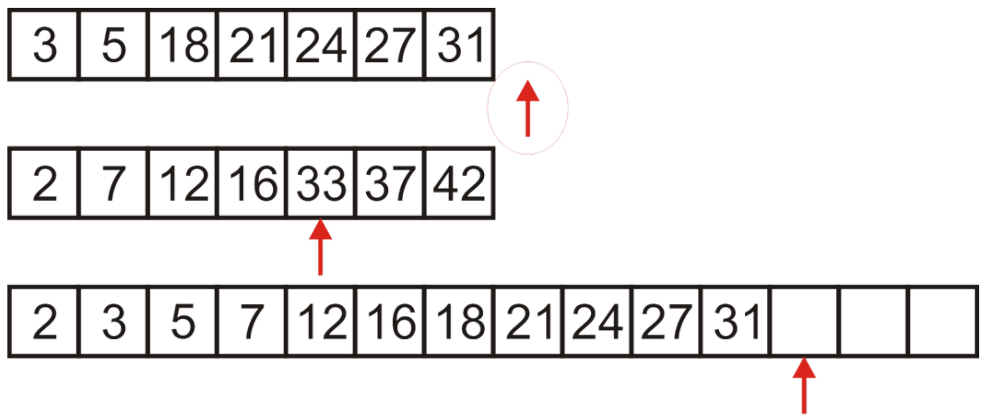
We compare 24 and 33

- Copy 24 down
- Increment...

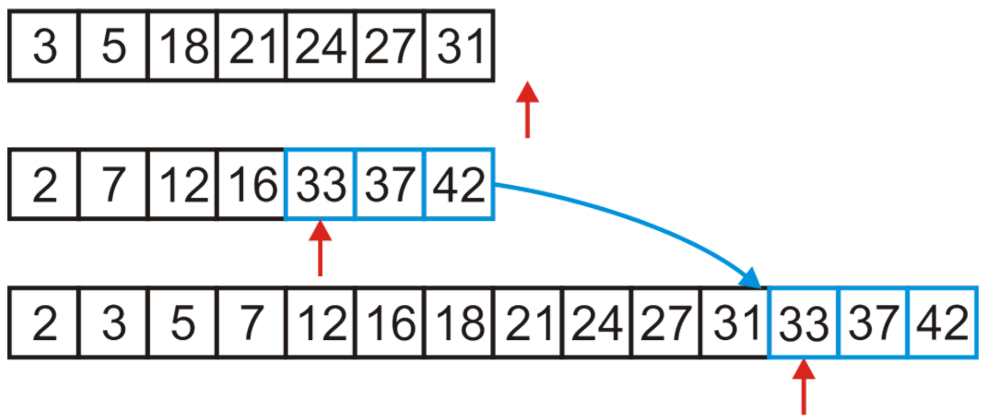


Merging Example

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

Time: we have to copy $n_1 + n_2$ elements

- Hence, merging may be performed in $\Theta(n_1 + n_2)$ time
- If the arrays are approximately the same size, $n = n_1 \approx n_2$, we can say that the run time is $\Theta(n)$

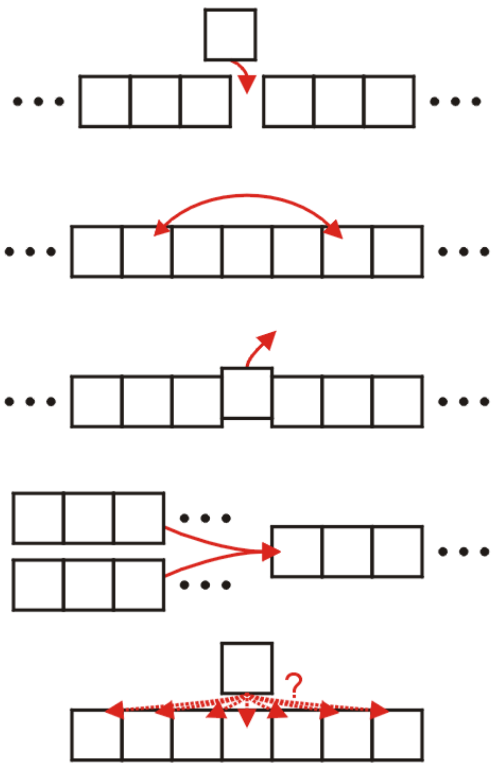
Space: we cannot merge two arrays in-place

