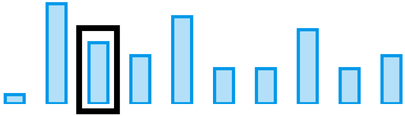
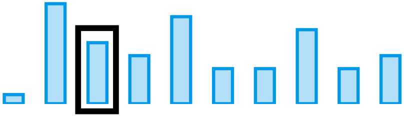
Quick Sort (Divide & Conquer)

1. Select an element of the array, which we call **pivot**.

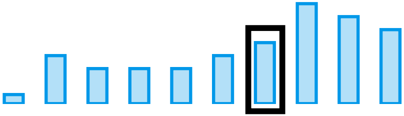


Quick Sort

1. Select an element of the array, which we call **pivot**.

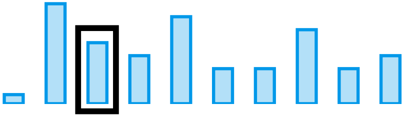


2. Partition the array so that the *"small entries"* ($\leq$ pivot) are on the left, then the pivot, then the *"large entries"* ($>$ pivot).

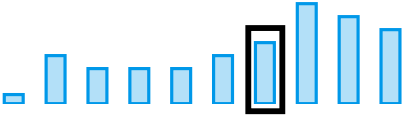


Quick Sort

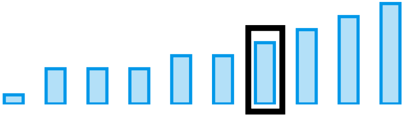
1. Select an element of the array, which we call **pivot**.



2. Partition the array so that the *"small entries"* ($\leq$ pivot) are on the left, then the pivot, then the *"large entries"* ($>$ pivot).



3. Recursively (quick)sort the two partitions.



For the time being it is not important how the pivot is selected. We will see later that there are different strategies that select the pivot and they might affect the time complexity of quicksort.

Remark: In order for quicksort to be a *stable* sorting algorithm, it is useful to allow the *large entries* to also be $\geq$ pivot.

On the other hand, it is easier to understand how quicksort works if we require the large entries to be strictly larger than the pivot. Of course, this is only an issue if there are duplicate values in the array.

Example: Quick Sort run

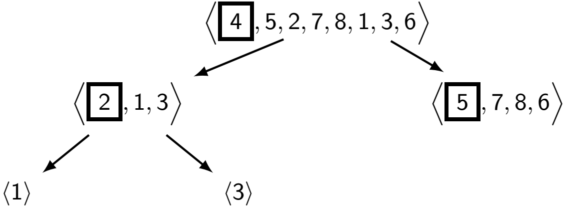
Pivot selection strategy: we always choose the leftmost entry.

$$\left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle$$

$$\left\langle \boxed{2}, 1, 3 \right\rangle \qquad\qquad \left\langle \boxed{5}, 7, 8, 6 \right\rangle$$

Example: Quick Sort run

Pivot selection strategy: we always choose the leftmost entry.



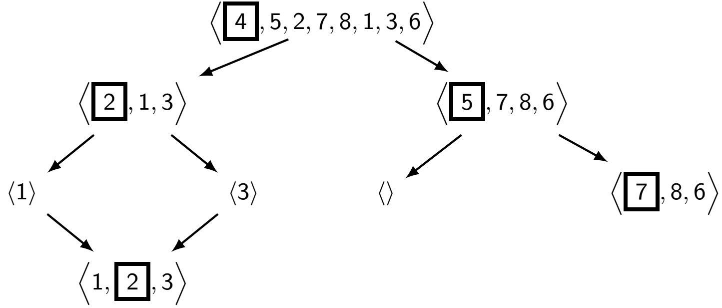
Example: Quick Sort run

Pivot selection strategy: we always choose the leftmost entry.

