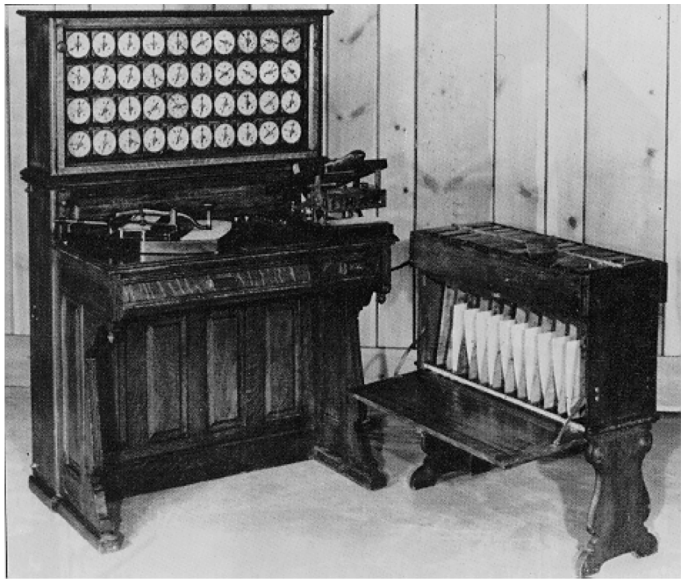
SAMUEL GINN
COLLEGE OF ENGINEERING

Sorting

History

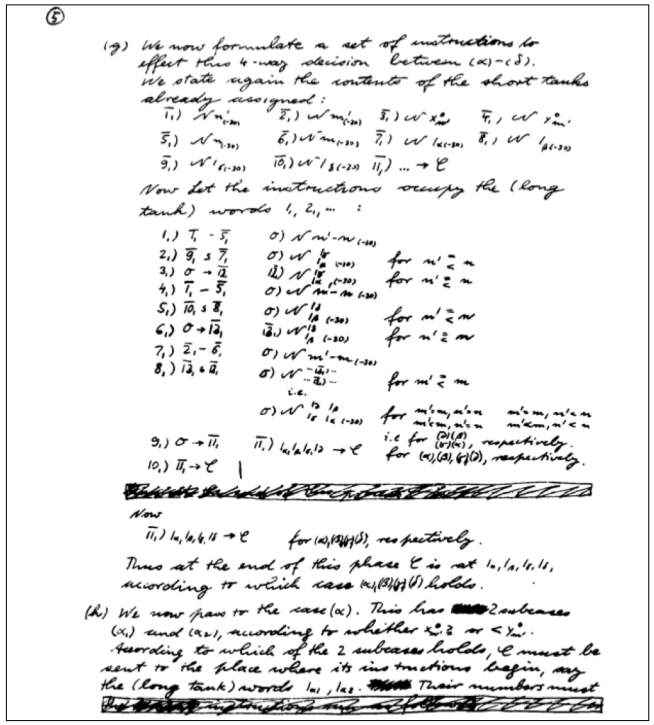


Herman Hollerith's tabulating machine and "sorting box"

1880 Census: 8 years to process
1890 Census: 1 year to process

"This apparatus works unerringly as the mills of the gods, but beats them hollow as to speed." The Electrical Engineer, 11 Nov 1891.

History



(g) We now formulate a set of instructions to effect this 4-way decision between (α)-(δ). We state again the contents of the short tanks already assigned:

$\overline{1}_1$) $W^{n'}_{-m}$ $\overline{2}_1$) $W^{m}_{(-2n)}$ $\overline{3}_1$) $W^{x^0_m}$ $\overline{4}_1$) $W^{y^0_m}$.

$\overline{5}_1$) $W^{\quad}_{(-2n)}$ $\overline{6}_1$) $W^{m}_{(-2n)}$ $\overline{7}_1$) $W^{\quad}_{(-2n)}$ $\overline{8}_1$) $W^{\quad}_{(-2n)}$

$\overline{9}_1$) $W^{\quad}_{(-2n)}$ $\overline{10}_1$) $W^{\quad}_{(-2n)}$ $\overline{11}_1$) ... → $\mathcal{C}$

Now let the instructions occupy the (long tank) words $1_1, 2_1, \ldots$:

1_1) $\overline{1}_1 - \overline{5}_1$ σ) $W^{n'-m}_{\quad(-2n)}$

2_1) $\overline{9}_1 \leq \overline{7}_1$ σ) $W^{\quad}_{\quad(-2n)}$ for $n' \geq n$

3_1) $\sigma \to \overline{12}_1$ $\overline{12}_1$) $W^{\quad}_{\quad(-2n)}$ for $n' \geq n$

4_1) $\overline{7}_1 - \overline{5}_1$ σ) $W^{\quad}_{\quad(-2n)}$

5_1) $\overline{10}_1 \leq \overline{8}_1$ σ) $W^{\quad}_{\quad(-2n)}$ for $n' \geq m$

6_1) $\sigma \to \overline{13}_1$ $\overline{13}_1$) $W^{\quad}_{\quad(-2n)}$ for $n' \geq m$

7_1) $\overline{2}_1 - \overline{6}_1$ σ) $W^{m'-m}_{\quad(-2n)}$

8_1) $\overline{13}_1 \leq \overline{11}_1$ σ) $W^{\quad}_{\quad}$ for $m' \geq m$

i.e.

σ) $W^{\quad}_{\quad(-2n)}$ for $m' \geq m, m' \geq n$ $m' < m, m' \geq n$ $m' \geq m, m' < n$ $m' < m, m' < n$

9_1) $\sigma \to \overline{11}_1$ $\overline{11}_1$) $l_\alpha l_\beta l_\gamma l_\delta \to \mathcal{C}$ i.e. for $\binom{(\alpha)(\beta)}{(\gamma)(\delta)}$, respectively.

10_1) $\overline{11}_1 \to \mathcal{C}$ for $(\alpha),(\beta),(\gamma),(\delta)$, respectively.

Now

$\overline{11}_1$) $l_\alpha l_\beta l_\gamma l_{11} \to \mathcal{C}$ for $(\alpha),(\beta),(\gamma),(\delta)$, respectively.

Thus at the end of this phase $\mathcal{C}$ is set $l_\alpha, l_\beta, l_\gamma, l_{11}$, according to which case $(\alpha),(\beta),(\gamma),(\delta)$ holds.

(h) We now pass to the case (α). This has [redacted] 2 subcases (α_1) and (α_2), according to whether $x^0_m \geq 2$ or $< y^0_m$. According to which of the 2 subcases holds, $\mathcal{C}$ must be sent to the place where its instructions begin, say the (long tank) words $l_{\alpha 1}, l_{\alpha 2}$. [redacted] Their numbers must [redacted]

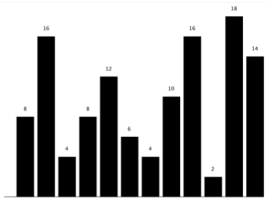
John von Neumann's first program for the EDVAC

"I have also worked on sorting questions … We wish to formulate code instructions for sorting … and to see how much … time they require. … At any rate, the moral seems to be that the EDVAC … is definitely faster than the [IBM sorters]. It is legitimate to conclude … that the EDVAC is very nearly an "all purpose" machine." JvN, 1945

Sorting

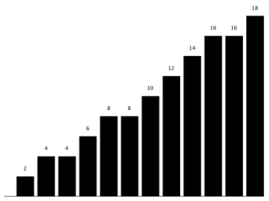
Input: List a of N elements (a_1, a_2, …, a_N)

8	16	4	8	12	6	4	10	16	2	18	14
0	1	2	3	4	5	6	7	8	9	10	11



Output: A permutation of the elements in a such that $a_i \le a_{i+1}$ for $1 \le i \le N - 1$

2	4	4	6	8	8	10	12	14	16	16	18
0	1	2	3	4	5	6	7	8	9	10	11



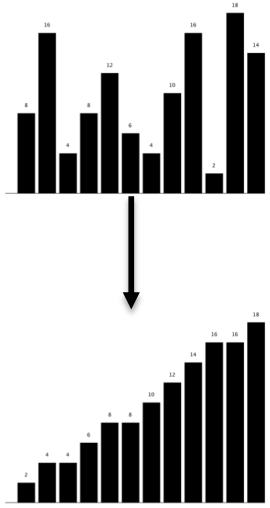
Lots of algorithms … selection sort, insertion sort, bubblesort, shaker sort, quicksort, merge sort, heapsort, samplesort, shellsort, solitaire sort, red-black sort, splaysort, psort, radix sort, counting sort, bucket sort, distribution sort, timsort, comb sort, …

Inversions and exchanges

An **inversion** is a pair of elements that are out of order.

8	16	4	8	12	6	4	10	16	2	18	14
0	1	2	3	4	5	6	7	8	9	10	11

20 inversions: (8,4), (8,6), (8,4), (8,2), (16, 12), (4,2), (8,6), (8,4), (8,2), (12,6), (12,4), (12,10), (12,2), (6,4), (6,2), (4,2), (10,2), (16,2), (16,14), (18,14)



2	4	4	6	8	8	10	12	14	16	16	18
0	1	2	3	4	5	6	7	8	9	10	11

0 inversions

exchange = swap the locations of inverted elements

Sorting could be seen as a sequence of exchanges.

Inversions and exchanges

Randomly ordered data:

4	10	6	8	12	2
0	1	2	3	4	5

7 inversions

12	6	4	8	2	10
0	1	2	3	4	5

9 inversions

8	10	12	2	4	6
0	1	2	3	4	5

9 inversions

A list in reverse order (non increasing) would have the maximum number of inversions.

list	# inversions	
4, 3, 2, 1	3 + 2 + 1 + 0	6
5, 4, 3, 2, 1	4 + 3 + 2 + 1 + 0	10
6, 5, 4, 3, 2, 1	5 + 4 + 3 + 2 + 1 + 0	15
. . .		

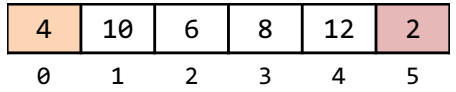
$$\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2} = \frac{N^2-N}{2}$$

In the worst case, there will be $O(N^2)$ inversions in the data.

Inversions and exchanges

Detecting an inversion:

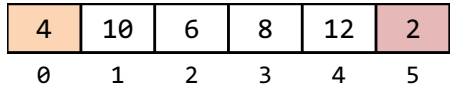
```
private boolean less(Comparable x, Comparable y) {
   return x.compareTo(y) < 0;
}
```



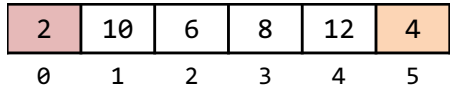
less(a[5], a[0]) -> true

Correcting an inversion:

```
private void swap(Comparable[] a, int i, int j) {
   T temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```



swap(a, 5, 0)



Properties of sorts

Comparison sort: The only assumption about the data being sorted is that the data elements can be compared to each other on the basis of "less than" - that is, a total order exists.

In-place: The list itself is rearranged and only a constant amount of extra space is required.

Adaptive: Running time can be improved by taking advantage of certain input arrangements.

Stable: Equal elements maintain the same relative order.

all four are comparison sorts

(as presented)	Selection	Insertion	Mergesort	Quicksort
In-place?	Yes	Yes	No	Yes
Stable?	No	Yes	Yes	No
Adaptive?	No	Yes	No	No

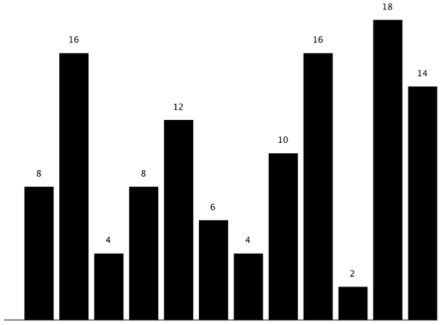
Selection Sort

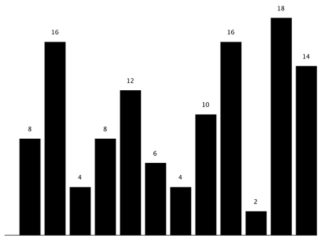
Selection sort

Walk from left to right through the array.

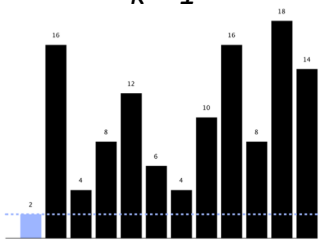
On each step, **select** the element that goes in the current location in sorted order and put it there.

After k steps, the first k elements are in sorted order and are in their final positions.

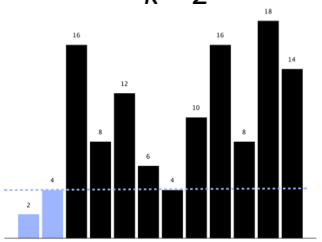




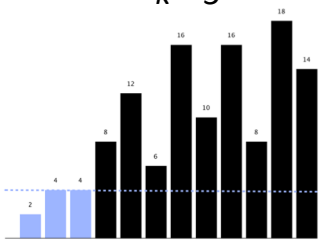
k = 1



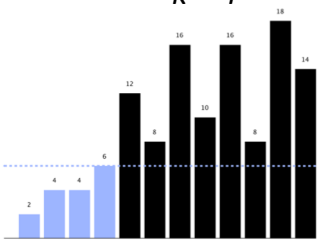
k = 2



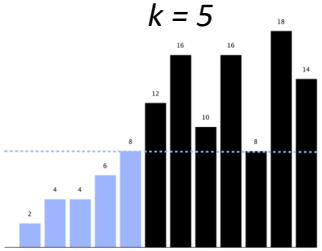
k = 3



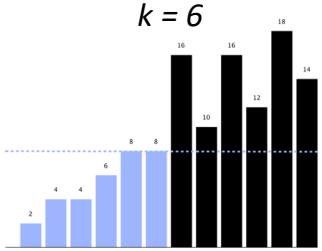
k = 4



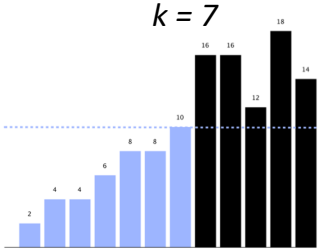
k = 5



k = 6

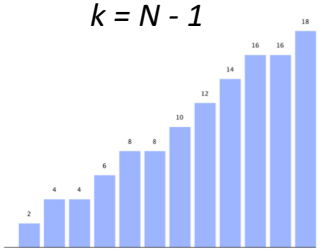


k = 7



. . .

k = N - 1



Selection sort

