# MERGE & QUICK SORT

# Two classic sorting algorithms: mergesort and quicksort

Critical components in the world's computational infrastructure.

- Quicksort honored as one of top 10 algorithms of $20^{th}$ century in science and engineering.

Mergesort.

Quicksort.

# QUICK SORT

# HIGH LEVEL BUILD UP
# SOME INTUITION

# High level build up some intuition

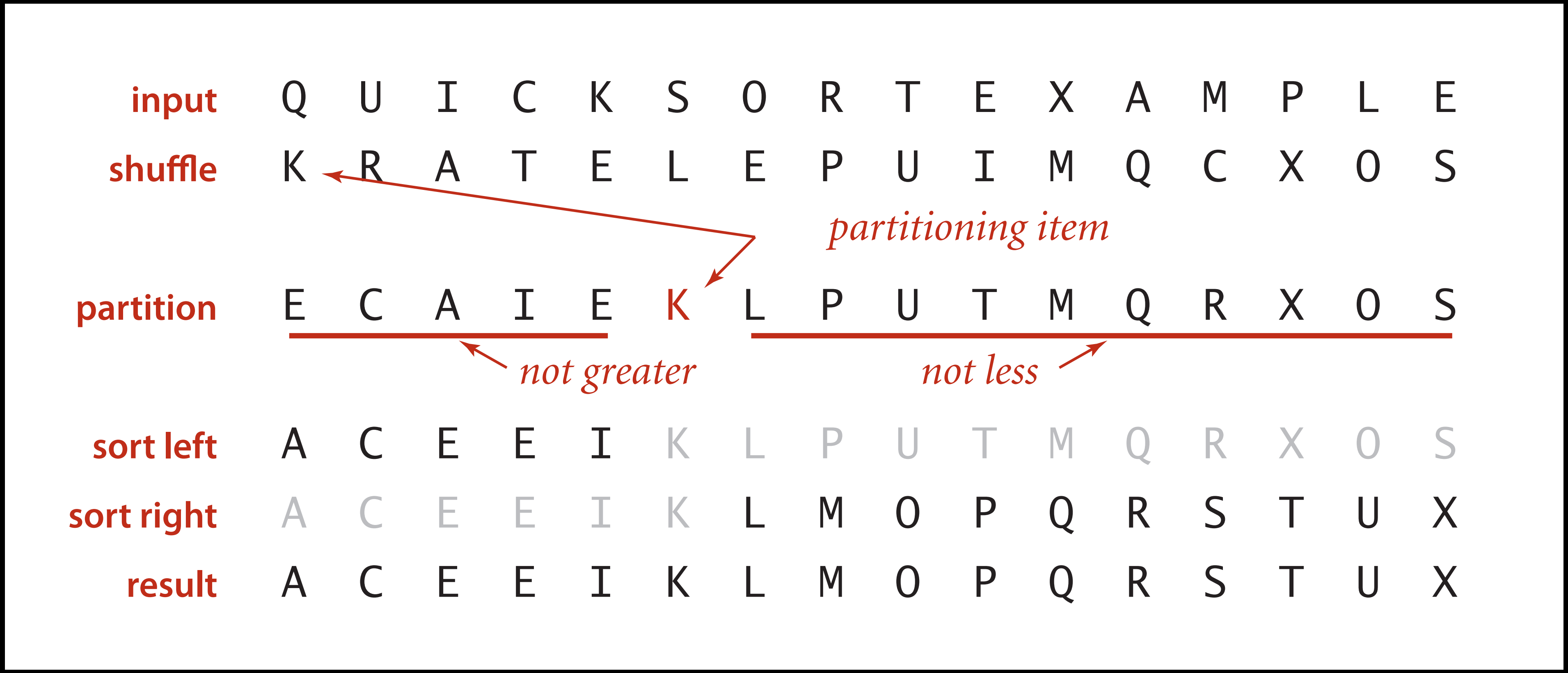
What if we took each element one a time:

- Found it’s place in the array.
- As searching for it’s place in the array moved all stuff less than it to the left.

# Quicksort

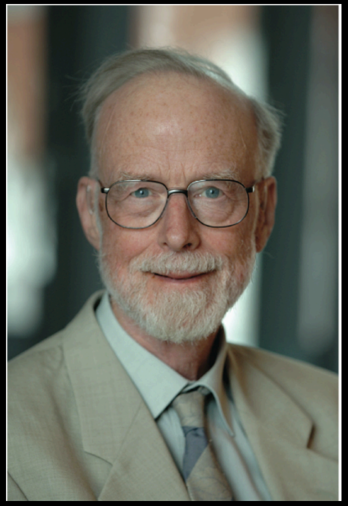
Basic plan.