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

The Algorithm

Recall the five sorting techniques:

- Insertion
- Exchange
- Selection
- Merging
- Distribution



Clearly merge sort falls into the fourth category

The Algorithm

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

In practice:

- If the list size is less than a threshold, use an algorithm like insertion sort 与寄存器互换相关，更快
- Otherwise:
  - Divide…

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,

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

a call to

```
void merge( array, 14, 20, 26 );
```

merges the two sub-lists

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

forming

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

Implementation

We implement a 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`

- 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

```
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 );
    }
}
```

Implementation

Like merge sort, insertion sort will sort 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: ;
    }
}
```

Example

Consider the following is of 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

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

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

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 )`

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

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 )`

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

Now, $6 - 0 \le 6$, 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`

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

```
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`

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

– 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

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

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

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( 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

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

– 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

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

– 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 )`

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

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 )`

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

Now, $18 - 12 \le 6$, 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

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

```
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

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

– 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

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

```
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 )`

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

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 )`

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

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 )`

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

Now, $21 - 18 \le 6$, 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`

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( 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`

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

– 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

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

```
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 )`

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

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 )`

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

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

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( 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

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

- 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

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

```
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 )`

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

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 )`

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

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 )`

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

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 )`

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

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 )`

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

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 )`

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

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 )`

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
- 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 )`

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

The time required to sort an array of size $n > 1$ is:

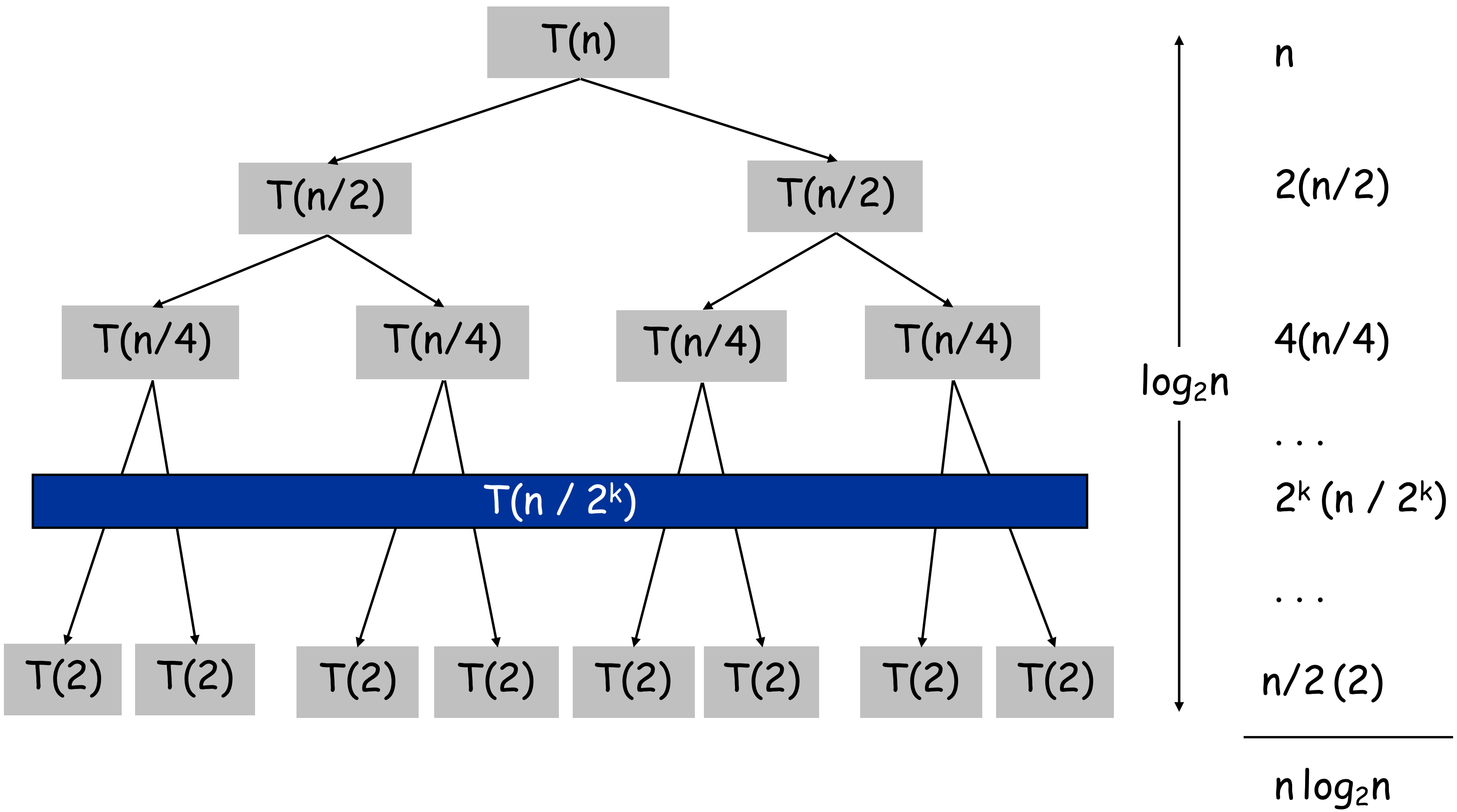
- the time required to sort the first half,
- the time required to sort the second half, and
- the time required to merge the two lists