```
public void selectionSort(T[] a) {
   for (int i = 0; i < a.length; i++) {
      int min = i;
      for (int j = i + 1; j < a.length; j++) {
         if (less(a[j], a[min])) {
            min = j;
         }
      }
      swap(a, i, min);
   }
}
```

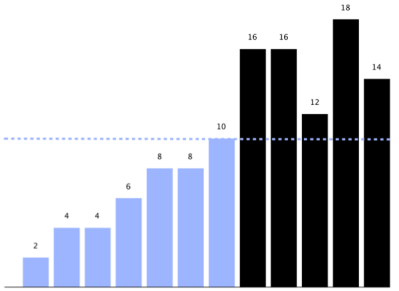
For each index in the array …

select the minimum value from the current index to the right …

and swap it with the value at the current index.

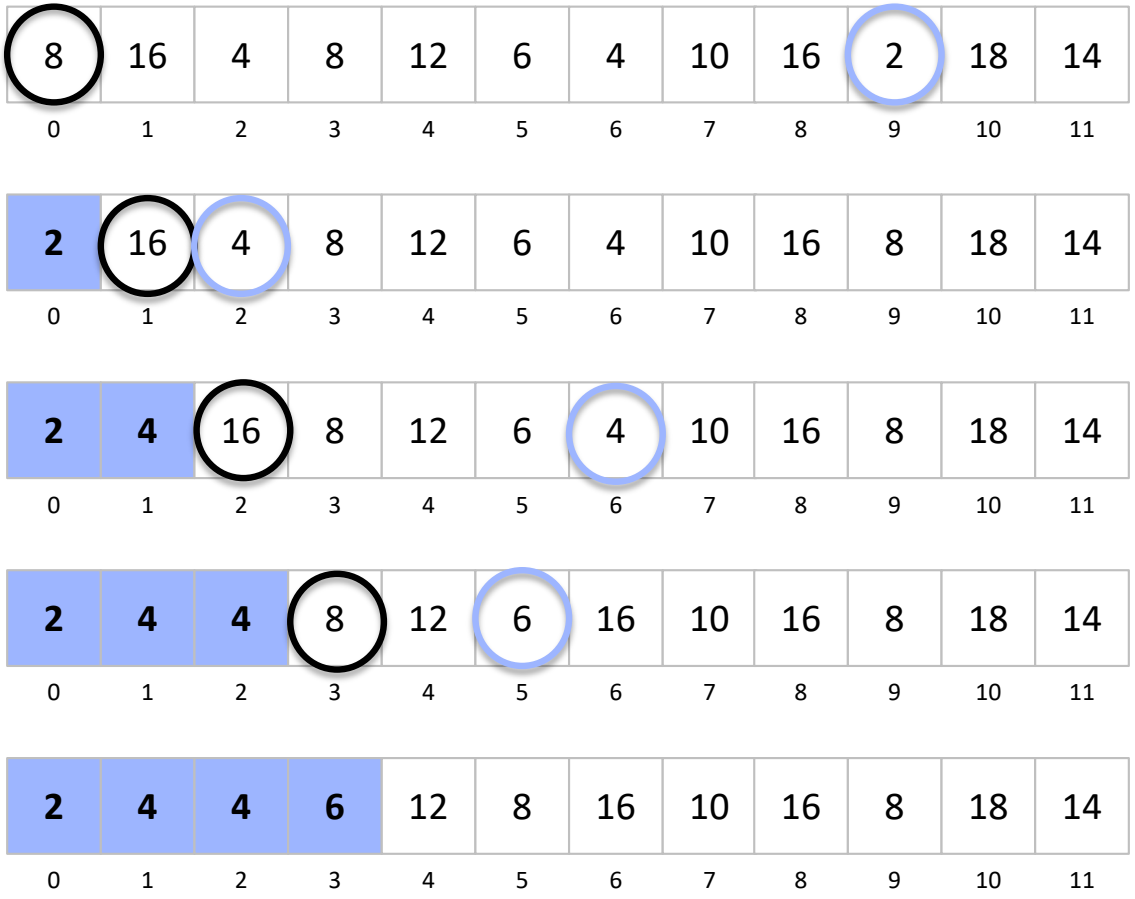
After any given iteration of the outer loop, the array is divided into two parts:

- values at indices 0 through i are in sorted order and are in their final locations
- values at indices i + 1 through a.length - 1 are unsorted



Selection sort

```
public void selectionSort(T[] a) {
   for (int i = 0; i < a.length; i++) {
      int min = i;
      for (int j = i + 1; j < a.length; j++) {
         if (less(a[j], a[min])) {
            min = j;
         }
      }
      swap(a, i, min);
   }
}
```



Invariants

After any iteration of the outer loop:

- The first i elements are in their correct sorted position.
- Elements i+1 through the last element are all greater than or equal to the first i elements.

Selection sort

```
public void selectionSort(T[] a) {
   for (int i = 0; i < a.length; i++) {
      int min = i;
      for (int j = i + 1; j < a.length; j++) {
         if (less(a[j], a[min])) {
            min = j;
         }
      }
      swap(a, i, min);
   }
}
```

Time complexity

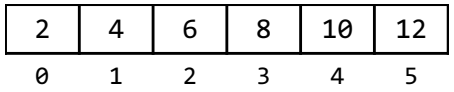
Selection sort is $O(N^2)$.

$(N^2-N)/2$ comparisons (calls to `less`)

N-1 exchanges (calls to `swap`)

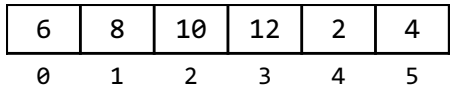
Selection sort is **not *adaptive*** to its input, so all arrangements of data in the array will require a quadratic amount of work.

$O(N^2)$
already sorted



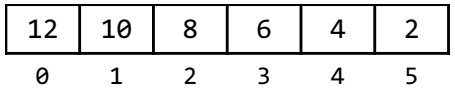
~N^2 comparisons
~N exchanges

$O(N^2)$
"almost" sorted



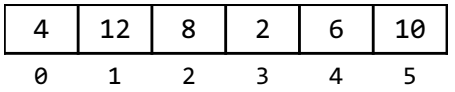
~N^2 comparisons
~N exchanges

$O(N^2)$
in reverse order



~N^2 comparisons
~N exchanges

$O(N^2)$
in random order



~N^2 comparisons
~N exchanges

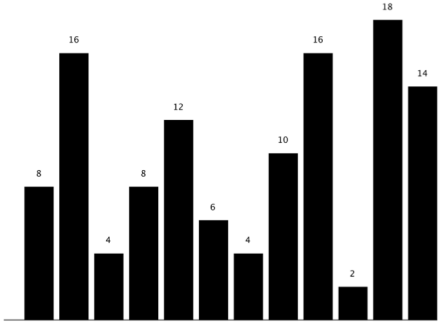
Insertion Sort

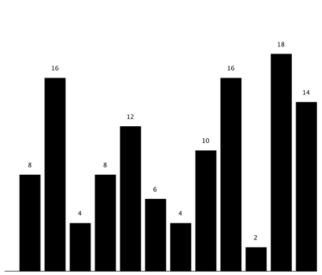
Insertion sort

Walk from left to right through the array.

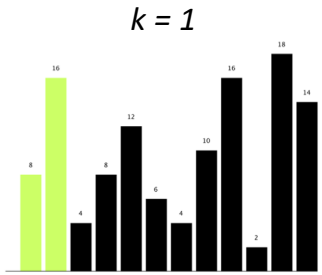
On each step, **insert** the element in the current location in sorted order to its left.

After k steps, the first k + 1 elements are in sorted order relative to each other.

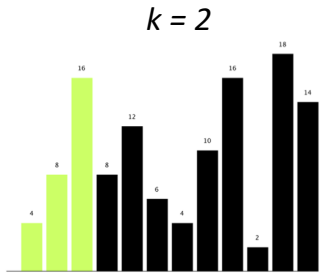




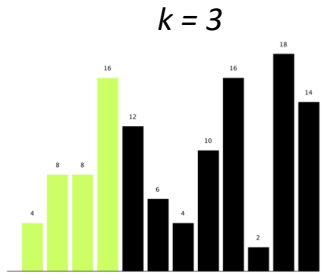
k = 1



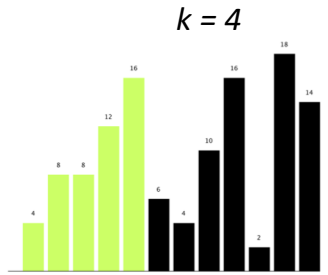
k = 2



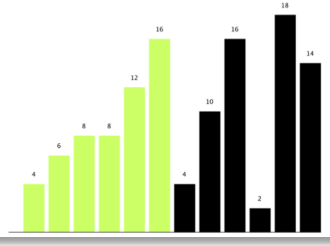
k = 3



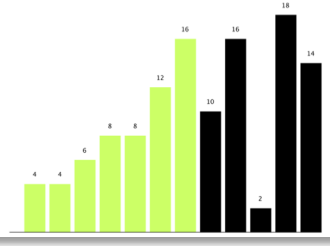
k = 4



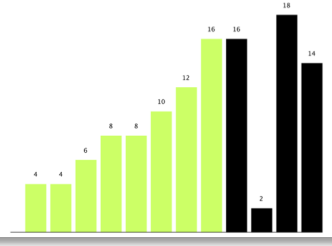
k = 5



k = 6

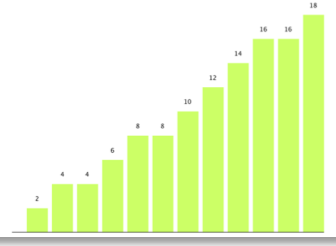


k = 7



. . .

k = N - 1



Insertion sort

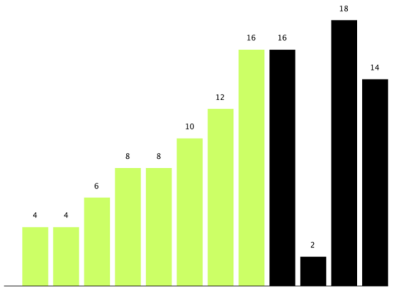
```
public void insertionSort(T[] a) {
   for (int i = 0; i < a.length; i++) {
      for (int j = i; j > 0; j--) {
         if (less(a[j], a[j - 1])) {
            swap(a, j, j - 1);
         } else {
            break;
         }
      }
   }
}
```

For each value in the array …

insert it in sorted order relative to the values to its left …

After any given iteration of the outer loop, the array is divided into two parts:

- values at indices 0 through i are in sorted order, but only with respect to each other
- values at indices i + 1 through a.length - 1 are unknown



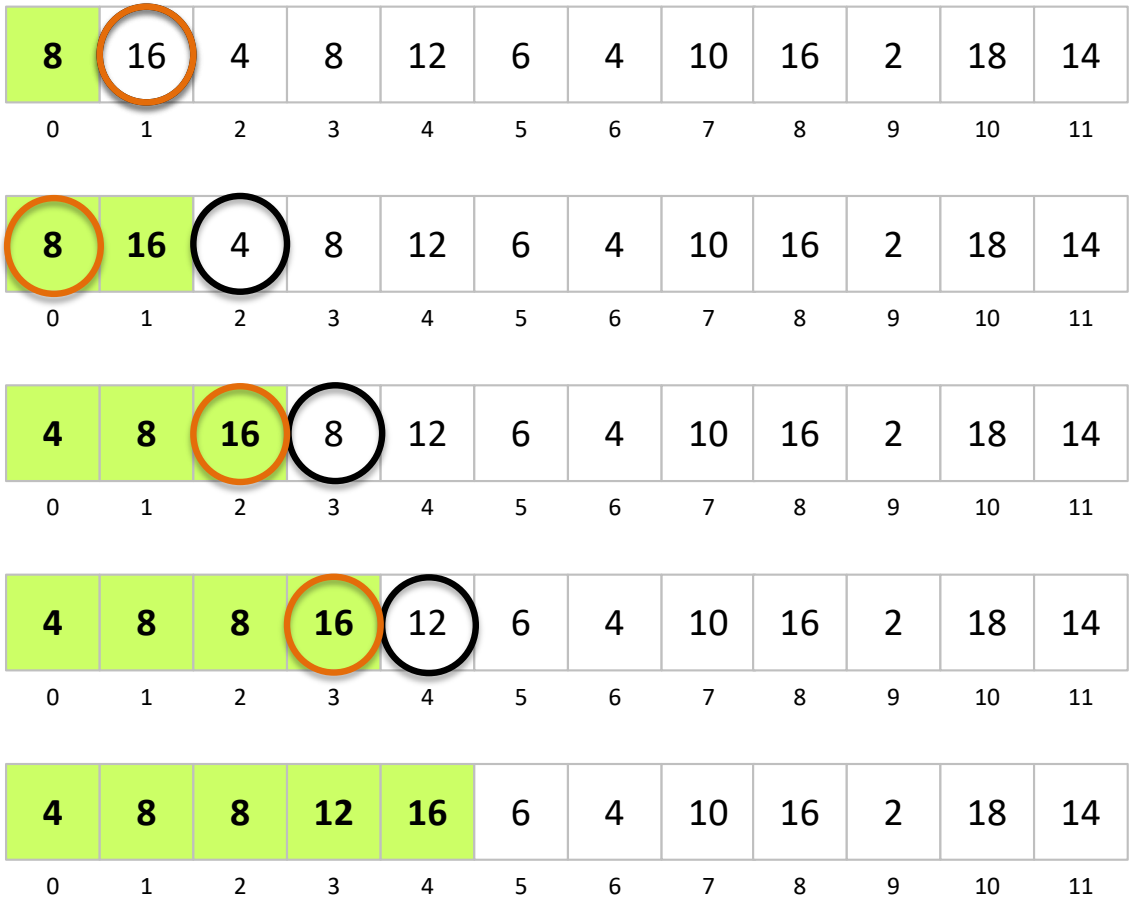
Insertion sort

```
public void insertionSort(T[] a) {
   for (int i = 0; i < a.length; i++) {
      for (int j = i; j > 0; j--) {
         if (less(a[j], a[j - 1])) {
            swap(a, j, j - 1);
         } else {
            break;
         }
      }
   }
}
```

Invariants:

After any iteration of the outer loop:

- The first i elements are in sorted order relative to each other.
- Elements i + 1 through the last element are in their original positions and have not been accessed.



Insertion sort