- Shuffle the array.
- Partition so that, for some `j`
  - entry `a[j]` is in place
  - no larger entry to the left of `j`
  - no smaller entry to the right of `j`
- Sort each subarray recursively.



# Tony Hoare

- Invented quicksort to translate Russian into English.
- Learned Algol 60 (and recursion).
- Implemented quicksort.



**Tony Hoare**
**1980 Turing Award**

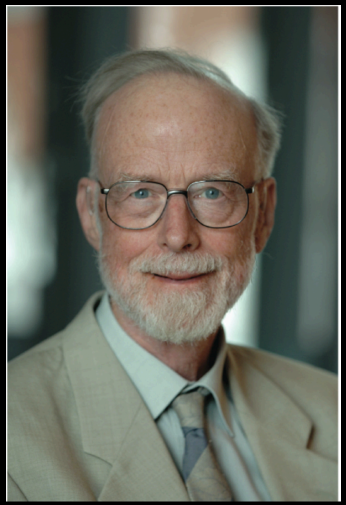
**Algorithms**

ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure   quicksort (A,M,N);  value M,N;
              array A;  integer M,N;
comment  Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin       integer I,J;
            if M < N then begin partition (A,M,N,I,J);
                                quicksort (A,M,J);
                                quicksort (A, I, N)
                          end
end         quicksort
```

# Tony Hoare

- Invented quicksort to translate Russian into English.
- Learned Algol 60 (and recursion).
- Implemented quicksort.



*" There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. "*

*" I call it my billion-dollar mistake. It was the invention of the null reference in 1965… This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. "*

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
  - (Things to left are smaller)
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