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

Solution: $\mathrm{T}(n) = \Theta(n \ln(n))$

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Run-time Summary

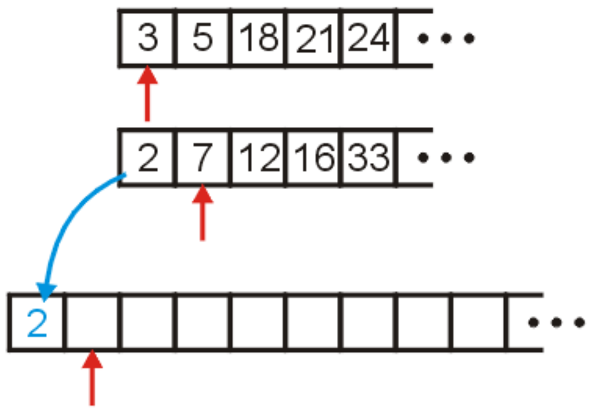
The following table summarizes the run-times of merge sort

Case	Run Time	Comments
Worst	$\Theta(n \ln(n))$	No worst case
Average	$\Theta(n \ln(n))$	
Best	$\Theta(n \ln(n))$	No best case

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

The (likely) first proposal of merge sort was by John von Neumann in 1945

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

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

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

Outline

- Insertion sort
- Bubble sort
- Heap sort
- Merge sort
- Quicksort

Outline

In this topic we will look at quicksort:

- The idea behind the algorithm
- The run time and worst-case scenario
- Strategy for avoiding the worst-case: median-of-three
- Implementing quicksort in place
- Examples

Strategy

We have seen two $\Theta(n \ln(n))$ sorting algorithms:

- Heap sort which allows in-place sorting, and
- Merge sort which is faster but requires more memory

We will now look at a recursive algorithm which may be done *almost* in place but which is faster than heap sort

- Use an object in the array (a pivot) to divide the two
- Average case: $\Theta(n \ln(n))$ time and $\Theta(\ln(n))$ memory
- Worst case: $\Theta(n^2)$ time and $\Theta(n)$ memory

We will look at strategies for avoiding the worst case

Quicksort

Merge sort splits the array into two sub-lists and sorts them

– It splits the larger problem into two sub-problems based on *location* in the array

Consider the following alternative:

– Chose an object in the array and partition the remaining objects into two groups relative to the chosen entry

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 recursively applying the algorithm to the first six and last eight entries

Run-time analysis

Like merge sort, we can either:

- Sort the sub-lists using quicksort
- If the size of the sub-list is sufficiently small, apply insertion sort

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: $\Theta(n \ln(n))$

What happens if we don't get that lucky?