```
public void insertionSort(T[] a) {
   for (int i = 0; i < a.length; i++) {
      for (int j = i; j > 0; j--) {
         if (less(a[j], a[j - 1])) {
            swap(a, j, j - 1);
         } else {
            break;
         }
      }
   }
}
```

Time complexity

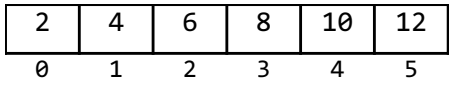
Insertion sort is $O(N^2)$.

$\leq (N^2-N)/2$ comparisons (calls to `less`)

$\leq (N^2-N)/2$ exchanges (calls to `swap`)

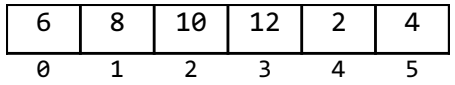
Insertion sort **is *adaptive***, so input that is sorted or almost sorted will only require a linear amount of time.

$O(N)$
already sorted



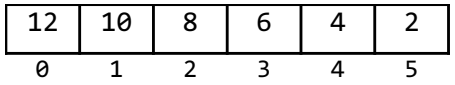
~N comparisons
0 exchanges

$O(N)$
"almost" sorted



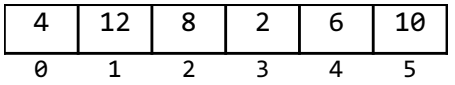
~N comparisons
~N exchanges

$O(N^2)$
in reverse order



~N^2 comparisons
~N^2 exchanges

$O(N^2)$
in random order



~N^2 comparisons
~N^2 exchanges

Sorting more efficiently

Sorting more efficiently

Insertion sort and selection sort both have $O(N^2)$ performance, so they don't scale very well.

	1,000	10,000	100,000	1,000,000	10,000,000
O(log N)	< 1 ns	10 ns	132 ns	166 ns	199 ns
O(N)	1 ns	10 ns	100 ns	1 ms	10 ms
O(NlogN)	10 ns	100 ns	2 ms	20 ms	0.2 sec
$O(N^2)$	**1 ms**	**0.1 sec**	**10 sec**	**17 min**	**28 hours**
$O(N^3)$	1 sec	17 min	12 days	32 yrs	32,000 yrs
$O(N^4)$	17 min	4 months	3,200 yrs	3.2 M yrs	3.17E15 yrs
$O(2^N)$	3.4E284 yrs	??	??	??	??
O(N!)	??	??	??	??	??

Sorting more efficiently

How could we sort more efficiently?

	1,000	10,000	100,000	1,000,000	10,000,000
O(log N)	< 1 ns	10 ns	132 ns	166 ns	199 ns
O(N)	1 ns	10 ns	100 ns	1 ms	10 ms
O(NlogN)	10 ns	100 ns	2 ms	20 ms	0.2 sec
$O(N^2)$	1 ms	0.1 sec	10 sec	17 min	28 hours
$O(N^3)$	1 sec	17 min	12 days	32 yrs	32,000 yrs
$O(N^4)$	17 min	4 months	3,200 yrs	3.2 M yrs	3.17E15 yrs
$O(2^N)$	3.4E284 yrs	??	??	??	??
O(N!)	??	??	??	??	??

← Counting sort, radix sort, etc. O(N)

Impose additional constraints on the problem.

Example: The values being sorted must be integers in a given range. => Counting sort O(N)

Sorting more efficiently

How could we sort more efficiently?

	1,000	10,000	100,000	1,000,000	10,000,000
O(log N)	< 1 ns	10 ns	132 ns	166 ns	199 ns
O(N)	1 ns	10 ns	100 ns	1 ms	10 ms
O(NlogN)	**10 ns**	**100 ns**	**2 ms**	**20 ms**	**0.2 sec**
$O(N^2)$	**1 ms**	**0.1 sec**	**10 sec**	**17 min**	**28 hours**
$O(N^3)$	1 sec	17 min	12 days	32 yrs	32,000 yrs
$O(N^4)$	17 min	4 months	3,200 yrs	3.2 M yrs	3.17E15 yrs
$O(2^N)$	3.4E284 yrs	??	??	??	??
O(N!)	??	??	??	??	??

Merge sort, quicksort, O(NlogN)

Use a divide-and-conquer algorithm

Example: Divide the array in half, sort each half, then combine the sorted halves.
=> Merge sort O(NlogN)

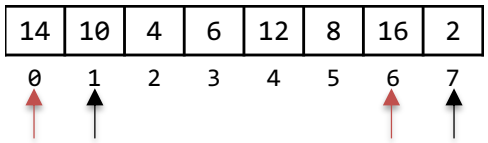
Divide and Conquer

Divide and conquer

Divide and conquer is an algorithm design technique where we **divide** the problem into two or more smaller parts, solve (**conquer**) each part, and then **combine** the solutions for the parts into a solution for the whole problem.

Example: Find the maximum element in an array.

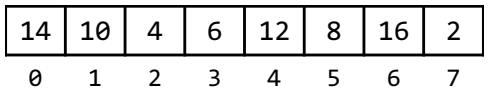
Typical strategy: linear scan



16

```
public int max(int[] a) {
   int m = a[0];
   for (int i : a) {
      if (i > m)
         m = i;
   }
   return m;
}
```

Divide and conquer strategy Just for illustration!



Divide: Partition the array into two halves.

Conquer: Find the largest value in each half.

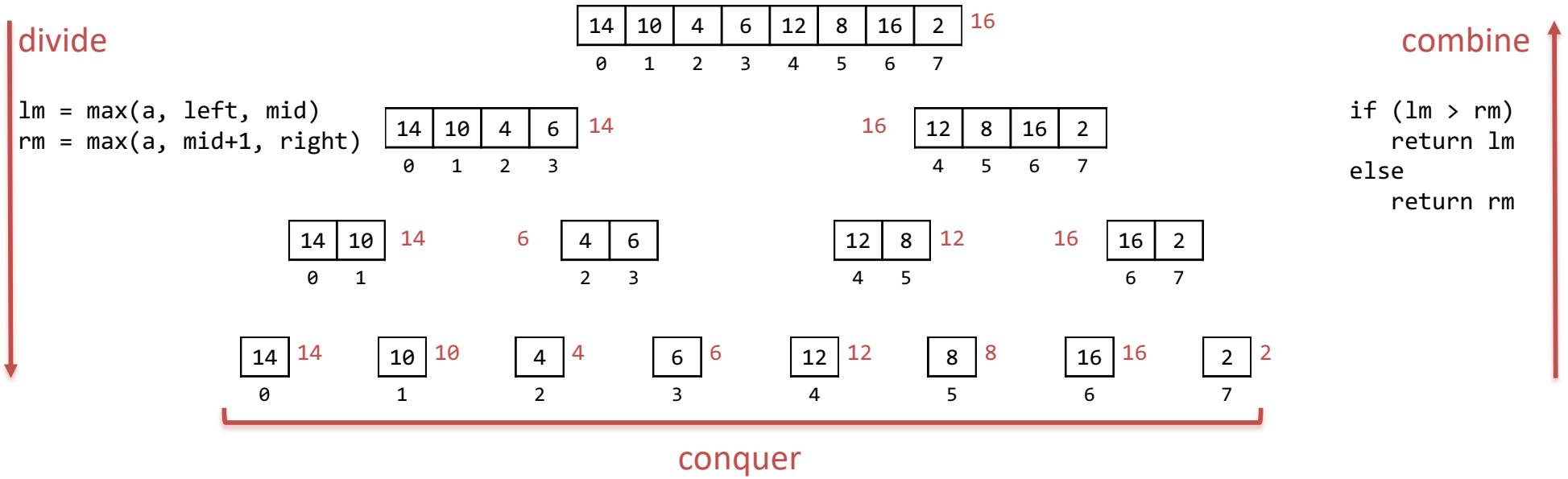
largest in left: 14 largest in right: 16

Combine: Pick the larger of these two values. **16**

Divide and conquer

Divide and conquer algorithms are usually expressed **recursively**, and the division is repeated until each part is small enough to be solved directly or trivially.

```
public int max(int[] a, int left, int right) { . . . }
```



Divide and conquer

A divide-and-conquer solution for the maximum element in an array:

Just for illustration!

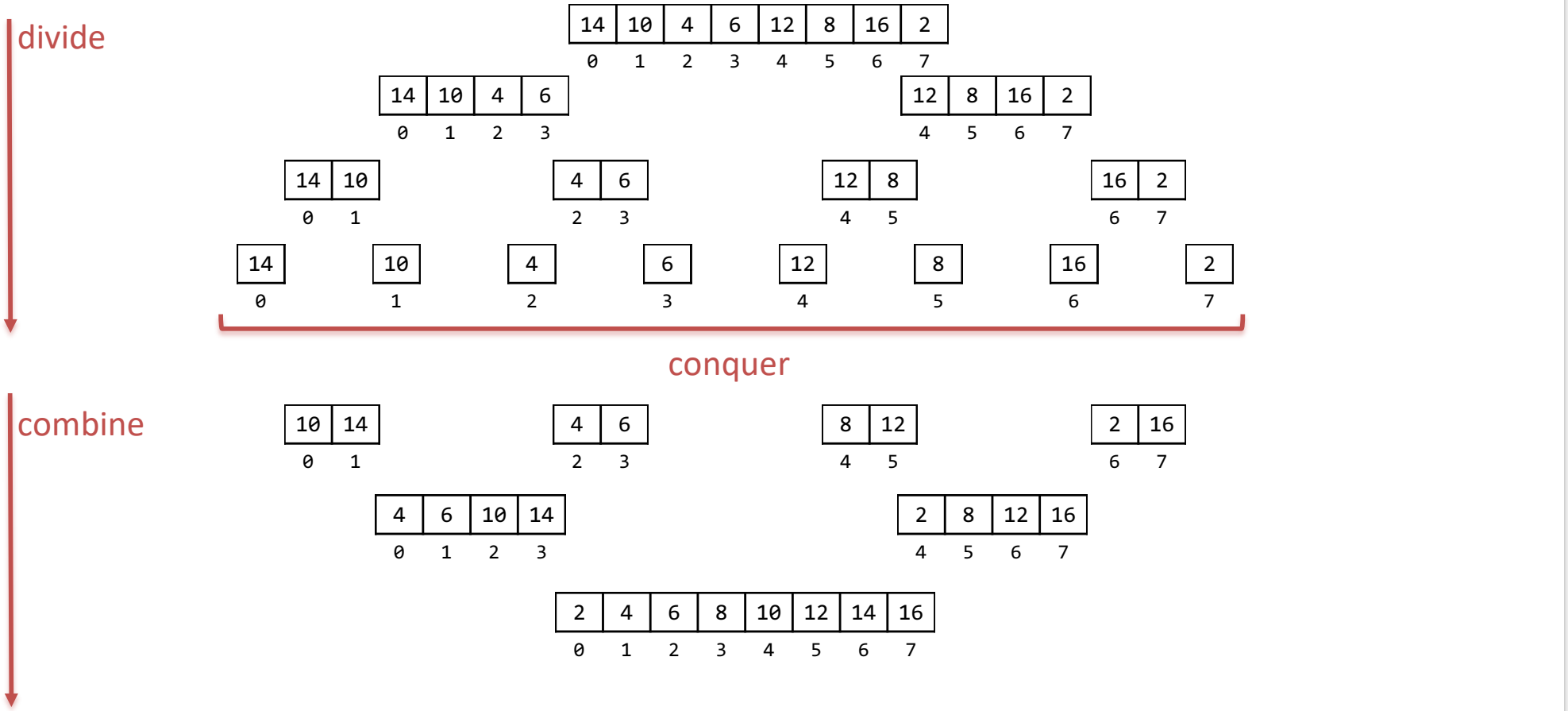
```
public static int max(int[] a, int l, int r) {
   if (l == r) {
      return a[l];
   }

   int mid = (l + r) / 2;
   int lm = max(a, l, mid);
   int rm = max(a, mid + 1, r);

   if (lm > rm )
      return lm;
   else
      return rm;
}
```

conquer

divide

combine

D&C sorting?


divide
14 10 4 6 12 8 16 2
0 1 2 3 4 5 6 7
14 10 4 6
0 1 2 3
12 8 16 2
4 5 6 7
14 10
0 1
4 6
2 3
12 8
4 5
16 2
6 7
14
0
10
1
4
2
6
3
12
4
8
5
16
6
2
7
conquer
combine
10 14
0 1
4 6
2 3
8 12
4 5
2 16
6 7
4 6 10 14
0 1 2 3
2 8 12 16
4 5 6 7
2 4 6 8 10 12 14 16
0 1 2 3 4 5 6 7

Merge Sort

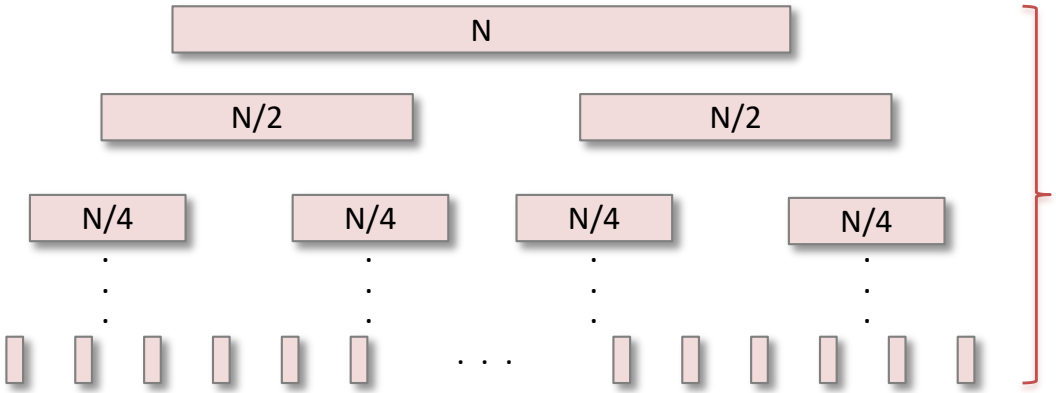
Merge sort

Merge sort is a comparison sort based on the **divide-and-conquer** strategy.

First described by John von Neumann in 1945 as part of his work on EDVAC.

Asymptotically optimal for comparison sorting – **O(NlogN)**, but requires ~N extra memory (it's **not** *in-place*).

Divide: Divide the array in half.
Conquer: Sort each half (recursively).
Combine: Merge the sorted halves into a single array.



~$\log_2$N levels

N elements to compare, move per level

O(NlogN) work

Merge sort