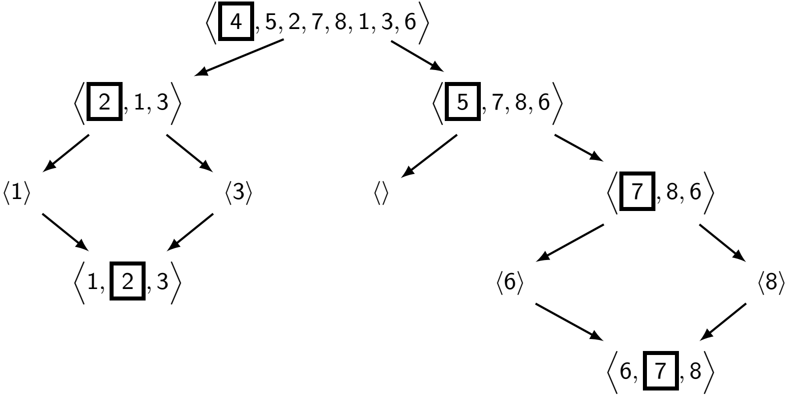
$$
\begin{tikzcd}
 & & & \left\langle \boxed{4}, 5, 2, 7, 8, 1, 3, 6 \right\rangle \arrow[dll] \arrow[dr] & \\
 & \left\langle \boxed{2}, 1, 3 \right\rangle \arrow[dl] \arrow[dr] & & & \left\langle \boxed{5}, 7, 8, 6 \right\rangle \\
\langle 1 \rangle \arrow[dr] & & \langle 3 \rangle \arrow[dl] & & \\
 & \left\langle 1, \boxed{2}, 3 \right\rangle & & &
\end{tikzcd}
$$

Example: Quick Sort run

Pivot selection strategy: we always choose the leftmost entry.

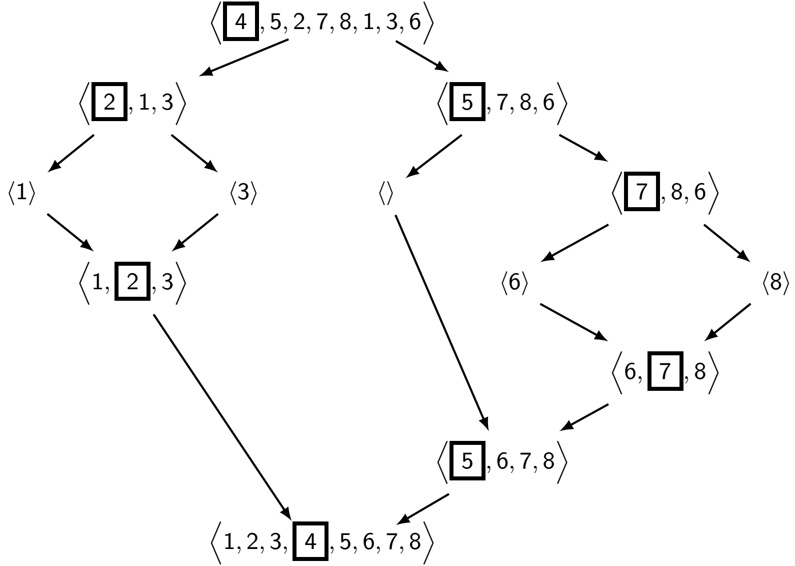


Pivot selection strategy: we always choose the leftmost entry.



Example: Quick Sort run

Pivot selection strategy: we always choose the leftmost entry.



Quick Sort (pseudocode)

```
void quicksort(a, n){
    quicksort_run(a, 0, n-1)
}

quicksort_run(a, left, right) {
   if ( left < right ) {
       pivotindex = partition(a, left, right)
       quicksort_run(a, left, pivotindex-1)
       quicksort_run(a, pivotindex+1, right)
   }
}
```

Where `partition` rearranges the array so that

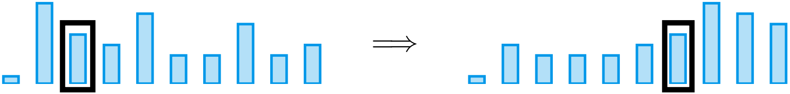
- the small entries are stored on positions `left, left+1, left+2, ..., pivot_index-1`,
- pivot is stored on position `pivot_index` and
- the large entries are stored on `pivot_index+1, pivot_index+2, ..., right`.

Partitioning array `a`

Idea:

1. Choose a pivot `p` from `a`.
2. Allocate two temporary arrays: `tmpLE` and `tmpG`.
3. Store all elements *less than or equal to* `p` to `tmpLE`.
4. Store all elements *greater than* `p` to `tmpG`.
5. Copy the arrays `tmpLE` and `tmpG` back to `a` and return the index of `p` in `a`.

The time complexity of partitioning is $O(n)$.