  - (Things to right are larger)
- Exchange `a[i]` with `a[j]`.
  - One nothing left smaller and nothing is bigger and they are not the same it means:
  - (Something of the left is larger that something on right so swap)

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| ↑ lo | ↑ i | | | | | | | | | | | | | | ↑ j |

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

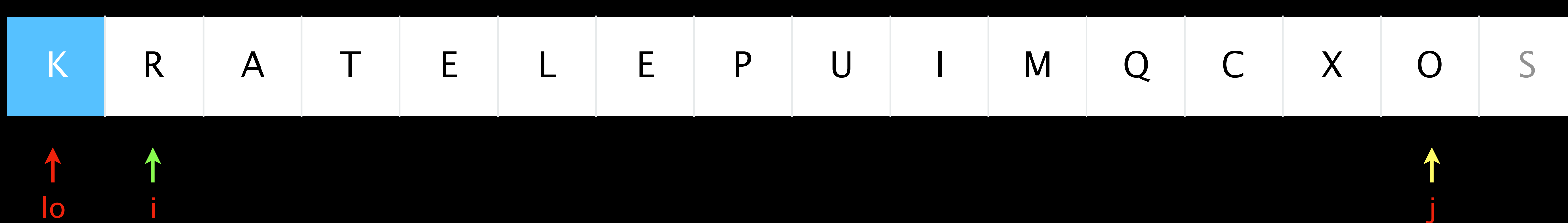
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| ↑ lo | ↑ i | | | | | | | | | | | | | | ↑ j |

**stop i scan because a[i] >= a[lo]**

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

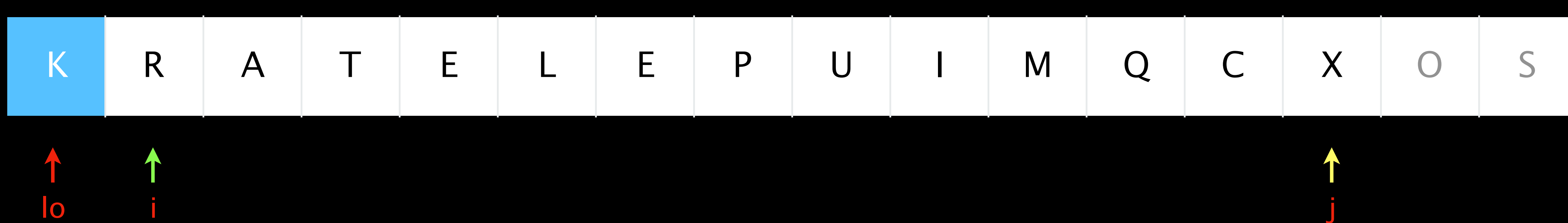
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

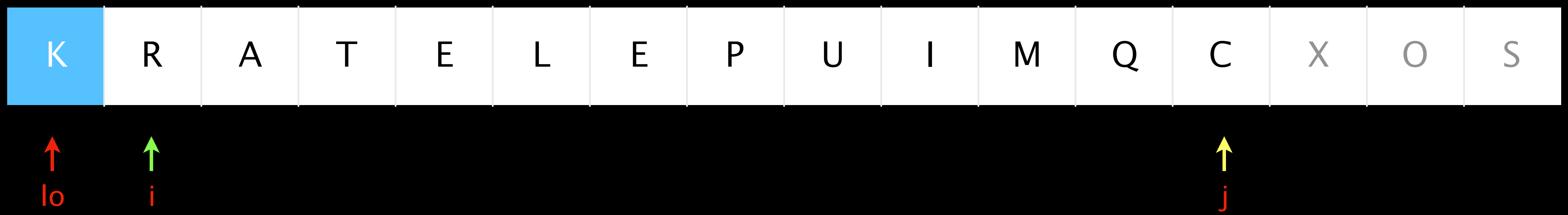
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

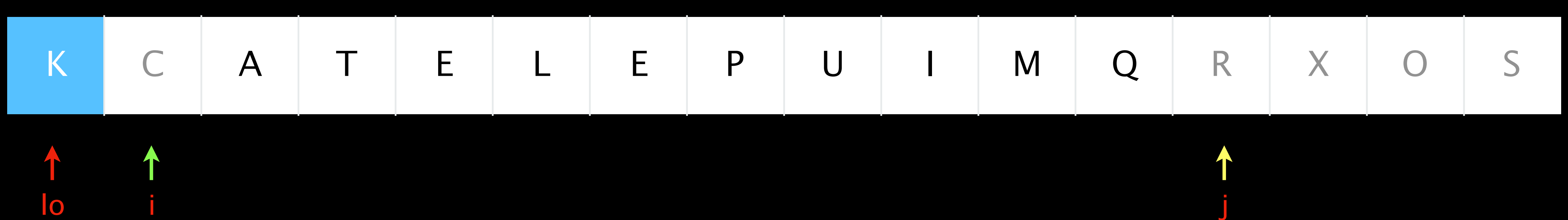



**stop j scan and exchange a[i] with a[j]**

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

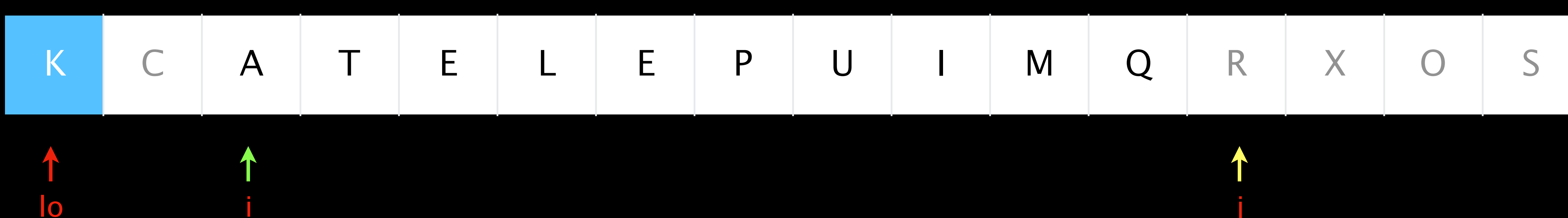
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

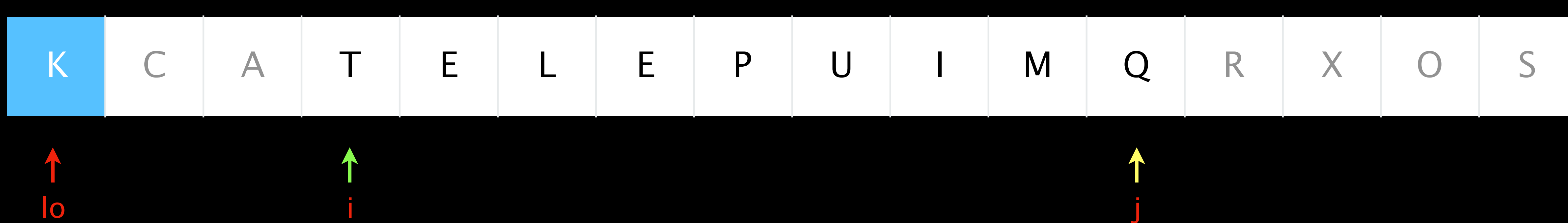
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | C | A | T | E | L | E | P | U | I | M | Q | R | X | O | S |
| lo | | | i | | | | | | | | | j | | | |

**stop i scan because a[i] >= a[lo]**

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