```
public void mergeSort(Comparable[] a, int left, int right) {

   if (right == left) return;

   int mid = left + (right - left) / 2;
   mergeSort(a, left, mid);
   mergeSort(a, mid + 1, right);

   merge(a, left, mid, right);
}
```

conquer

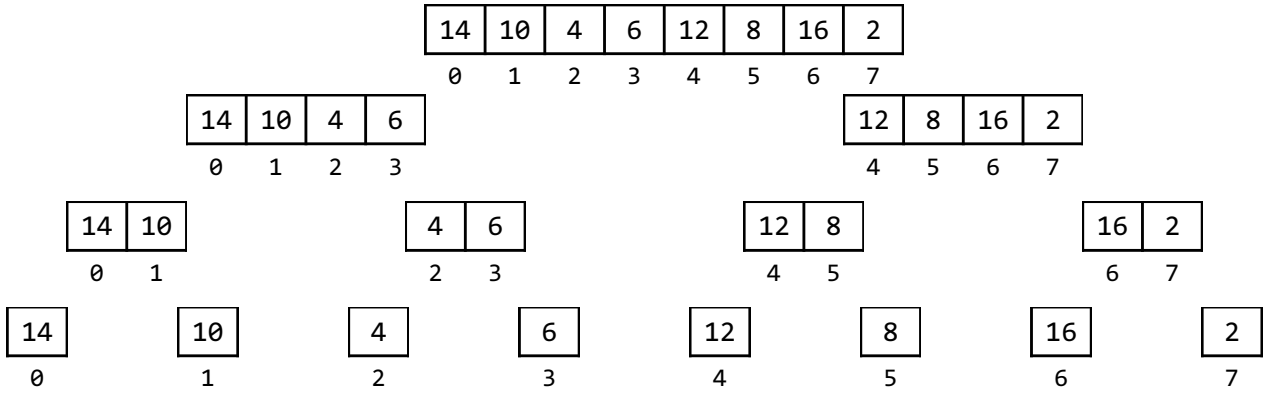
divide

combine

Merge sort

```
public static void mergeSort(Comparable[] a, int left, int right) {

   if (right == left) return;

   int mid = left + (right - left) / 2;
   mergeSort(a, left, mid);
   mergeSort(a, mid + 1, right);

   merge(a, left, mid, right);
}
```

divide



Merge sort

```
public static void mergeSort(Comparable[] a, int left, int right) {

   if (right == left) return;

   int mid = left + (right - left) / 2;
   mergeSort(a, left, mid);
   mergeSort(a, mid + 1, right);

   merge(a, left, mid, right);
}
```

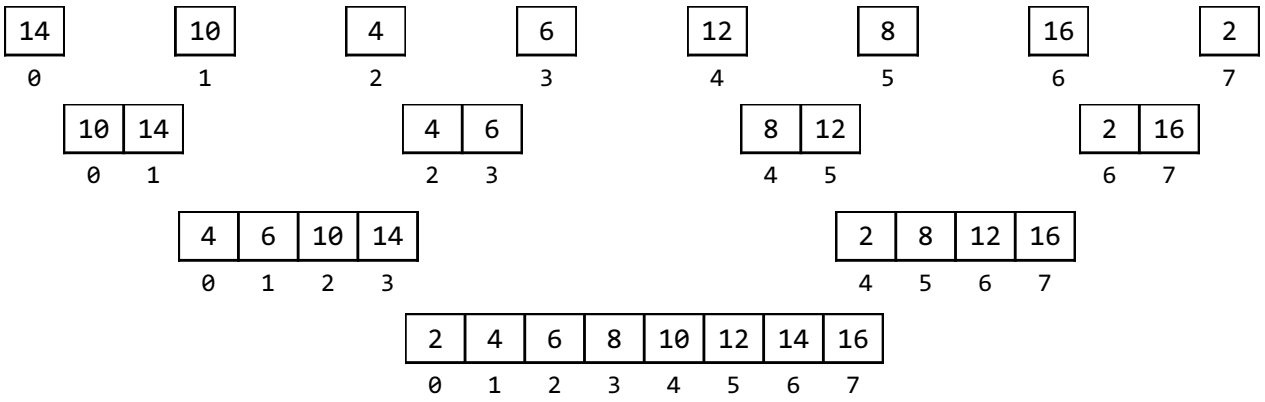
conquer

Sorting a one-element array is trivial – it's already sorted so there's nothing to do.

Merge sort

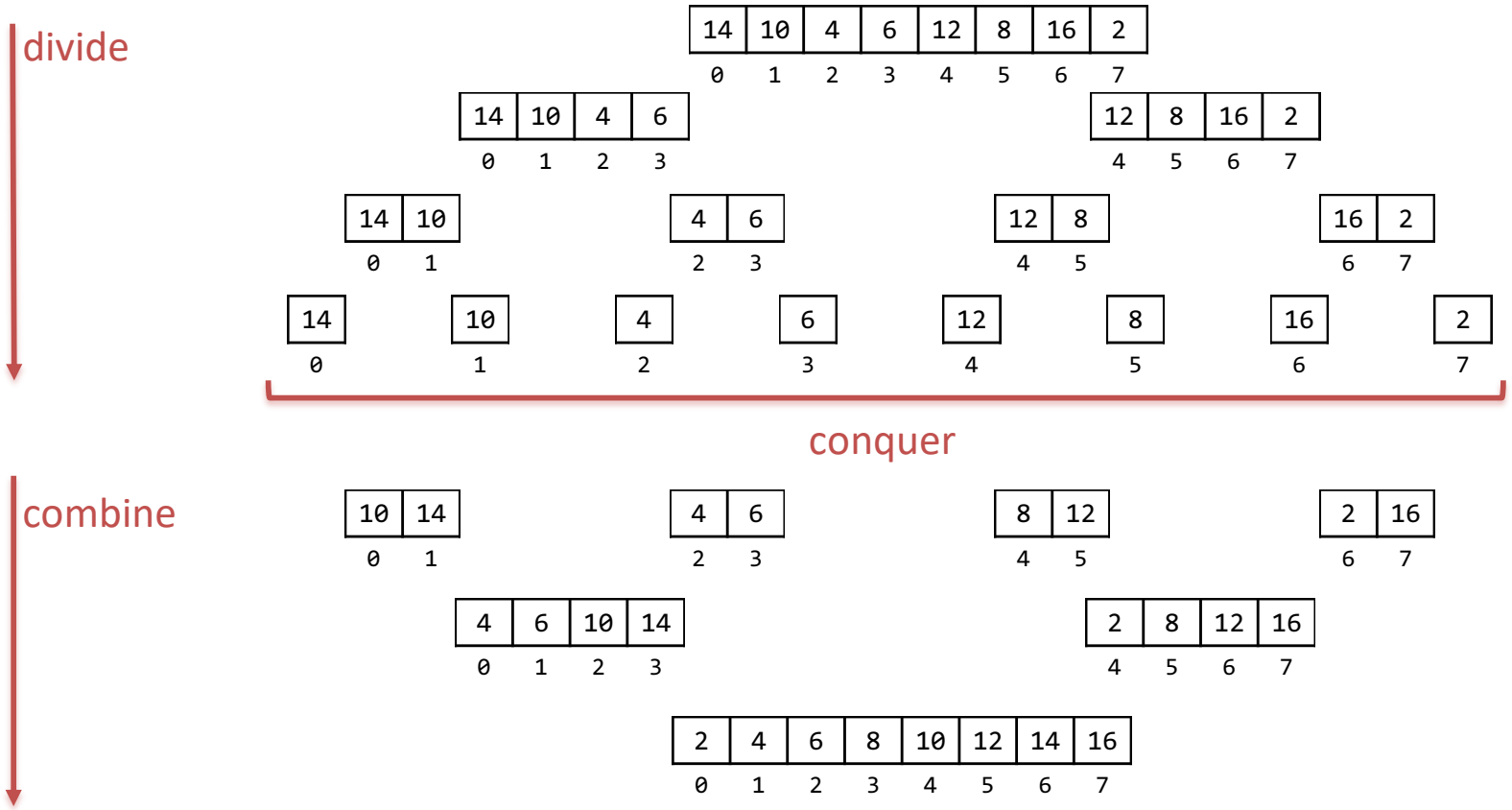
```
public static void mergeSort(Comparable[] a, int left, int right) {

   if (right == left) return;

   int mid = left + (right - left) / 2;
   mergeSort(a, left, mid);
   mergeSort(a, mid + 1, right);

   merge(a, left, mid, right);
}
```

combine



Merge sort

Truth in advertising: order of execution



Merge sort

The real work in merge sort happens during the combine phase: merging two sorted arrays into one sorted array.

```
public static void mergeSort(Comparable[] a, int left, int right) {

   if (right == left) return;

   int mid = left + (right – left) / 2;
   mergeSort(a, left, mid);
   mergeSort(a, mid + 1, right);