Partitioning array `a` in-place (unstable)

```
partition(array a, int left, int right) {
  pivotindex = choosePivot(a, left, right)
  pivot = a[pivotindex]
  swap a[pivotindex] and a[right]
  leftmark = left
  rightmark = right - 1
  while (leftmark <= rightmark) {
    while (leftmark <= rightmark and
           a[leftmark] <= pivot)
      leftmark++
    while (leftmark <= rightmark and
           a[rightmark] >= pivot)
      rightmark--
    if (leftmark < rightmark)
      swap a[leftmark++] and a[rightmark--]
  }
  swap a[leftmark] and a[right]
  return leftmark
}
```

Partitioning array `a`, using temporary storage (stable)

```
partition(array a, int left, int right) {
  create new array b of size right-left+1
  pivotindex = choosePivot(a, left, right)
  pivot = a[pivotindex]
  acount = left
  bcount = 1
  for ( i = left ; i <= right ; i++ ) {
    if ( i == pivotindex )
      b[0] = a[i]
    else if ( a[i] < pivot ||
              (a[i] == pivot && i < pivotindex) )
      a[acount++] = a[i]
    else
      b[bcount++] = a[i]
  }
  for ( i = 0 ; i < bcount ; i++ )
    a[acount++] = b[i]
  return right-bcount+1
}
```

Time Complexity of Quicksort

Best Case: If the pivot is the *median* in every iteration, then the two partitions have approximately $\frac{n}{2}$ elements.

$\Longrightarrow$ The time complexity is as for Merge Sort, i.e. $O(n \log n)$.

Time Complexity of Quicksort

Best Case: If the pivot is the *median* in every iteration, then the two partitions have approximately $\frac{n}{2}$ elements.

$\Longrightarrow$ The time complexity is as for Merge Sort, i.e. $O(n \log n)$.

Worst Case: If the pivot is always the *least* element in every iteration, then the second partition contains all elements except for the pivot; it has $n - 1$ elements. In the consecutive iterations:

the second partition has $n - 1, n - 2, n - 3, ..., 1$ elements.

$\Longrightarrow$ The time complexity is $O(n^2)$.

Time Complexity of Quicksort

Best Case: If the pivot is the *median* in every iteration, then the two partitions have approximately $\frac{n}{2}$ elements.

$\implies$ The time complexity is as for Merge Sort, i.e. $O(n \log n)$.

Worst Case: If the pivot is always the *least* element in every iteration, then the second partition contains all elements except for the pivot; it has $n-1$ elements. In the consecutive iterations:

the second partition has $n-1, n-2, n-3, \ldots, 1$ elements.

$\implies$ The time complexity is $O(n^2)$.

Average Case: Depends on the strategy which chooses the pivots! If there are $\geq 25\%$ many small entries or $\geq 25\%$ many large entries in almost every iteration, then the partitioning happens approximately $\log_{4/3} n$-many times

$\implies$ The time complexity is $O(n \log n)$.

Pivot-selection strategies

Choose pivot as:

1. the middle entry
   (good for sorted sequences, unlike the leftmost-strategy),
2. the median of the leftmost, rightmost and middle entries,
3. a random entry (there is 50% chance for a good pivot).

Remark: In practice, usually 3. or a variant of 2. is used.

Also, for both quicksort and mergesort, when you reach a small region that you want to sort, it's faster to use selection sort or other sort algorithms. The overhead of Q.S. or M.S. is big for small inputs.

Strategies (1) and (2) don't guarantee that the pivot will be such that $\geq$ 25% entries is small and $\geq$ 25% is large for *every input* sequence. However, this property holds *on average* (= for a random sequence).

Strategy (3), although it does not guarantee that we will find a perfect pivot every single time, we pick it *often* (with 50% probability) which suffices.

Comparison of sorting algorithms

	Selection Sort	Heap Sort	Merge Sort	Quick Sort temp array (stable)	Quick Sort in-place (unstable)
Time Complexity:					
Average C.	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Worst C.	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n^2)$
Space Complexity:					
Average C.	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
Worst C.	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Stability	No	No	Yes	Yes	No

So why is quicksort used so much if its Worst Case complexity is as bad as that of selection sort?

It is because quicksort's constants hidden by the big-O are *smaller*. However, if guaranteed $O(n \log n)$ time complexity is required, it is probably better to use merge sort. Moreover, if we are working with very restricted memory, then it is reasonable to also consider heap sort.