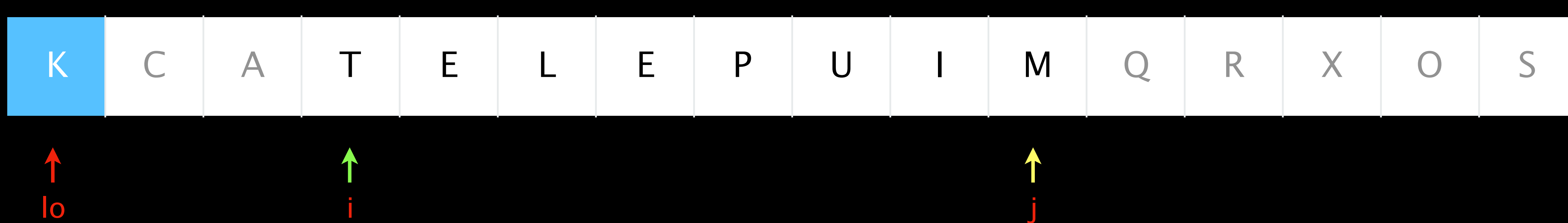
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

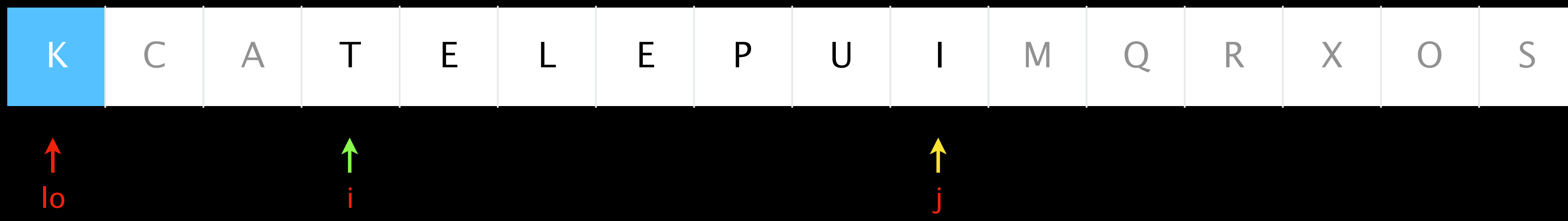
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

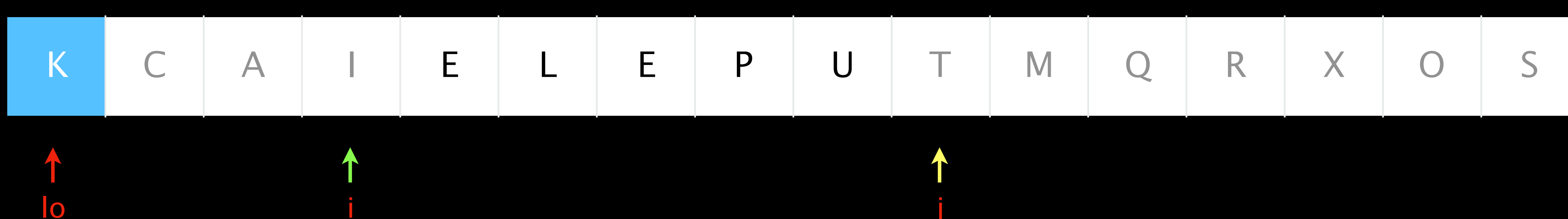



**stop j scan and exchange a[i] with a[j]**

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

| K | C | A | I | E | L | E | P | U | T | M | Q | R | X | O | S |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lo | | | | i | | | | | j | | | | | | |

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

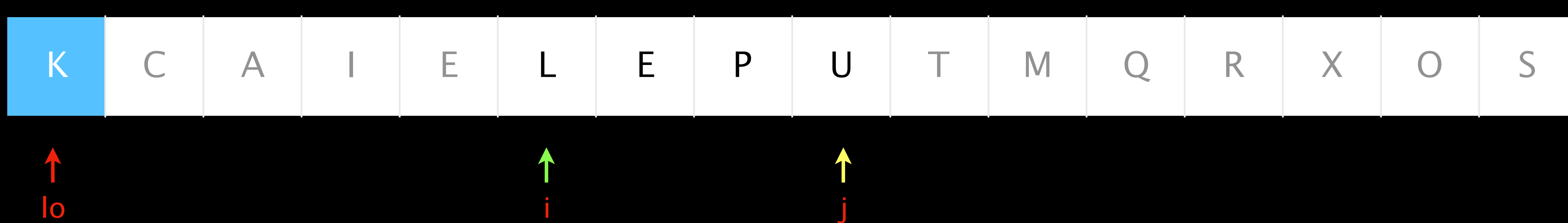
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | C | A | I | E | L | E | P | U | T | M | Q | R | X | O | S |
| ↑ lo | | | | | ↑ i | | | | ↑ j | | | | | | |

**stop i scan because a[i] >= a[lo]**

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

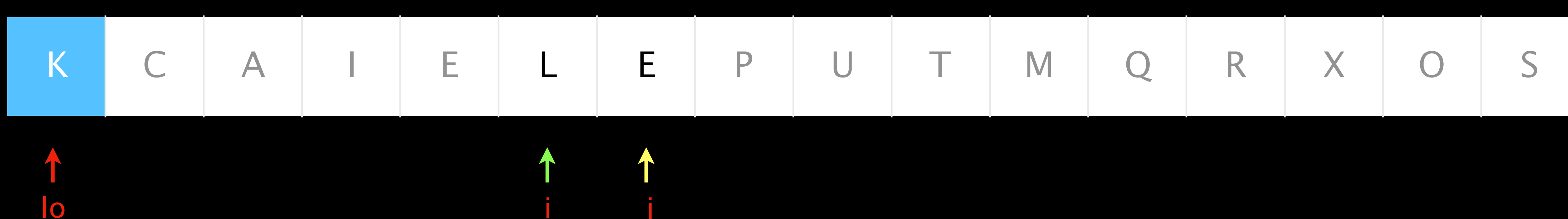
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | C | A | I | E | L | E | P | U | T | M | Q | R | X | O | S |
| lo | | | | | i | | j | | | | | | | | |

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

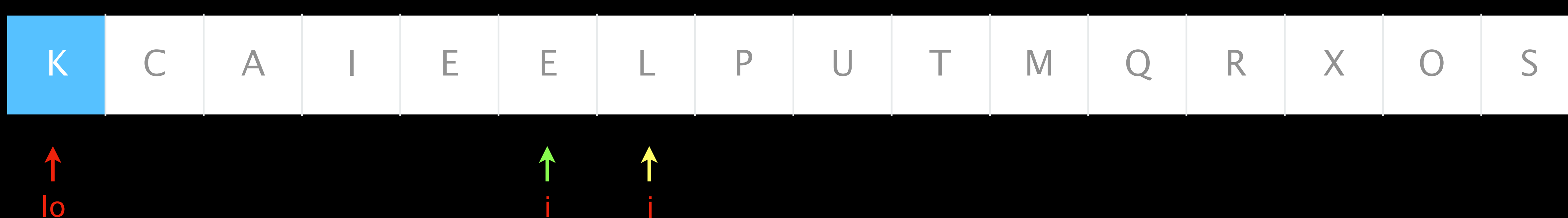



**stop j scan and exchange a[i] with a[j]**

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

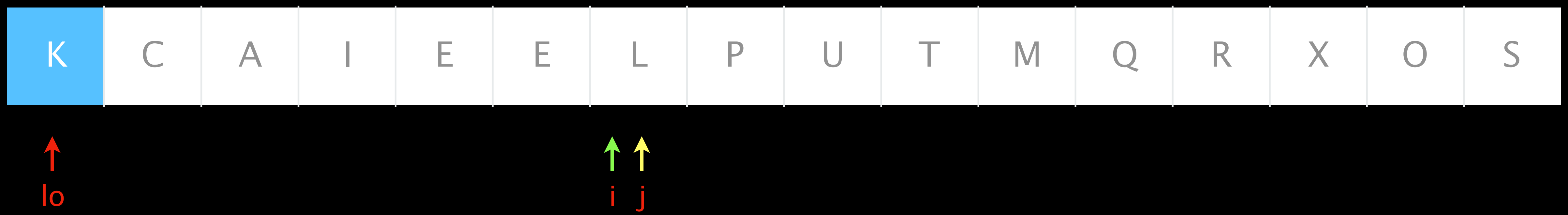
- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.



# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.




**stop i scan because a[i] >= a[lo]**

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| K | C | A | I | E | E | L | P | U | T | M | Q | R | X | O | S |
| lo | | | | | j | i | | | | | | | | | |

**stop j scan because a[j] <= a[lo]**

# Quicksort partitioning demo

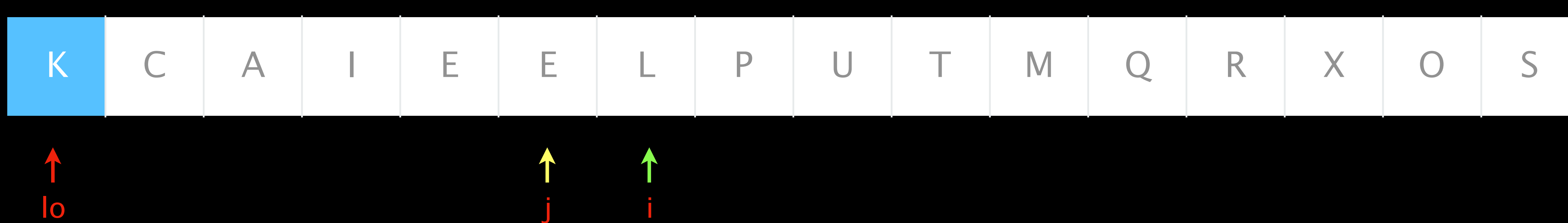
Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

When pointers cross.

- Exchange `a[lo]` with `a[j]`.

**Intuition: everything to left of J**




**pointers cross: exchange a[lo] with a[j]**

# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

When pointers cross.

- Exchange `a[lo]` with `a[j]`.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| lo | | | | | j | | | | | | | | | | hi |

**partitioned!**

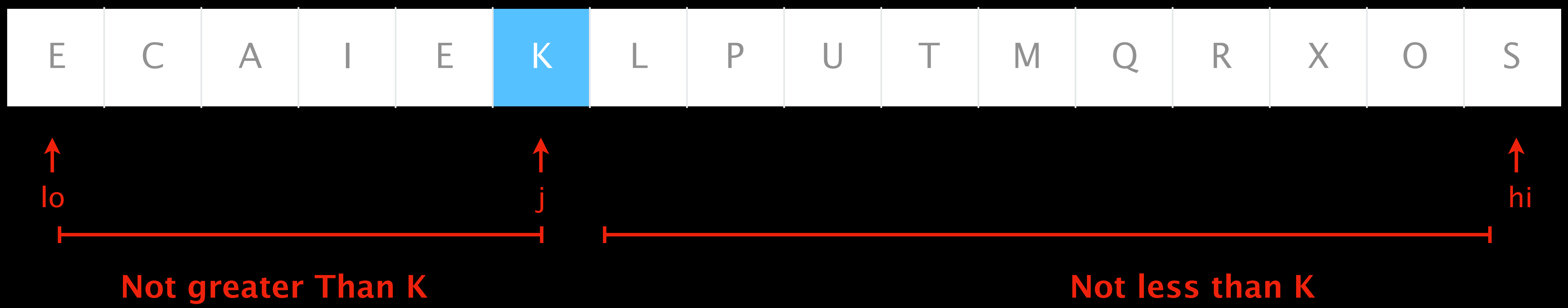
# Quicksort partitioning demo

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

When pointers cross.

- Exchange `a[lo]` with `a[j]`.




partitioned!

# HOW WOULD WE IMPLEMENT THIS?

What are the invariants?

# Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (true)
   {
      while (less(a[++i], a[lo]))
         if (i == hi) break;

      while (less(a[lo], a[--j]))
         if (j == lo) break;

      if (i >= j) break;
      exch(a, i, j);
   }

   exch(a, lo, j);
   return j;
}
```