   merge(a, left, mid, right);
}
```

Merge sort

The real work in merge sort happens during the combine phase: merging two sorted arrays into one sorted array.

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i])    a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```

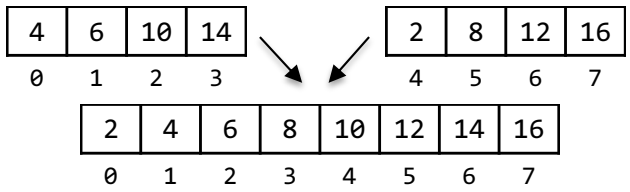
A field in the enclosing class.

O(N) extra space

Time complexity:

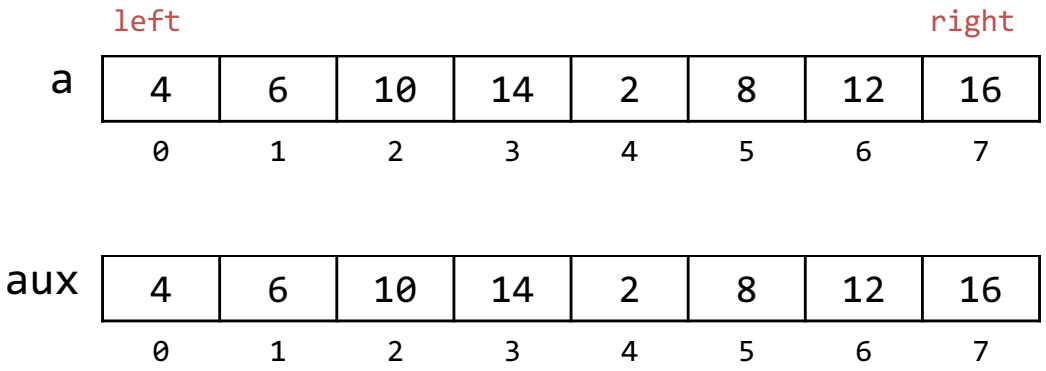
O(N)

O(N)



Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i])    a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```

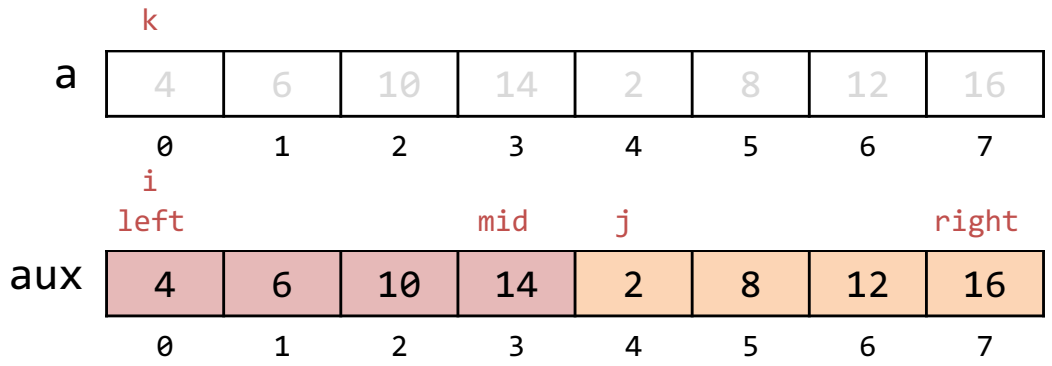


Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```

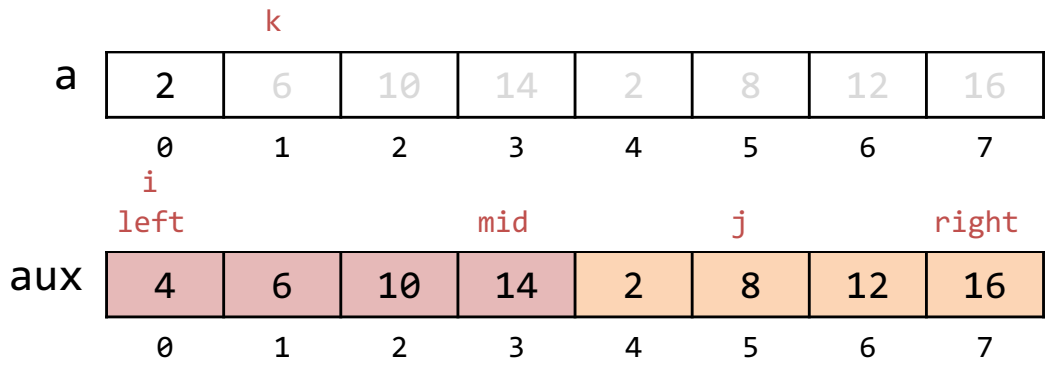
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



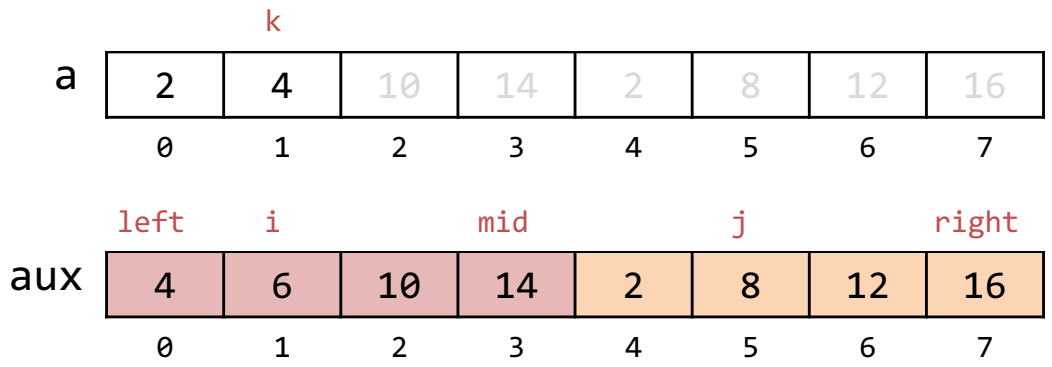
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



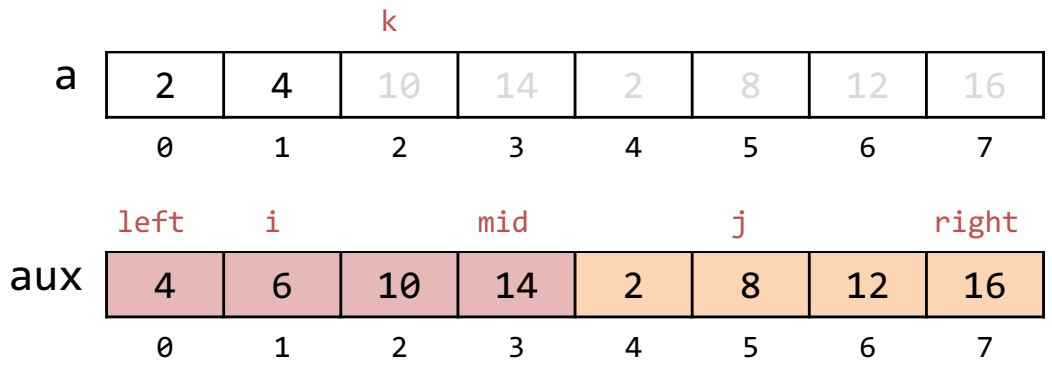
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



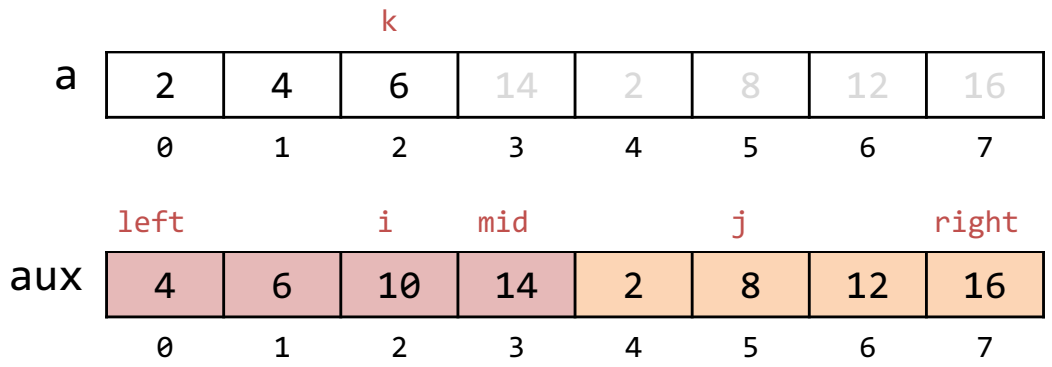
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



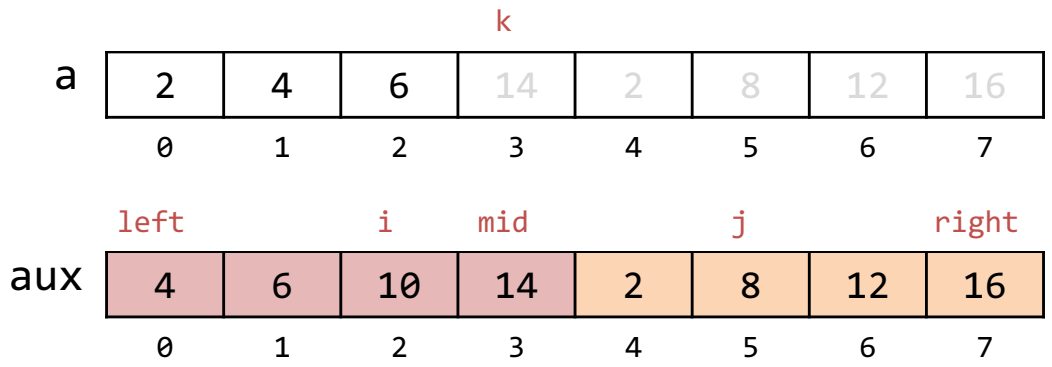
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



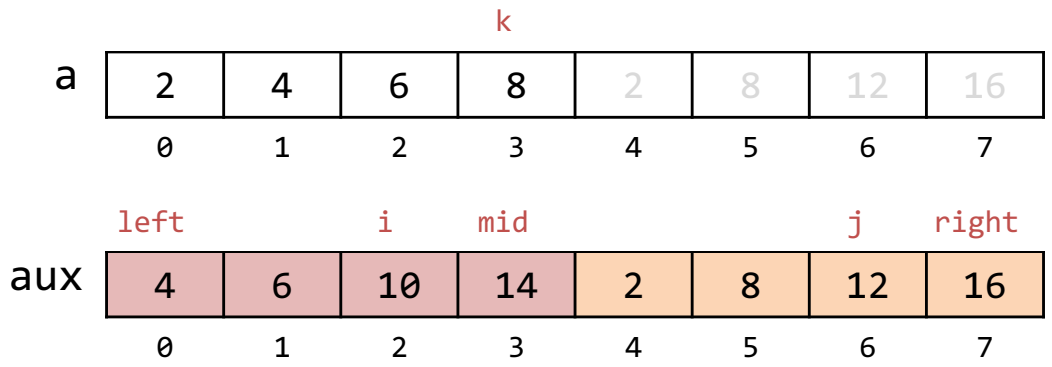
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



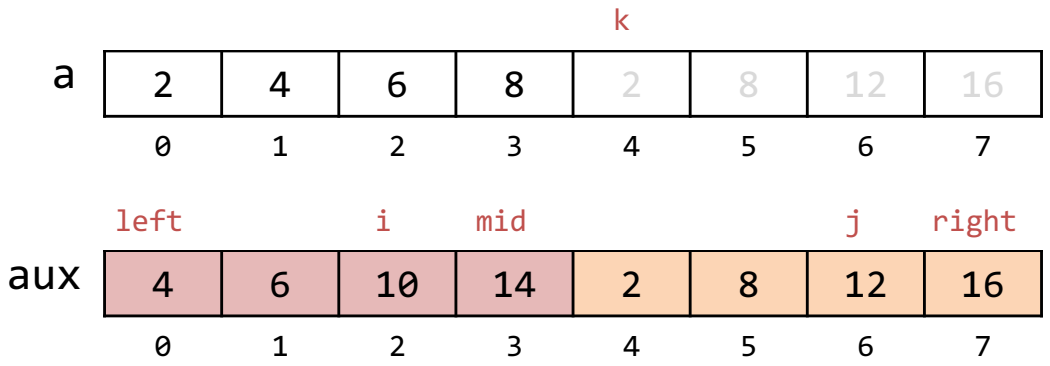
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



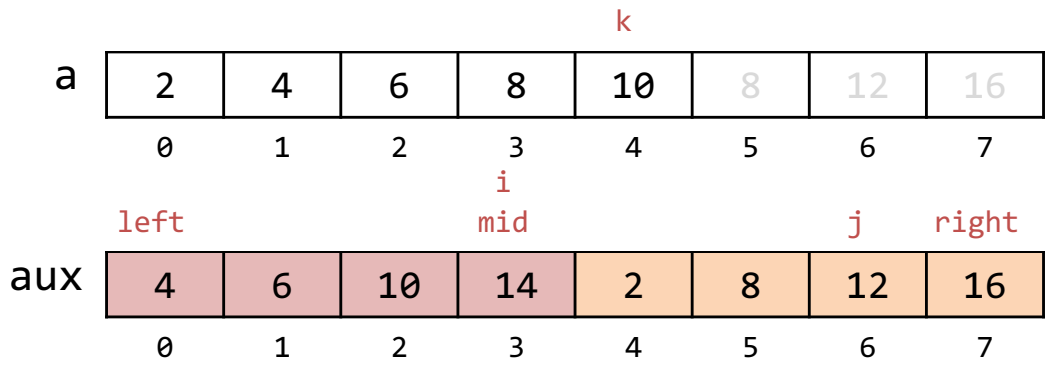
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



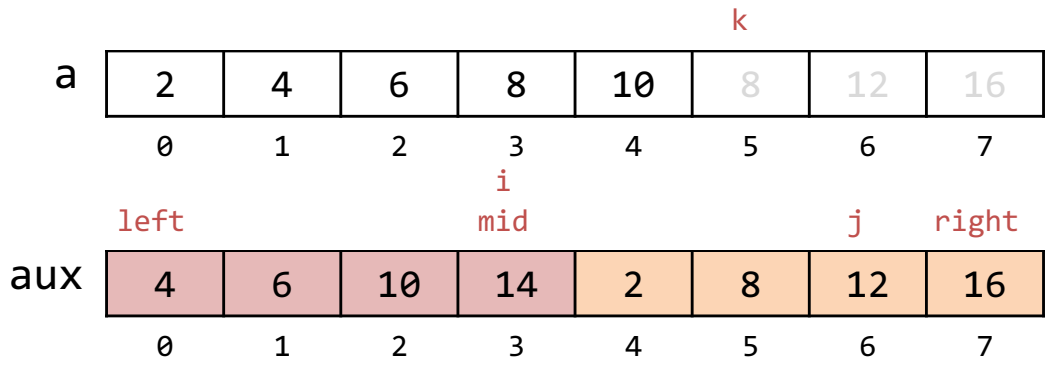
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



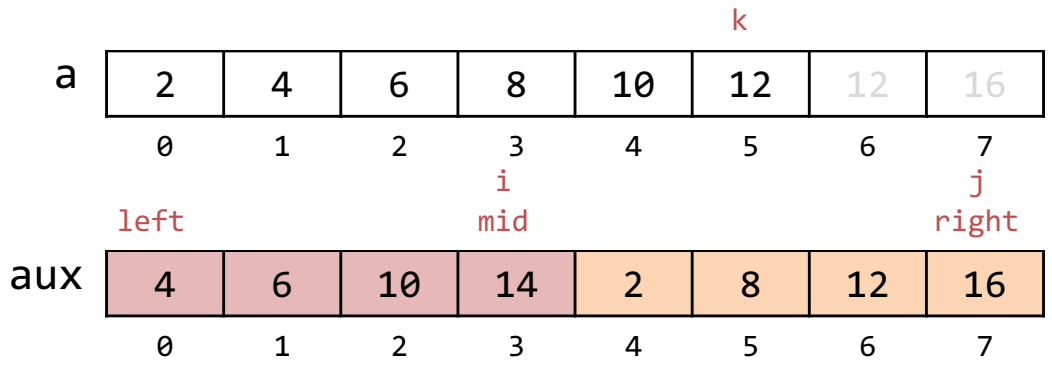
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



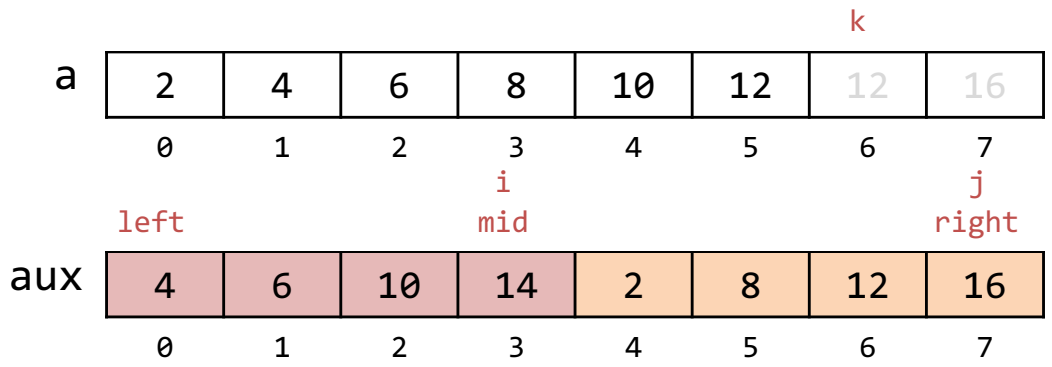
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



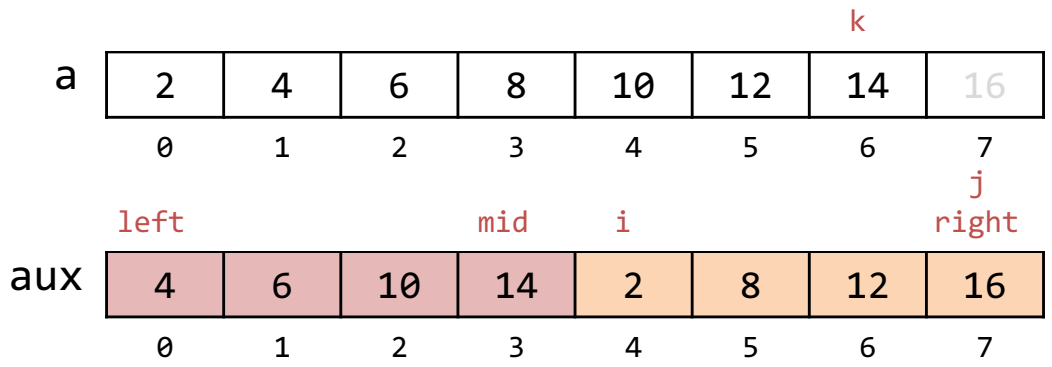
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



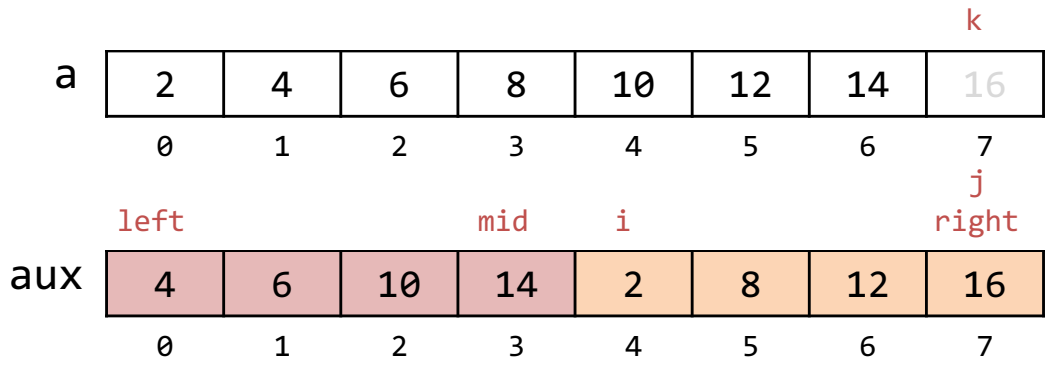
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



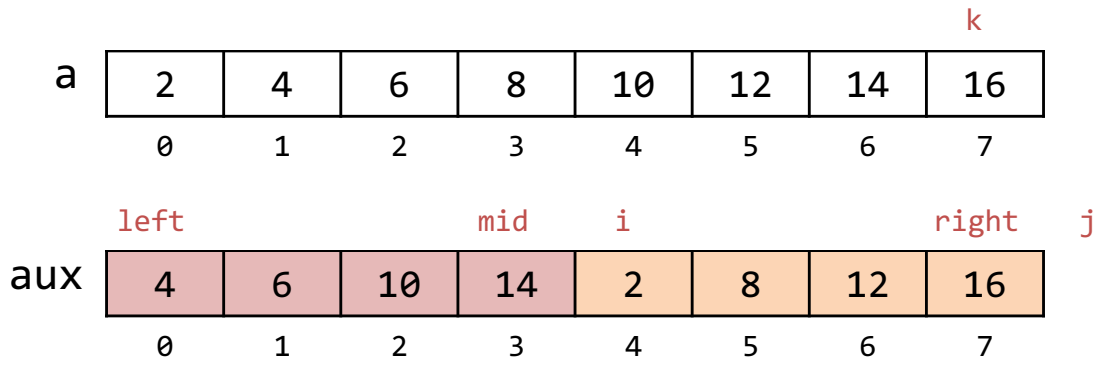
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



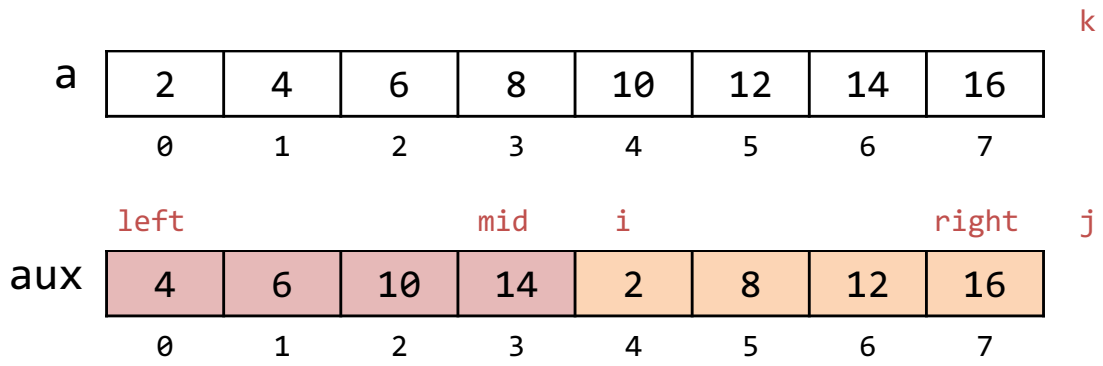
Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



Merge sort

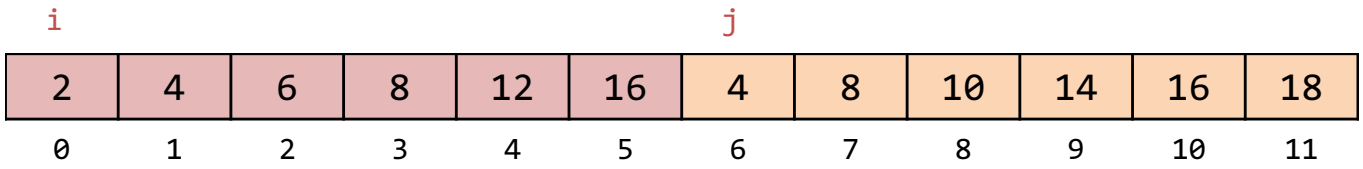
```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                a[k] = aux[j++];
      else if   (j > right)              a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))   a[k] = aux[j++];
      else                               a[k] = aux[i++];
   }
}
```



Merge sort

