Quicksort

Merge sort:

- Divide input into two parts X and Y
- Sort each part separately (recursion)
- Merge the two sorted parts into one sorted part

Base case: $n \leq 1$ (already sorted, do nothing)

Quicksort

Merge sort:

- Divide input into two parts X and Y
- Sort each part separately (recursion)
- Merge the two sorted parts into one sorted part

Base case: $n \leq 1$ (already sorted, do nothing)

Quicksort:

- Divide input into two parts X and Y so $X \leq Y$
- Sort each part separately (recursion)
- Return X followed by Y

Base case: $n \leq 1$ (already sorted, do nothing)

[Hoare, 1960]

Quicksort

As pseudo-code:

```
QUICKSORT(A, p, r)
if p < r
    // Partition the subarray around the pivot, which ends up in A[q].
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)  // recursively sort the low side
    QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Quicksort

As pseudo-code:

(array, starting index, ending index)

```
QUICKSORT(A, p, r)
  if p < r
      // Partition the subarray around the pivot, which ends up in A[q].
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q - 1)  // recursively sort the low side
      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

A call to **Quicksort(A, p, r)** is responsible for arranging the elements in **A[p … r]** in sorted order.
The first call is **Quicksort(A, 1, n)**, which is responsible for sorting the entire array **A**.
A call to **Partition(A, p, r)** selects an element **x** ∈ **A** and partitions **A[p … r]** such that:

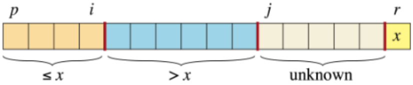
$$A[q] = x \qquad A[p \ldots q-1] \le x \qquad A[q+1 \ldots r] > x$$

Partition

How to implement Partition?

Idea: Choose an element **x** from the input to partition around (here, the last element in the array segment). Build the two parts during a single pass through the array based on the following.

Principle:



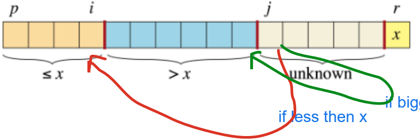
Partition

How to implement Partition?

Idea: Choose an element **x** from the input to partition around (here, the last element in the array segment). Build the two parts during a single pass through the array based on the following.

Principle:

p = staring index   i = ending of i<= x   j = ending of j>x   r = ending index



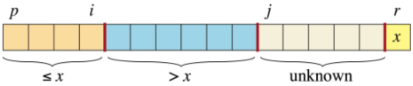
x = pivot element

if bigger than x

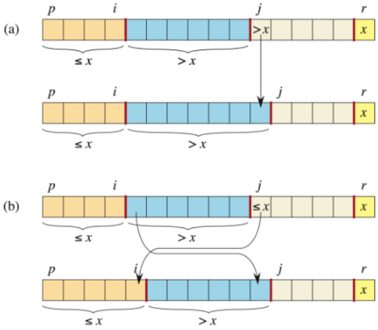
if less then x

*Choose a pivot element.
*Rearrange the subarray so that elements smaller than or equal to the pivot are on one side, and elements greater are on the other side.
*Place the pivot in its correct sorted position and return its index.

Principle:



How to take a step during the iteration?



j moves to the right, and compare the value to x.
here it is bigger, so it just stays.
And is know included in the part that is bigger than x

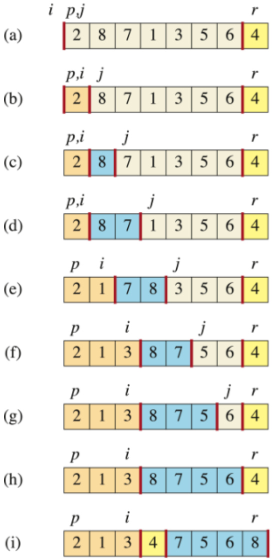
J moves again

j finds a value less than x, moves it to the end of p-i array

i is move, one spot because it just got bigger

Partition

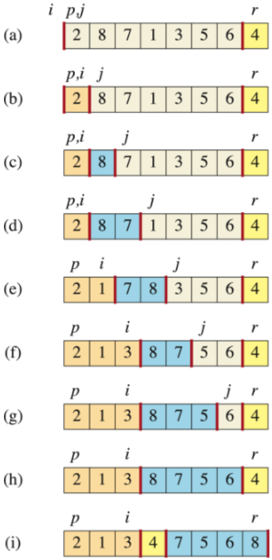
An example of iteration:



when it is <r, we change j
and i+1. Shifting the bluepart (bigger)

Partition

An example of iteration:



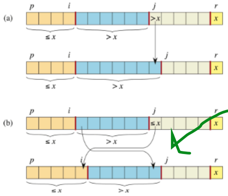
Time: **O(n)** where **n** is the number of elements in A[p . . .r].

Partition

As pseudo-code:

p start index, r end index

```
PARTITION(A, p, r)
x = A[r]                              // the pivot        last element is the pivot
i = p - 1                             // highest index into the low side
for j = p to r - 1                    // process each element other than the pivot
    if A[j] ≤ x                       // does this element belong on the low side?
        i = i + 1                     // index of a new slot in the low side
        exchange A[i] with A[j]       // put this element there     swap, the blue part i moved one to the right
exchange A[i + 1] with A[r]           // pivot goes just to the right of the low side
return i + 1                          // new index of the pivot
```



Quicksort runtime

Depends on how partitioning divides the input during recursion.

Quicksort runtime

Depends on how partitioning divides the input during recursion.

Two extremes of recursive call sizes:

- Completely unbalanced: 0 and n - 1
- Perfectly balanced: $\lceil (n-1)/2 \rceil$ and $\lfloor (n-1)/2 \rfloor$

Quicksort runtime

Depends on how partitioning divides the input during recursion.

Two extremes of recursive call sizes:

- Completely unbalanced: 0 and n - 1 if we are "unlucky" and all ways choose the biggest or smallest number as our pivot point
- Perfectly balanced: $\lceil (n-1)/2 \rceil$ and $\lfloor (n-1)/2 \rfloor$ divides the array roughly in the middel
- If all partitions are perfectly balanced: **O(n log n)** (approximately the same analysis as for Mergesort).
- If all partitions are completely unbalanced:
  $O(n+(n-1)+(n-2)+\cdots+2+1) = \mathbf{O(n^2)}$.

This represents the **best-case** and **worst-case** scenarios for Quicksort.

Quicksort runtime

- In practice, **O(n log n)** for almost all inputs.
- **However, sorted input leads to $\Theta(n^2)$ complexity** with the above choice of pivot element **x** (so this choice should not be used in practice).
- Suggestions for more robust choice of partition element x: either as the middle element, as the median of several elements, as a random element, or as the median of several randomly chosen elements.
- Quicksort is in place: does not use more space than the input array.
- Code is very efficient in practice. A well-implemented Quicksort is often the best all-round sorting algorithm (and chosen in many libraries, e.g. Java and C++/STL).