Worst-case scenario

Suppose we choose the middle element as our pivot and we try ordering a sorted list:

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

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 $\mathrm{T}(n) = \mathrm{T}(n-1) + \boldsymbol{\Theta}(n) = \boldsymbol{\Theta}(n^2)$

– Thus, the run time drops from $n \ln(n)$ 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

Using the median element 66, we can get two equal-size sub-lists

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

Unfortunately, median is difficult to find

Median-of-three

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 results in two sub-lists, each of which must be sorted (again, using quicksort)

Select the 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

Median-of-three

If we choose a random pivot, this will, on average, divide a set of n items into two sets of size $1/4\ n$ and $3/4\ n$, *why?*

1, 100 / 2, 98 相同，非对称

Choosing the median-of-three, this will, on average, divide the n items into two sets of size $5/16\ n$ and $11/16\ n$

- Median-of-three helps speed the algorithm
- This requires order statistics:

$$2\int_0^{\frac{1}{2}} x \cdot \left(6x\left(1-x\right)\right)dx = \frac{5}{16} = 0.3125$$

Median-of-three

Recall that merge sort always divides a list into two equal halves:

– The median-of-three will require $\dfrac{\ln\left(\frac{1}{2}\right)}{\ln\left(\frac{11}{16}\right)} \approx 1.8499$ or 85 % more recursive steps, how to get this?

– A single random pivot will require $\dfrac{\ln\left(\frac{1}{2}\right)}{\ln\left(\frac{3}{4}\right)} \approx 2.4094$ or 141 % more recursive steps

Median-of-three

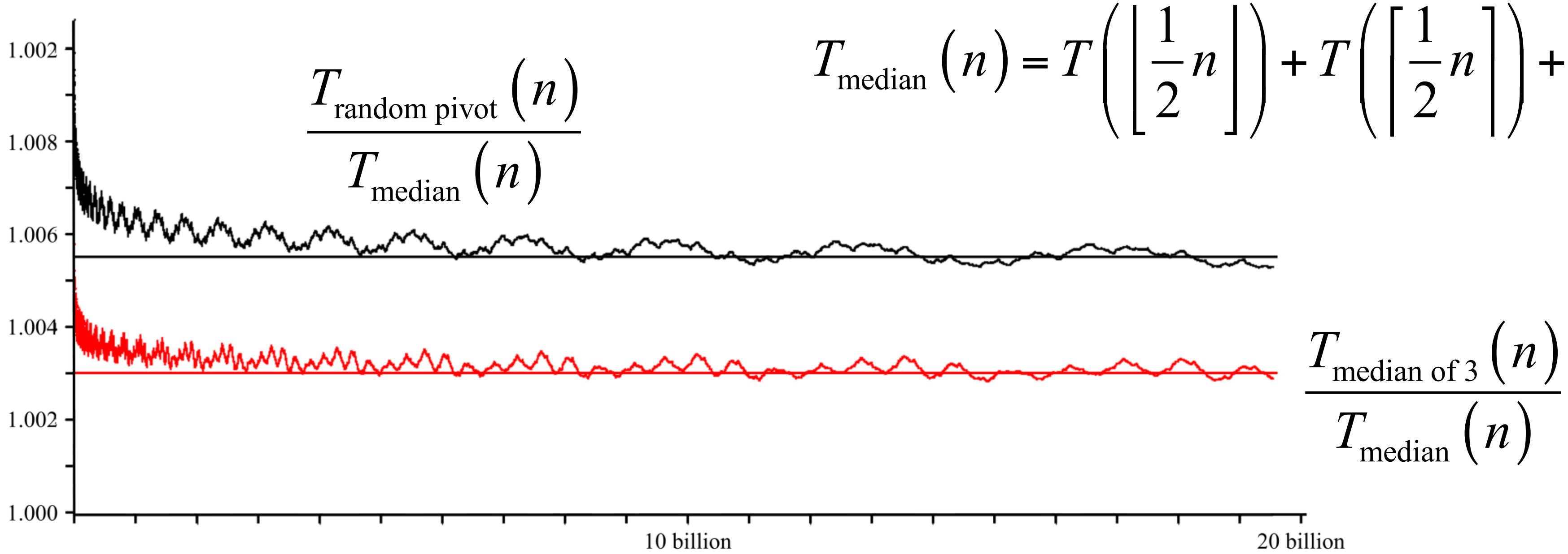
Question: what is the affect on run time?