```
public void merge(Comparable[] a, int left, int mid, int right) {

   for (int k = left; k <= right; k++)   aux[k] = a[k];

   int i = left; j = mid + 1;
   for (int k = left; k <= right; k++) {
      if        (i > mid)                 a[k] = aux[j++];
      else if   (j > right)               a[k] = aux[i++];
      else if   (less(aux[j], aux[i]))    a[k] = aux[j++];
      else                                a[k] = aux[i++];
   }
}
```

Note that this makes merge sort *stable*.



Quicksort

Quicksort

Quicksort is a comparison sort based on the **divide-and-conquer** strategy.

First described by Tony Hoare in 1960 as part of his work on machine translation.

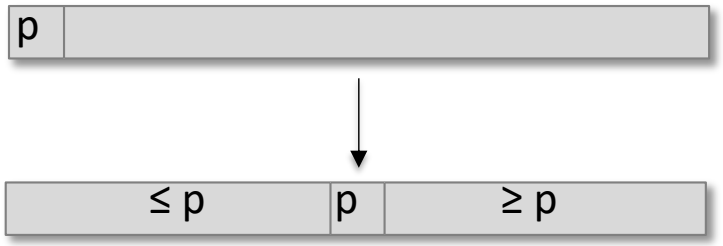
Asymptotically optimal for comparison sorting – **O(NlogN)**, but only in the average case. Has worst-case time that **is O(N^2)**.

Divide: Select a pivot then partition the array so that:

- pivot is in its correct sorted position
- no larger element is to the left of pivot
- no smaller element is to the right of pivot

Conquer: Sort each partition (recursively)

Combine: Nothing to do



The partitioning can be done in linear time.

The choice of pivot determines the size of the partitions.

Quicksort

```
public void qsort(Comparable[] a, int left, int right)
{
    if (right <= left)
       return;

    int j = partition(a, left, right);
    qsort(a, left, j-1);
    qsort(a, j+1, right);
 }
```

conquer

divide

Partitioning does the real work of the sort, so there's nothing to do in the combine phase.

Quicksort

```
public void qsort(Comparable[] a, int left, int right)
{
    if (right <= left)
       return;

    int j = partition(a, left, right);
    qsort(a, left, j-1);
    qsort(a, j+1, right);
 }
```

2	16	4	8	12	6	4	10	16	8	18	14
0	1	2	3	4	5	6	7	8	9	10	11

pivot = 10

2	4	8	6	4	8	10	14	16	16	18	12
0	1	2	3	4	5	6	7	8	9	10	11

Quicksort

```
public void qsort(Comparable[] a, int left, int right)
{
    if (right <= left)
       return;