find item on left to swap

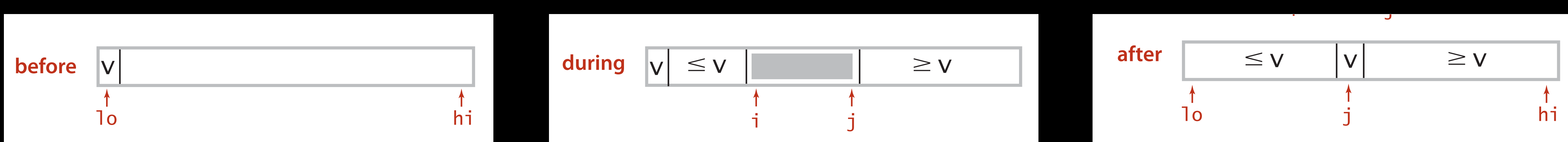
find item on right to swap

check if pointers cross

swap

swap with partitioning item

return index of item now known to be in place



# Quicksort: Java implementation

```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

   public static void sort(Comparable[] a)
   {
      StdRandom.shuffle(a);
      sort(a, 0, a.length - 1);
   }

   private static void sort(Comparable[] a, int lo, int hi)
   {
      if (hi <= lo) return;
      int j = partition(a, lo, hi);
      sort(a, lo, j-1);
      sort(a, j+1, hi);
  }
}
```

shuffle needed for
performance guarantee
(stay tuned)