- Surprisingly, not so much
- Here we see the ratios of the recurrence relations for large values of n

$$T_{\text{random pivot}}(n) = T\left(\left\lfloor \frac{1}{4}n \right\rfloor\right) + T\left(\left\lceil \frac{3}{4}n \right\rceil\right) + n$$

$$T_{\text{median of 3}}(n) = T\left(\left\lfloor \frac{5}{16}n \right\rfloor\right) + T\left(\left\lceil \frac{11}{16}n \right\rceil\right) + n$$

$$T_{\text{median}}(n) = T\left(\left\lfloor \frac{1}{2}n \right\rfloor\right) + T\left(\left\lceil \frac{1}{2}n \right\rceil\right) + n$$



Implementation

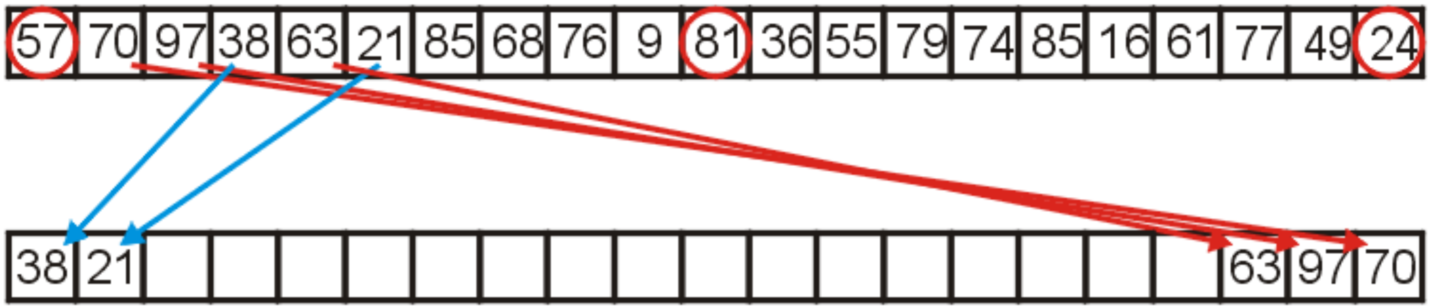
If we choose to allocate memory for an additional array, we can implement the partitioning by

- copying elements either to the front or the back of the additional array
- placing the pivot into the resulting hole

Implementation

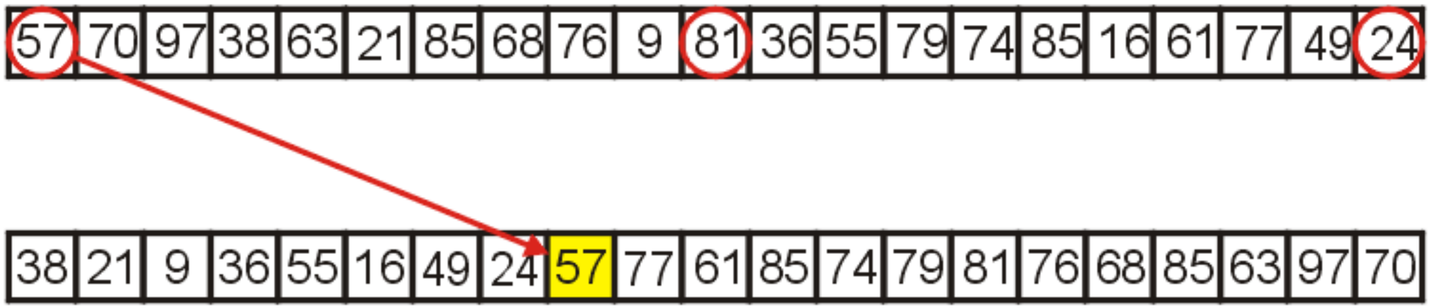
For example, consider the following:

- 57 is the median-of-three
- we go through the remaining elements, assigning them either to the front or the back of the second array



Implementation

Once we are finished, we copy the median-of-three, 57, into the resulting hole



Implementation

Can we implement quicksort in place?

Yes!

- Swap the pivot to the last slot of the list
- We repeatedly try to find two entries:
  - Staring from the front: an entry larger than the pivot
  - Starting from the back: an entry smaller than the pivot
- Such two entries are out of order, so we swap them
- Repeat until all the entries are in order
- Move the leftmost entry larger than the pivot into the last slot of the list and fill the hole with the pivot