    int j = partition(a, left, right);
    qsort(a, left, j-1);
    qsort(a, j+1, right);
 }
```

The partition method takes the array and (at least) the left and right indexes of the current portion of the array, and returns the resulting index of the pivot value after partitioning.

Notice that the partition method above chooses the pivot value internally. We could move that decision out to the qsort method, as the following example assumes.

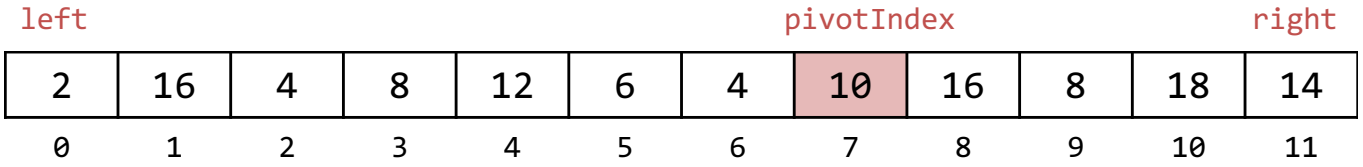
Quicksort

There are several ways to approach partitioning. Here's one that's easy to follow and has the pivot location provided as a parameter:

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```

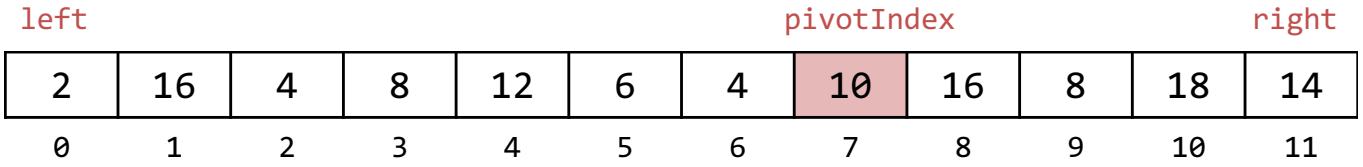
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



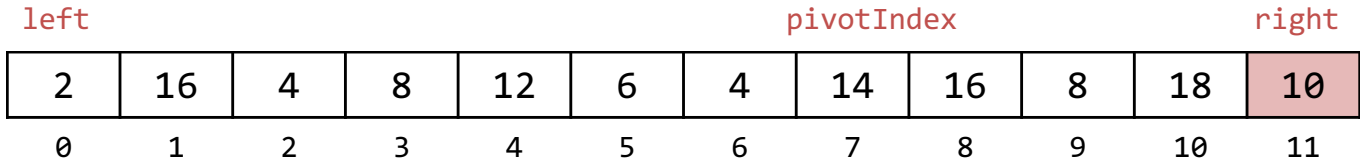
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



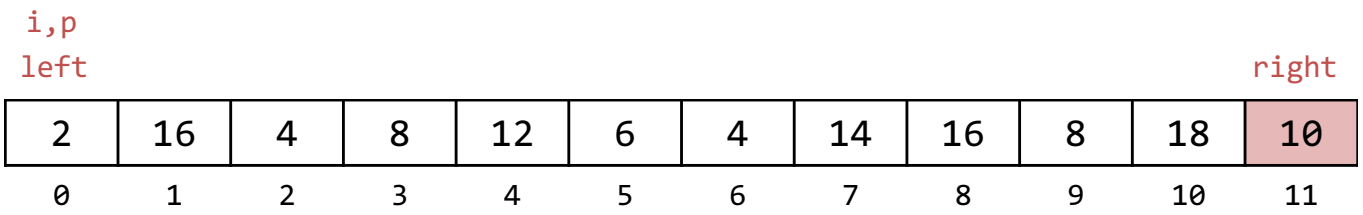
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



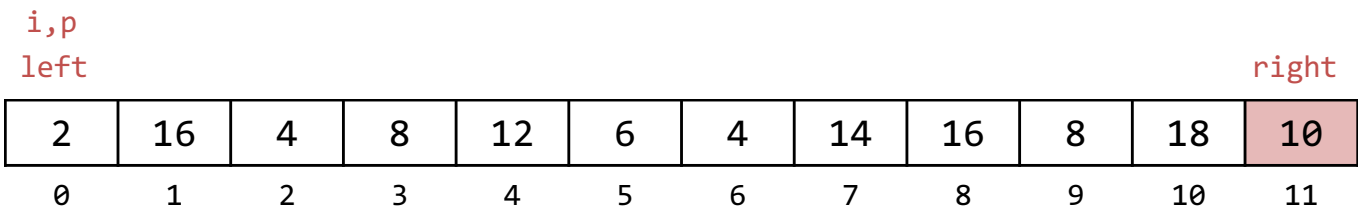
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



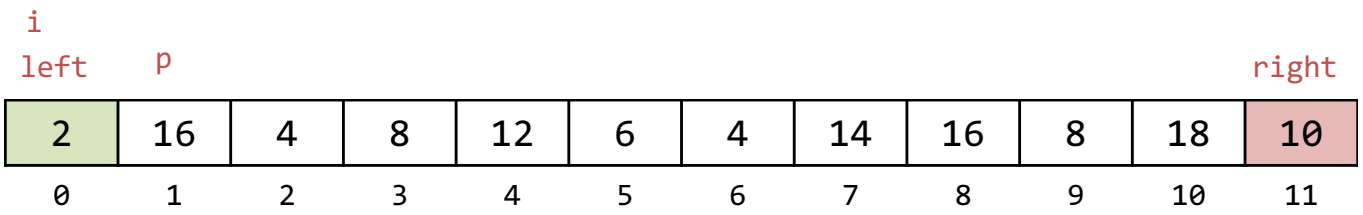
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



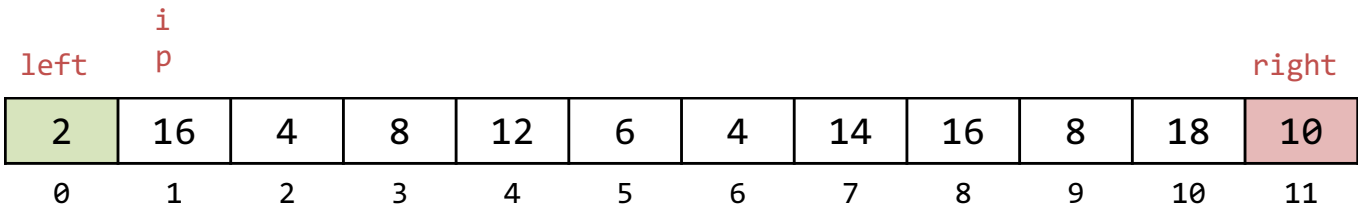
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



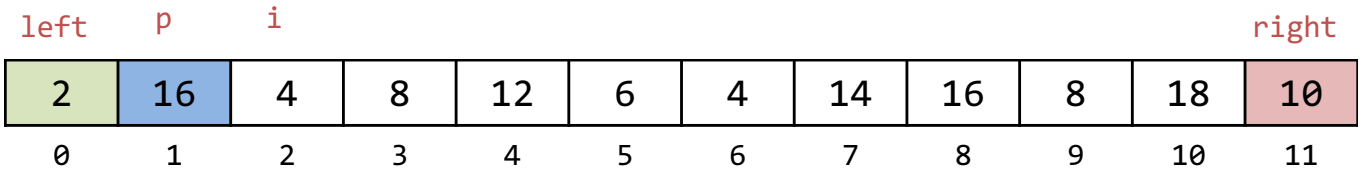
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



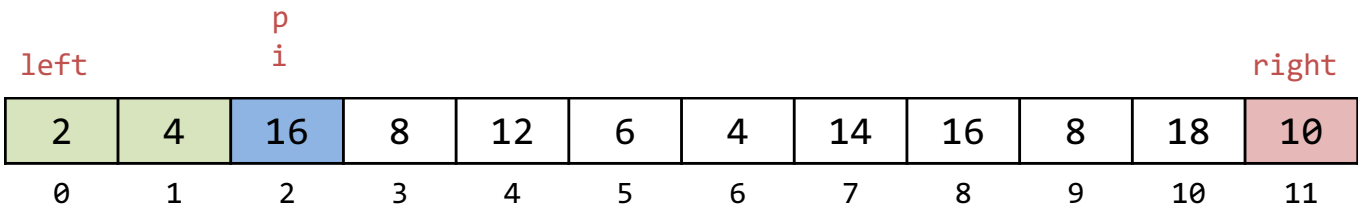
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



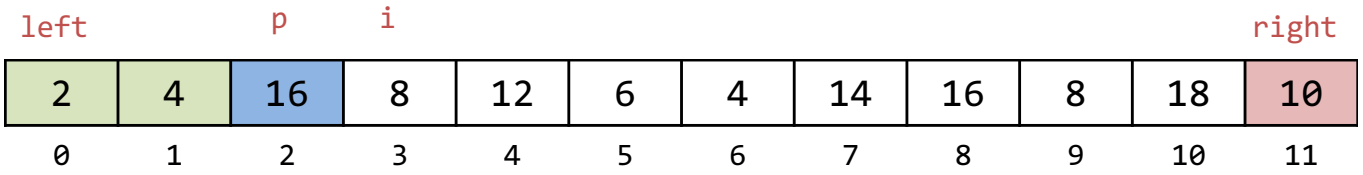
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



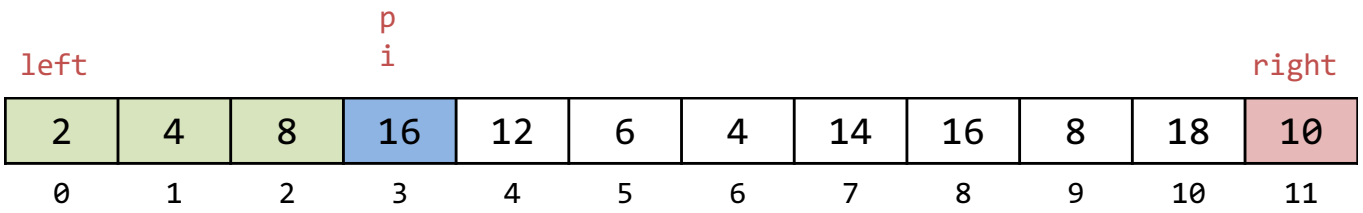
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



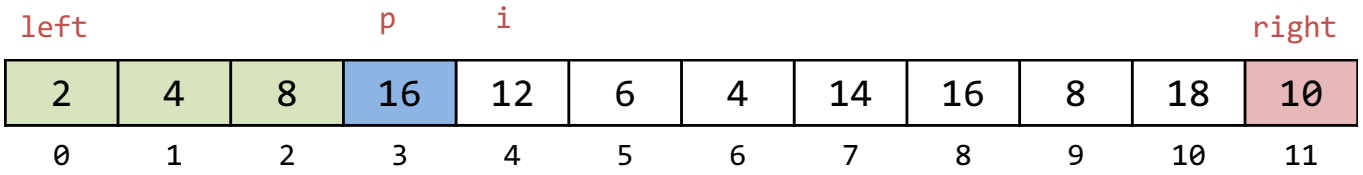
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



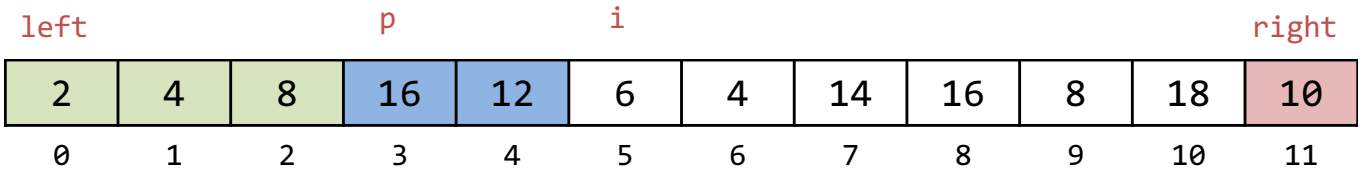
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



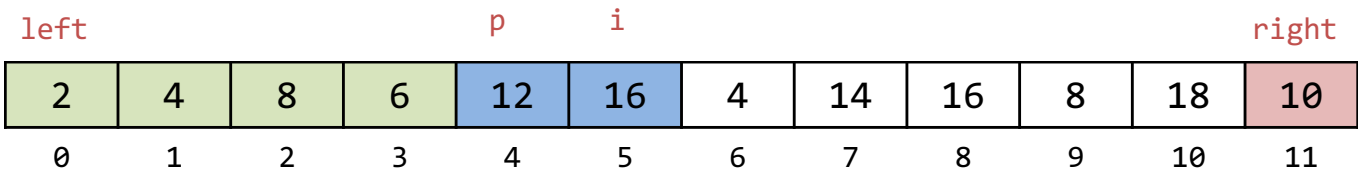
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



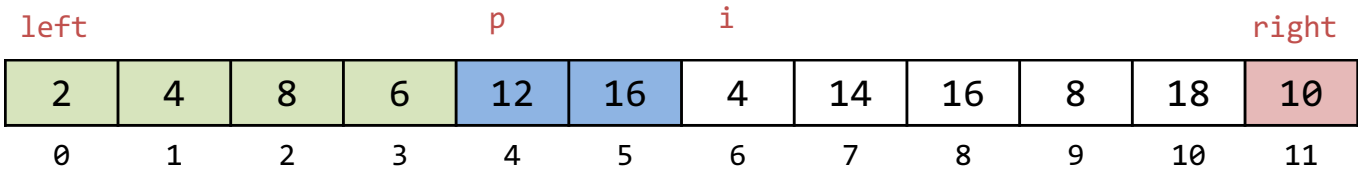
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



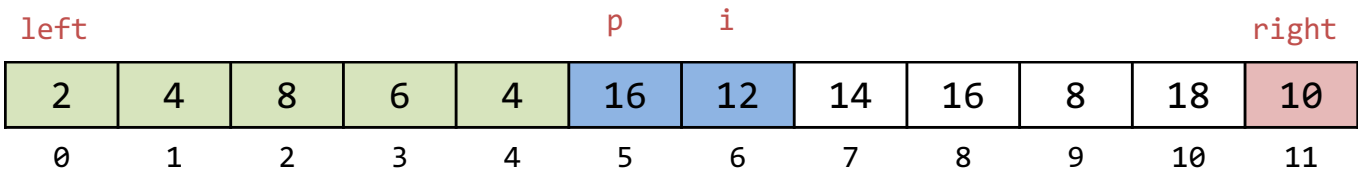
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



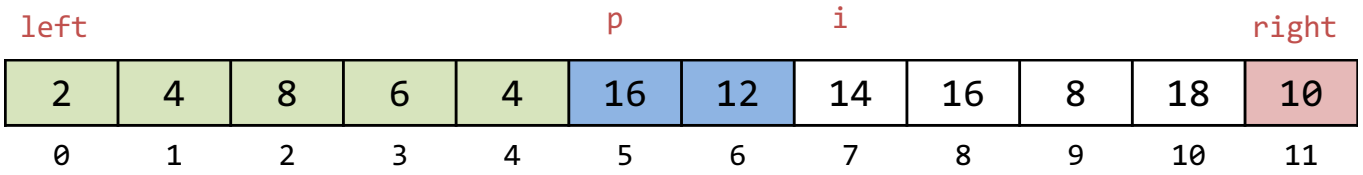
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



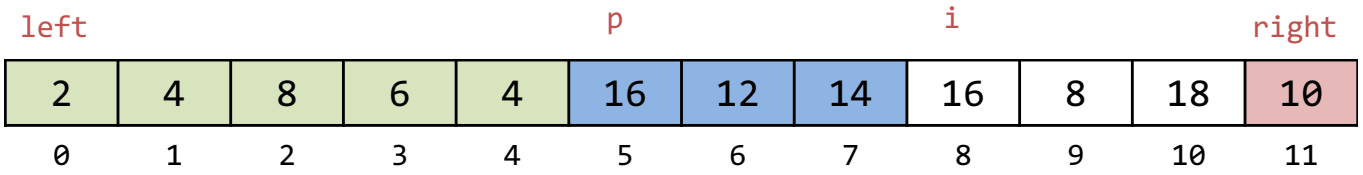
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



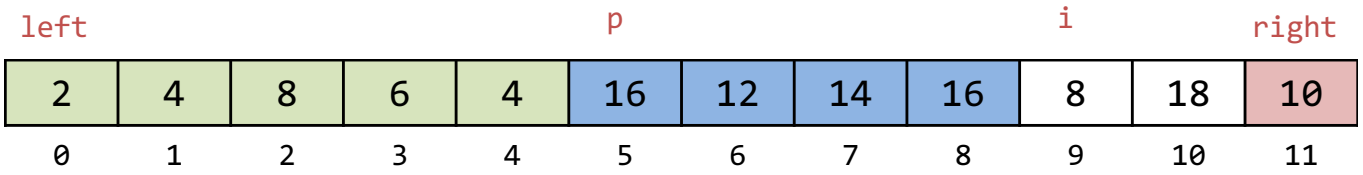
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



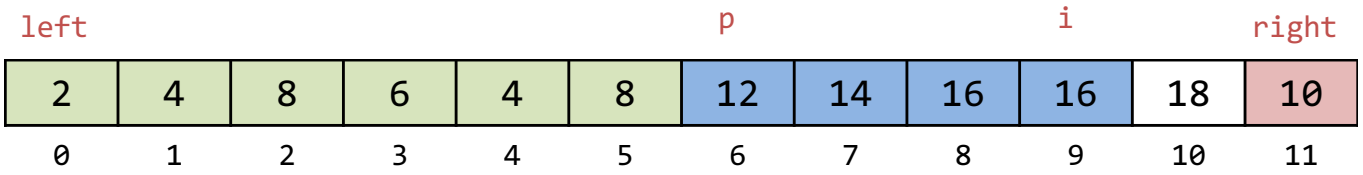
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



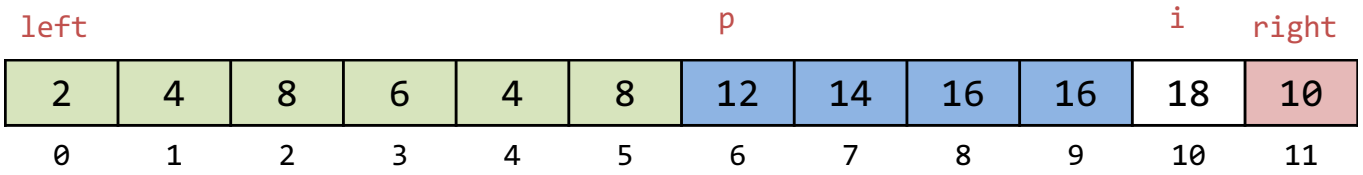
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



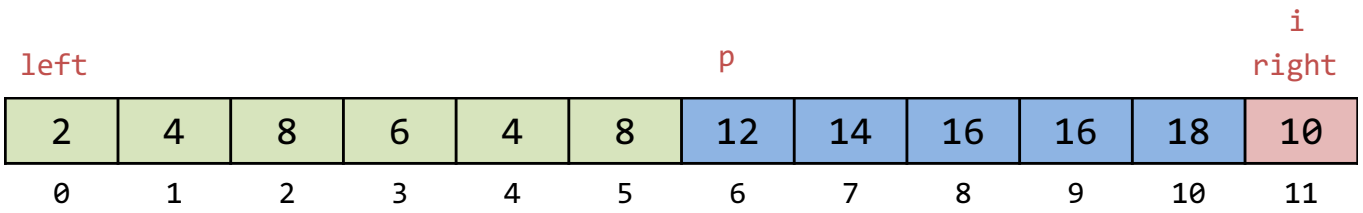
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



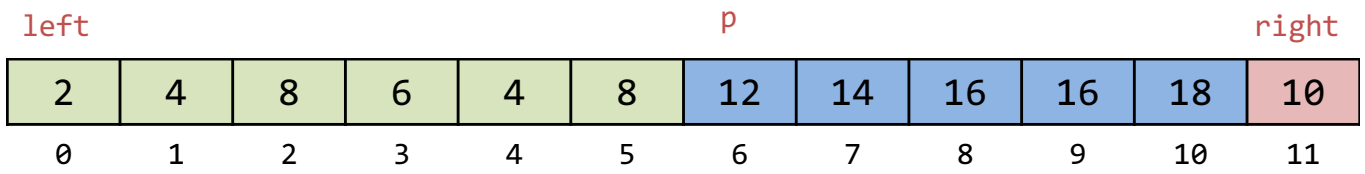
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



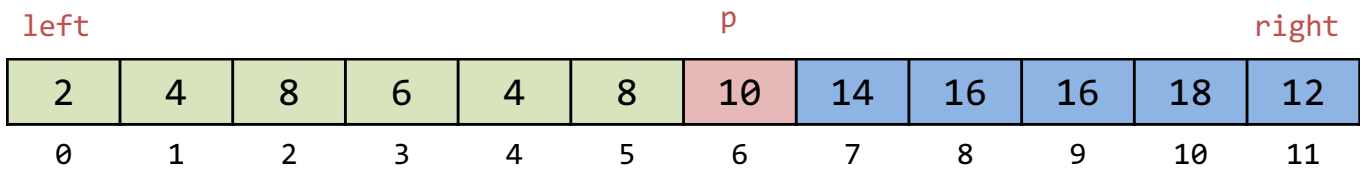
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



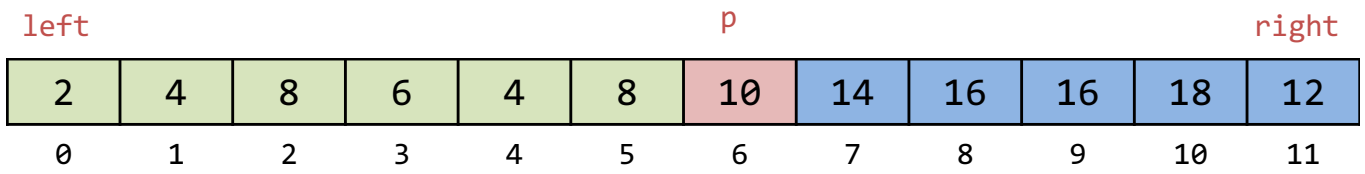
Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



Quicksort

```
private int partition(T[] a, int left, int right, int pivotIndex) {
   T pivot = a[pivotIndex];
   swap(a, pivotIndex, right); // move pivot to the end
   int p = left; // p will become the final index of pivot
   for (int i = left; i < right; i++) {
      if (less(a[i], pivot)) {
         swap(a, i, p);
         p++;
      }
   }
   swap(a, p, right); // move pivot to its correct location
   return p;
}
```



Quicksort