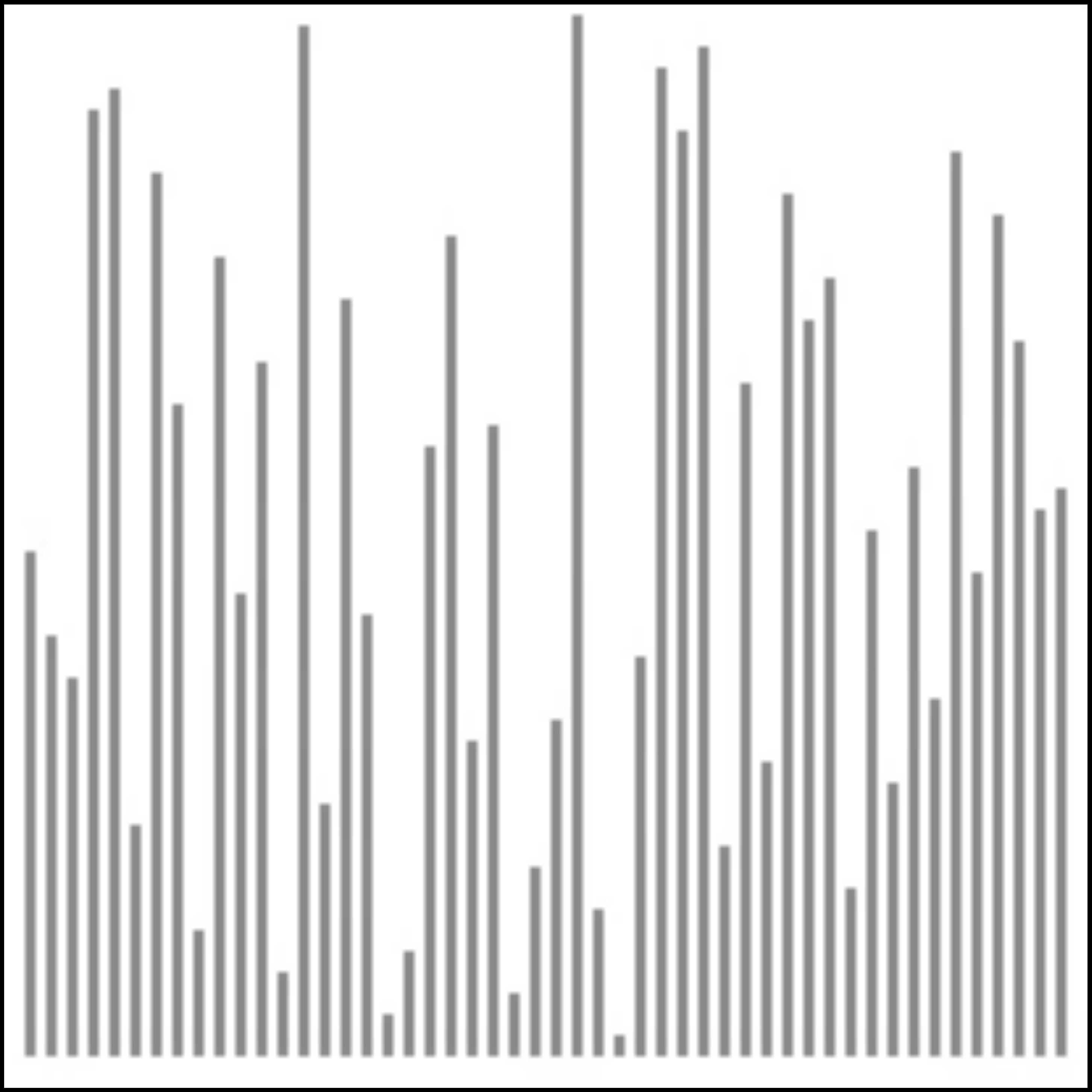
# Quicksort trace

| | lo | j | hi | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **initial values** | | | | Q | U | I | C | K | S | O | R | T | E | X | A | M | P | L | E |
| **random shuffle** | | | | K | R | A | T | E | L | E | P | U | I | M | Q | C | X | O | S |
| | 0 | 5 | 15 | E | C | A | I | E | K | L | P | U | T | M | Q | R | X | O | S |
| | 0 | 3 | 4 | E | C | A | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| | 0 | 2 | 2 | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| | 0 | 0 | 1 | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| | 1 | | 1 | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| | 4 | | 4 | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| | 6 | 6 | 15 | A | C | E | E | I | K | L | P | U | T | M | Q | R | X | O | S |
| | 7 | 9 | 15 | A | C | E | E | I | K | L | M | O | P | T | Q | R | X | U | S |
| | 7 | 7 | 8 | A | C | E | E | I | K | L | M | O | P | T | Q | R | X | U | S |
| | 8 | | 8 | A | C | E | E | I | K | L | M | O | P | T | Q | R | X | U | S |
| | 10 | 13 | 15 | A | C | E | E | I | K | L | M | O | P | S | Q | R | T | U | X |
| | 10 | 12 | 12 | A | C | E | E | I | K | L | M | O | P | R | Q | S | T | U | X |
| | 10 | 11 | 11 | A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |
| | 10 | | 10 | A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |
| | 14 | 14 | 15 | A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |
| | 15 | | 15 | A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |
| **result** | | | | A | C | E | E | I | K | L | M | O | P | Q | R | S | T | U | X |