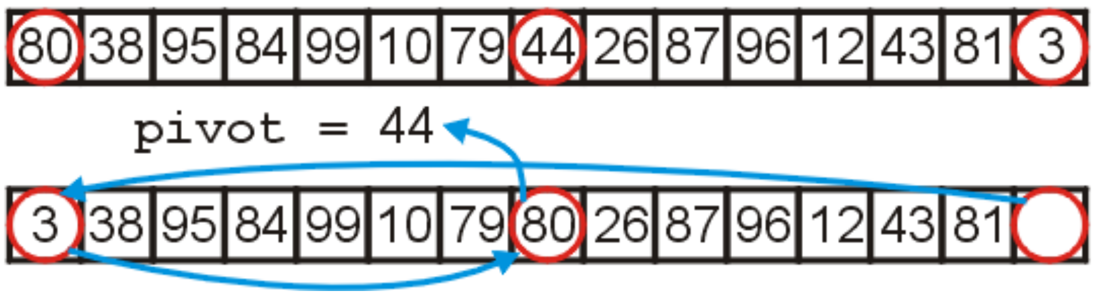
Implementation

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

The implementation is straight-forward

```
template <typename Type>
void quicksort( Type *array, int first, int last ) {
    if ( last - first <= N ) {
        insertion_sort( array, first, last );
    } else {
        Type pivot = find_pivot( array, first, last );
        int low  =     find_next( pivot, array, first + 1 );
        int high = find_previous( pivot, array,  last - 2 );

        while ( low < high ) {
            std::swap( array[low], array[high] );
            low  =     find_next( pivot, array,  low + 1 );
            high = find_previous( pivot, array, high - 1 );
        }

        array[last – 1] = array[low];
        array[low] = pivot;
        quicksort( array, first,  low );
        quicksort( array,  high, last );
    }
}
```

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

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

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**

First, $25 - 0 > 6$, so 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	

First, $25 - 0 > 6$, so 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

First, $14 - 0 > 6$, so 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

First, $14 - 0 > 6$, so 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

First, $14 - 0 > 6$, so 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, $4 - 0 \leq 6$, so find we call insertion sort

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

Quicksort example

Insertion sort just sorts the entries from `0` to `3`

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

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

Quicksort example

Insertion sort just sorts the entries from `0` to `3`

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

– This function call completes and so we exit

```
insertion_sort( array, 0, 4 )
quicksort( array,  0,  4 )
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	48	55	23	49	51	28	37	32	35	57	62	94	89	95	73	61	77	88	97	99

```
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

First, $14 - 5 > 6$, so find the midpoint and the pivot

```
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

First, $14 - 5 > 6$, so find the midpoint and the pivot

```
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

First, $14 - 5 > 6$, so find the midpoint and the pivot

```
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, $10 - 5 \le 6$, so find we call insertion sort

```
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, 11, 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, $15 - 11 \leq 6$, so find we call insertion sort

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

```
insertion_sort( array, 11, 14 )
quicksort( array, 11, 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

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

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

Quicksort example

We are back to executing `quicksort( array, 15, 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

First, $25 - 15 > 6$, so find the midpoint and the pivot

```
midpoint = (15 + 25)/2; // == 20
```

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

Quicksort example

We are back to executing `quicksort( array, 15, 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**	94	89	95	73	**99**	77	88	97	

First, $25 - 15 > 6$, so find the midpoint and the pivot

```
midpoint = (15 + 25)/2; // == 20
pivot = 62;
```

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

Quicksort example

We are back to executing `quicksort( array, 15, 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	94	89	95	73	99	77	88	97	

Searching forward and backward:

```
low = 16;
high = 15;
```

Now, `low > high`, so we stop

```
pivot = 62;
```

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

Quicksort example

We are back to executing `quicksort( array, 15, 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**	89	95	73	99	77	88	97	94

Searching forward and backward:

```
low = 16;
high = 15;
```

Now, `low > high`, so we stop

- Note, this is the worst-case scenario
- The pivot is the second smallest element

`pivot = 62;`

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

Quicksort example

We are back to executing `quicksort( array, 15, 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	89	95	73	99	77	88	97	94

We continue calling quicksort recursively on the first half

```
quicksort( array, 15, 16 );
```

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are executing `quicksort( array, 15, 16 )`

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	89	95	73	99	77	88	97	94

Now, $16 - 15 \leq 6$, so find we call insertion sort

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort immediately returns

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	89	95	73	99	77	88	97	94

```
insertion_sort( array, 15, 16 )
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
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	61	62	89	95	73	99	77	88	97	94

```
quicksort( array, 15, 16 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 15, 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**	89	95	73	99	77	88	97	94

We continue calling quicksort recursively on the second half