```
public void qsort(Comparable[] a, int left, int right)
{
    if (right <= left)
       return;

    int j = partition(a, left, right);
    qsort(a, left, j-1);
    qsort(a, j+1, right);
 }
```

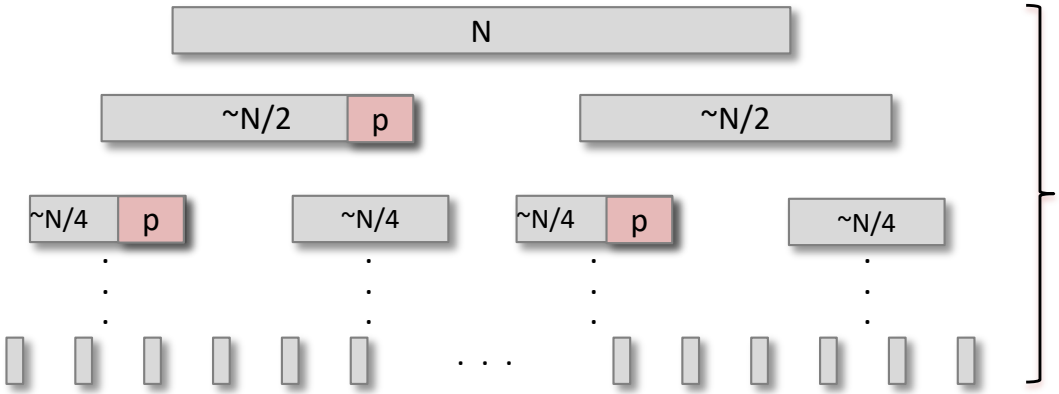
The choice of pivot value determines the size of each partition, and therefore determines the number of divide steps that will be necessary.

"*It becomes evident that sorting on the basis of Quicksort is somewhat like a gamble in which one should be aware of how much one may afford to lose if bad luck were to strike.*" – Niklaus Wirth

Quicksort

The choice of pivot value determines the size of each partition, and therefore determines the number of divide steps that will be necessary.

Best case pivot choices at each step lead to partitions that are approximately the same size.



~$\log_2$N levels

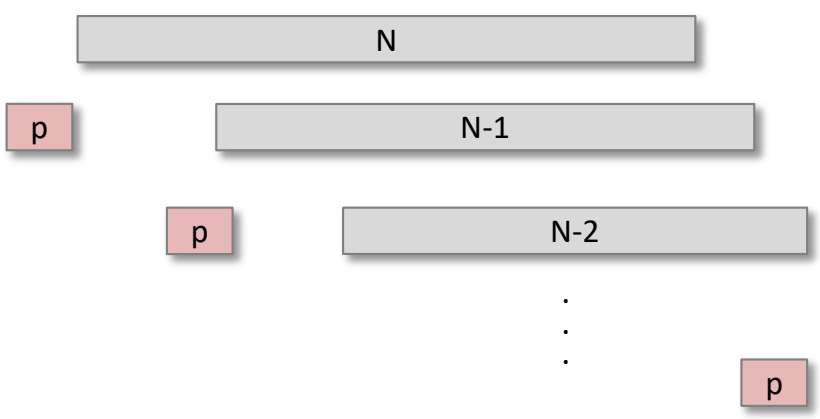
N elements to compare, move per level

O(NlogN) work

Quicksort

The choice of pivot value determines the size of each partition, and therefore determines the number of divide steps that will be necessary.

Worst case pivot choices at each step lead to one partition that is empty.



~N levels

N elements to compare, move per level

$O(N^2)$ work

Quicksort

Example partition based on a given pivot value for the following array:

2	16	4	8	12	6	4	10	16	8	18	14
0	1	2	3	4	5	6	7	8	9	10	11

pivot = 6

2	4	4	6	12	14	16	10	16	8	18	8
0	1	2	3	4	5	6	7	8	9	10	11

Quicksort

Example partition based on a given pivot value for the following array:

2	16	4	8	12	6	4	10	16	8	18	14
0	1	2	3	4	5	6	7	8	9	10	11

pivot = 10

2	4	8	6	4	8	10	14	16	16	18	12
0	1	2	3	4	5	6	7	8	9	10	11

Quicksort

Example partition based on a given pivot value for the following array:

2	16	4	8	12	6	4	10	16	8	18	14
0	1	2	3	4	5	6	7	8	9	10	11

pivot = 2

2	16	4	8	12	6	4	10	16	8	18	14
0	1	2	3	4	5	6	7	8	9	10	11

Quicksort

Example partition based on a given pivot value for the following array:

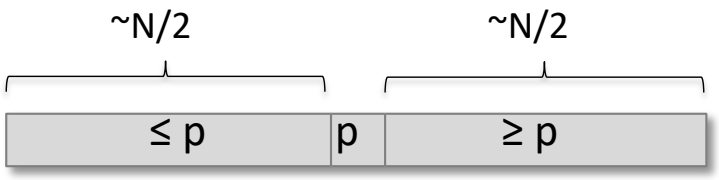
2	16	4	8	12	6	4	10	16	8	18	14
0	1	2	3	4	5	6	7	8	9	10	11

pivot = 18

2	16	4	8	12	6	4	10	16	8	14	18
0	1	2	3	4	5	6	7	8	9	10	11

Quicksort

Choosing a pivot value



Median value

The median of a list of N items can be found in O(N) time using a *selection* algorithm.

But, this linear time overhead would be added to each divide step in quicksort.

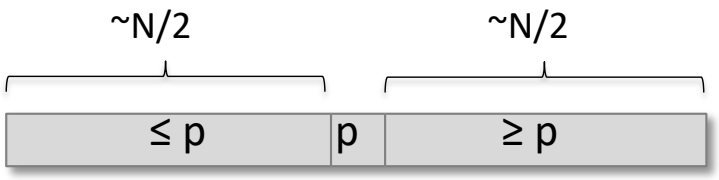
Median of three

Choose three elements of the array (usually first, middle, last) and then use the median of those three values as the pivot.

This is a good compromise and a popular choice in quicksort implementations. A proper analysis of quicksort will show that even though the partitions may not be ~N/2 at any given divide step, the overall performance will be O(NlogN) on average.

Quicksort

Choosing a pivot value



Randomly pick

Use a random number generator (PRNG) to pick an index and use the element at that index as the pivot. Again, a proper analysis will show that this will lead to O(NlogN) average case complexity.

But, this makes the sort itself dependent on a PRNG and not portable or deterministic.

Shuffle once, pick first element

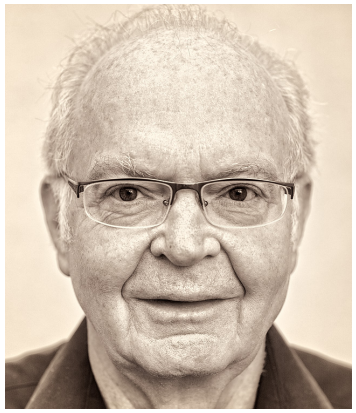
Randomize the order of elements in the array once up front (before the D&C part of the sort begins) and then just pick the first element as the pivot each time.

This is really just a variation of the first approach, but it pulls all calls to the PRNG out of the sort and make the pivot choice trivial (and fast).

Quicksort

Randomize the order of elements in an array in O(N) time.

```
public void shuffle(T[] a) {
   Random rng = new Random();
   for (int i = a.length - 1; i > 0; i--) {
      int j = rng.nextInt(i + 1);
      swap(a, i, j);
   }
}
```



This algorithm is known as the Knuth Shuffle (or the Fisher-Yates shuffle). It was first described by Ronald Fisher and Frank Yates, and then popularized by Don Knuth in TAOCP.

Quicksort

A "randomized" quicksort:

```
public void quicksort(Comparable[] a) {
   shuffle(a);
   qsort(a, 0, a.length - 1);
}
```

Shuffling happens once, before the sort begins.

```
public void qsort(Comparable[] a, int left, int right) {
   if (right <= left)
      return;

   int j = partition(a, left, right);
   qsort(a, left, j-1);
   qsort(a, j+1, right);
}
```

The partition method can use a[left] as the pivot – trivial and fast pivot selection.

Quicksort

There are many variations on the implementation of quicksort, but when it's done well, quicksort is typically the sort-of-choice in many situations.

Quicksort has a worst case complexity of $O(N^2)$. However, a good pivot choice strategy makes the worst case highly unlikely.

Because of this, we categorize quicksort by its average case complexity – **O(NlogN)**.

Note that quicksort **is an in-place sort**, but is usually **not stable**.

Sorts in the JCF / Stability

Sorts used in the JCF

The JCF offers sorting methods in both the Arrays class and the Collections class.

Quicksort is used for primitives.

> *"The sorting algorithm is a Dual-Pivot **Quicksort** by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations."*

Merge sort is used for references.

> *Implementation note: This implementation is a stable, adaptive, iterative **mergesort** that requires far fewer than n lg(n) comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to n/2 object references for randomly ordered input arrays. The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array. The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techiques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.*

Stability

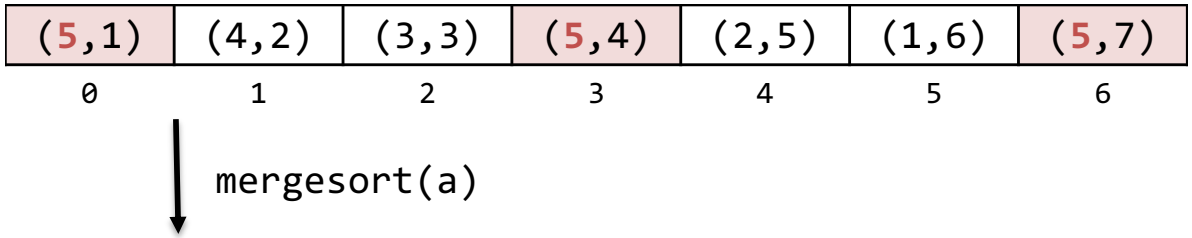
```
private static class Data implements Comparable<Data> {
   private Integer field1;
   private Integer field2;

   public Data(Integer f1, Integer f2) { . . . }

   public int compareTo(Data that) {
      if (this.field1 < that.field1)
         return -1;
      else if (this.field1 > that.field1)
         return 1;
      else
         return 0;
   }
 . . .
}
```

Compares only on field1

```
Data d1 = new Data(4, 1);
Data d2 = new Data(3, 9);

d1.compareTo(d2) ==
```

Stability

```
Data[] a = {new Data(5, 1), new Data(4, 2), new Data(3, 3), new Data(5, 4),
            new Data(2, 5), new Data(1, 6), new Data(5, 7)};
```



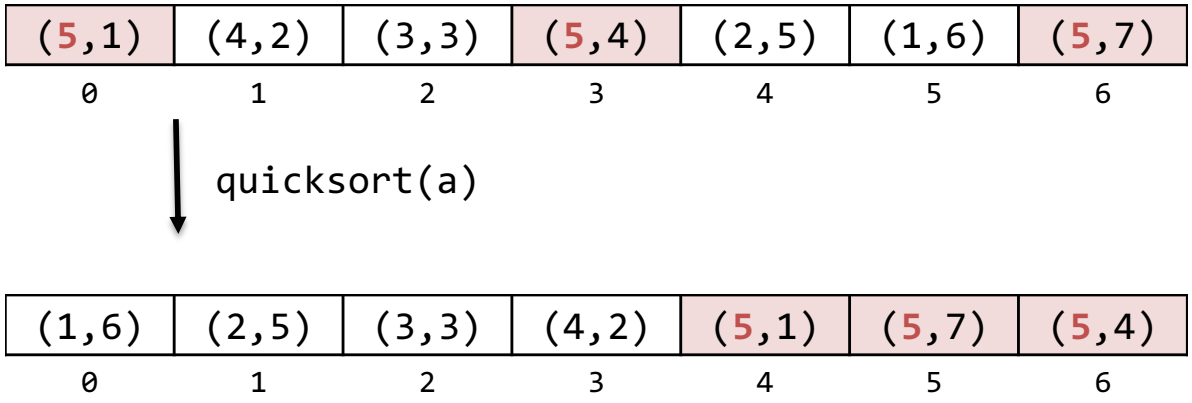
Three duplicates according to natural order.

The duplicates are in the same relative order as before the sort.

Merge sort is stable.

Stability

```
Data[] a = {new Data(5, 1), new Data(4, 2), new Data(3, 3), new Data(5, 4),
            new Data(2, 5), new Data(1, 6), new Data(5, 7)};
```



Three duplicates according to natural order.

The duplicates are not in the same relative order as before the sort.

Quicksort is not stable.

Quicksort could be stable, but typical implementations aren't in the interest of efficiency.

Stability

```
class Student implements Comparable<Student> {
   private String fname;
   private String lname;
   private int    section;

   . . .

   public String toString() {
      return lname + ", " + fname + ", " + section;
   }

   public int compareTo(Student s) {
      return this.toString().compareTo(s.toString());
   }
}
```

Stability

```
Student[] roll = new Student[7];
// new Student(“last name”, “first name”, section)
roll[0] = new Student("Wei", "Patricia", 1);
roll[1] = new Student("Dipaolo", "Jordan", 1);
roll[2] = new Student("Cullins", "Janna", 2);
roll[3] = new Student("Center", "Wallace", 2);
roll[3] = new Student("Trojacek", "Evelyn", 3);
roll[4] = new Student("Buffum", "Tawana", 3);
roll[5] = new Student("Laber", "Sharyl", 3);
roll[6] = new Student("Hageman", "Rachel", 2);

shuffle(roll);

Comparator<Student> orderByName = new CompareStudentsByName();
Comparator<Student> orderBySection = new CompareStudentsBySection();
```

Stability

```
print(a);

quicksort(roll, orderByName);
print(a);

quicksort(roll, orderBySection);
print(a);
```

```
% java SortStabilityExample

Laber,    Sharyl,   3
Hageman,  Rachel,   2
Cullins,  Janna,    2
Dipaolo,  Jordan,   1
Buffum,   Tawana,   3
Trojacek, Evelyn,   3
Wei,      Patricia, 1
```

Stability

```
print(a);

quicksort(roll, orderByName);
print(a);

quicksort(roll, orderBySection);
print(a);
```

```
% java SortStabilityExample


. . .

Buffum,   Tawana,   3
Cullins,  Janna,    2
Dipaolo,  Jordan,   1
Hageman,  Rachel,   2
Laber,    Sharyl,   3
Trojacek, Evelyn,   3
Wei,      Patricia, 1
```

Stability

```
print(a);

quicksort(roll, orderByName);
print(a);

quicksort(roll, orderBySection);
print(a);
```

```
% java SortStabilityExample

. . .

. . .

Wei,       Patricia, 1
Dipaolo,   Jordan,   1
Hageman,   Rachel,   2
Cullins,   Janna,    2
Laber,     Sharyl,   3
Buffum,    Tawana,   3
Trojacek,  Evelyn,   3
```

Stability

```
print(a);

mergesort(roll, orderByName);
print(a);

mergesort(roll, orderBySection);
print(a);
```

```
% java SortStabilityExample

Laber,    Sharyl,   3
Hageman,  Rachel,   2
Cullins,  Janna,    2
Dipaolo,  Jordan,   1
Buffum,   Tawana,   3
Trojacek, Evelyn,   3
Wei,      Patricia, 1
```

Stability

```
print(a);

mergesort(roll, orderByName);
print(a);

mergesort(roll, orderBySection);
print(a);
```

```
% java SortStabilityExample


. . .

Buffum,   Tawana,   3
Cullins,  Janna,    2
Dipaolo,  Jordan,   1
Hageman,  Rachel,   2
Laber,    Sharyl,   3
Trojacek, Evelyn,   3
Wei,      Patricia, 1
```

Stability

```
print(a);

mergesort(roll, orderByName);
print(a);

mergesort(roll, orderBySection);
print(a);
```

```
% java SortStabilityExample

. . .

. . .

Dipaolo,  Jordan,   1
Wei,      Patricia, 1
Cullins,  Janna,    2
Hageman,  Rachel,   2
Buffum,   Tawana,   3
Laber,    Sharyl,   3
Trojacek, Evelyn,   3
```

Sorted by name within section.

Summary

Here's a summary of the four sorting algorithms we've talked about – as illustrated by the given implementations – with respect to time complexity and basic properties of sorting.

	Selection	Insertion	Mergesort	Quicksort
Worst case	$O(N^2)$	$O(N^2)$	O(N log N)	$O(N^2)$
Average case	$O(N^2)$	$O(N^2)$	O(N log N)	O(N log N)
Best case	$O(N^2)$	O(N)	O(N log N)	O(N log N)
In-place?	Yes	Yes	No	Yes
Stable?	No	Yes	Yes	No
Adaptive?	No	Yes	No	No