```
quicksort( array, 15, 16 );
quicksort( array, 17, 25 );
```

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

Quicksort example

We are now calling `quicksort( array, 17, 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	89	95	73	99	77	88	97	94

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 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	**89**	95	73	99	**77**	88	97	**94**

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 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	**89**	95	73	99	**77**	88	97	**94**

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
pivot = 89
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 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	**77**	95	73	99	**94**	88	97	

First, $25 - 17 > 6$, so find the midpoint and the pivot

```
midpoint = (17 + 25)/2; // == 21
pivot = 89
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 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	77	95	73	99	94	88	97	

Searching forward and backward:

```
low = 18;
high = 22;
```

```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 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	77	88	73	99	94	95	97	

Searching forward and backward:

```
low = 18;
high = 22;
```

Swap them

```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 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	77	88	73	99	94	95	97	

Searching forward and backward:

```
low = 20;
high = 19;
```

Now, `low > high`, so we stop

```
pivot = 89;
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 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	77	88	73	**89**	94	95	97	99

Searching forward and backward:

```
low = 20;
high = 19;
```

Now, `low > high`, so we stop

`pivot = 89;`

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 17, 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	77	88	73	**89**	94	95	97	99

We start by calling quicksort recursively on the first half

```
quicksort( array, 17, 20 );
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now executing `quicksort( array, 17, 20 )`

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	77	88	73	89	94	95	97	99

Now, $4 - 0 \leq 6$, so find we call insertion sort

```
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 17 to 19

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	77	88	73	89	94	95	97	99

```
insertion_sort( array, 17, 20 )
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 17 to 19

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

– This function call completes and so we exit

```
insertion_sort( array, 17, 20 )
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
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	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 17, 20 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 17, 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

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are back to executing `quicksort( array, 17, 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 continue by calling quicksort on the second half

```
quicksort( array, 17, 20 );
quicksort( array, 21, 25 );
```

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

We are now calling `quicksort( array, 21, 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

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

```
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 21 to 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	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

```
insertion_sort( array, 21, 25 )
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
quicksort( array,  0, 25 )
```

Quicksort example

Insertion sort just sorts the entries from 21 to 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	7	13	15	17	23	28	32	35	37	48	49	51	55	57	61	62	73	77	88	89	94	95	97	99

- In this case, the sub-array was already sorted
- This function call completes and so we exit

```
insertion_sort( array, 21, 25 )
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
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	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 21, 25 )
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
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	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 17, 25 )
quicksort( array, 15, 25 )
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	61	62	73	77	88	89	94	95	97	99

```
quicksort( array, 15, 25 )
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	61	62	73	77	88	89	94	95	97	99

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

Quicksort example

We have now used quicksort to sort this 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
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

Black Board Example

Sort the following list using quicksort

– Use insertion sort for any sub-list of size 4 or less

0	1	2	3	4	5	6	7	8	9	10
34	15	65	59	68	42	40	80	50	65	23

Memory Requirements

The additional memory?

- Function call stack
  - Each recursive function call places its local variables, parameters, *etc*., on a stack
- Average case: the depth of the recursion is $\Theta(\ln(n))$
- Worst case: the depth of the recursion is $\Theta(n)$

Run-time Summary

To summarize all three $\Theta(n \ln(n))$ algorithms

	Average Run Time	Worst-case Run Time	Average Memory	Worst-case Memory
Heap Sort	$\mathrm{O}(n \ln(n))$		$\Theta(1)$	
Merge Sort	$\Theta(n \ln(n))$		$\Theta(n)$	
Quicksort	$\Theta(n \ln(n))$	$\Theta(n^2)$	$\Theta(\ln(n))$	$\Theta(n)$

Further modifications

Our implementation is by no means optimal:

An excellent paper on quicksort was written by Jon L. Bentley and M. Douglas McIlroy:

Engineering a Sort Function

found in Software—Practice and Experience, Vol. 23(11), Nov 1993

Summary

This topic covered quicksort

- On average faster than heap sort or merge sort
- Uses a pivot to partition the objects
- Using the median of three pivots is a reasonably means of finding the pivot
- Average run time of $\Theta(n \ln(n))$ and $\Theta(\ln(n))$ memory
- Worst case run time of $\Theta(n^2)$ and $\Theta(n)$ memory