*no partition for subarrays of size 1*

**Quicksort trace (array contents after each partition)**

# Quicksort animation




http://www.sorting-algorithms.com/quick-sort

# Quicksort: implementation details

Partitioning in-place. Using an extra array makes partitioning easier (and stable), but is not worth the cost.

Preserving randomness. Shuffling is needed for performance guarantee.

Equivalent alternative. Pick a random partitioning item in each subarray.

# Quicksort: empirical analysis

Running time estimates:

- Home PC executes $10^8$ compares/second.
- Supercomputer executes $10^{12}$ compares/second.

| | insertion sort ($N^2$) | | | mergesort (N log N) | | | quicksort (N log N) | | |
|---|---|---|---|---|---|---|---|---|---|
| computer | thousand | million | billion | thousand | million | billion | thousand | million | billion |
| home | instant | 2.8 hours | 317 years | instant | 1 second | 18 min | instant | 0.6 sec | 12 min |
| super | instant | 1 second | 1 week | instant | instant | instant | instant | instant | instant |

Lesson 1. Good algorithms are better than supercomputers.

Lesson 2. Great algorithms are better than good ones.

# Quicksort: best-case analysis

We will show this during our recurrence Master method lecture

Best case. Number of compares is $\sim N \lg N$.

| | | | a[ ] | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lo | j | hi | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| initial values | | | H | A | C | B | F | E | G | D | L | I | K | J | N | M | O |
| random shuffle | | | H | A | C | B | F | E | G | D | L | I | K | J | N | M | O |
| 0 | 7 | 14 | D | A | C | B | F | E | G | H | L | I | K | J | N | M | O |
| 0 | 3 | 6 | B | A | C | D | F | E | G | H | L | I | K | J | N | M | O |
| 0 | 1 | 2 | A | B | C | D | F | E | G | H | L | I | K | J | N | M | O |
| 0 | | 0 | A | B | C | D | F | E | G | H | L | I | K | J | N | M | O |
| 2 | | 2 | A | B | C | D | F | E | G | H | L | I | K | J | N | M | O |
| 4 | 5 | 6 | A | B | C | D | E | F | G | H | L | I | K | J | N | M | O |
| 4 | | 4 | A | B | C | D | E | F | G | H | L | I | K | J | N | M | O |
| 6 | | 6 | A | B | C | D | E | F | G | H | L | I | K | J | N | M | O |
| 8 | 11 | 14 | A | B | C | D | E | F | G | H | J | I | K | L | N | M | O |
| 8 | 9 | 10 | A | B | C | D | E | F | G | H | I | J | K | L | N | M | O |
| 8 | | 8 | A | B | C | D | E | F | G | H | I | J | K | L | N | M | O |
| 10 | | 10 | A | B | C | D | E | F | G | H | I | J | K | L | N | M | O |
| 12 | 13 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 12 | | 12 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 14 | | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| | | | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |

# Quicksort: worst-case analysis

Worst case. Number of compares is $\sim \frac{1}{2} N^2$ .

| | | | a[ ] | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lo | j | hi | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| initial values | | | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| random shuffle | | | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 0 | 0 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 1 | 1 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 2 | 2 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 3 | 3 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 4 | 4 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | 5 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 6 | 6 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 7 | 7 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 8 | 8 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 9 | 9 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 10 | 10 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 11 | 11 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 12 | 12 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 13 | 13 | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 14 | | 14 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| | | | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |

# IS IT STABLE?

# Quicksort properties

Proposition. Quicksort is not stable.

Pf. [ by counterexample ]

| 0 | 1 | 2 | 3 |
|---|---|---|---|
| $B_1$ | $C_1$ | $C_2$ | $A_1$ |
| $B_1$ | $C_1$ | $C_2$ | $A_1$ |
| $B_1$ | $A_1$ | $C_2$ | $C_1$ |
| $A_1$ | $B_1$ | $C_2$ | $C_1$ |

**C1 and C2 are not out of order**

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

When pointers cross.

- Exchange `a[lo]` with `a[j]`.

# IMPROVEMENTS?

# Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo + CUTOFF - 1)
   {
      Insertion.sort(a, lo, hi);
      return;
   }
   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

# Quicksort: practical improvements

## Median of sample.

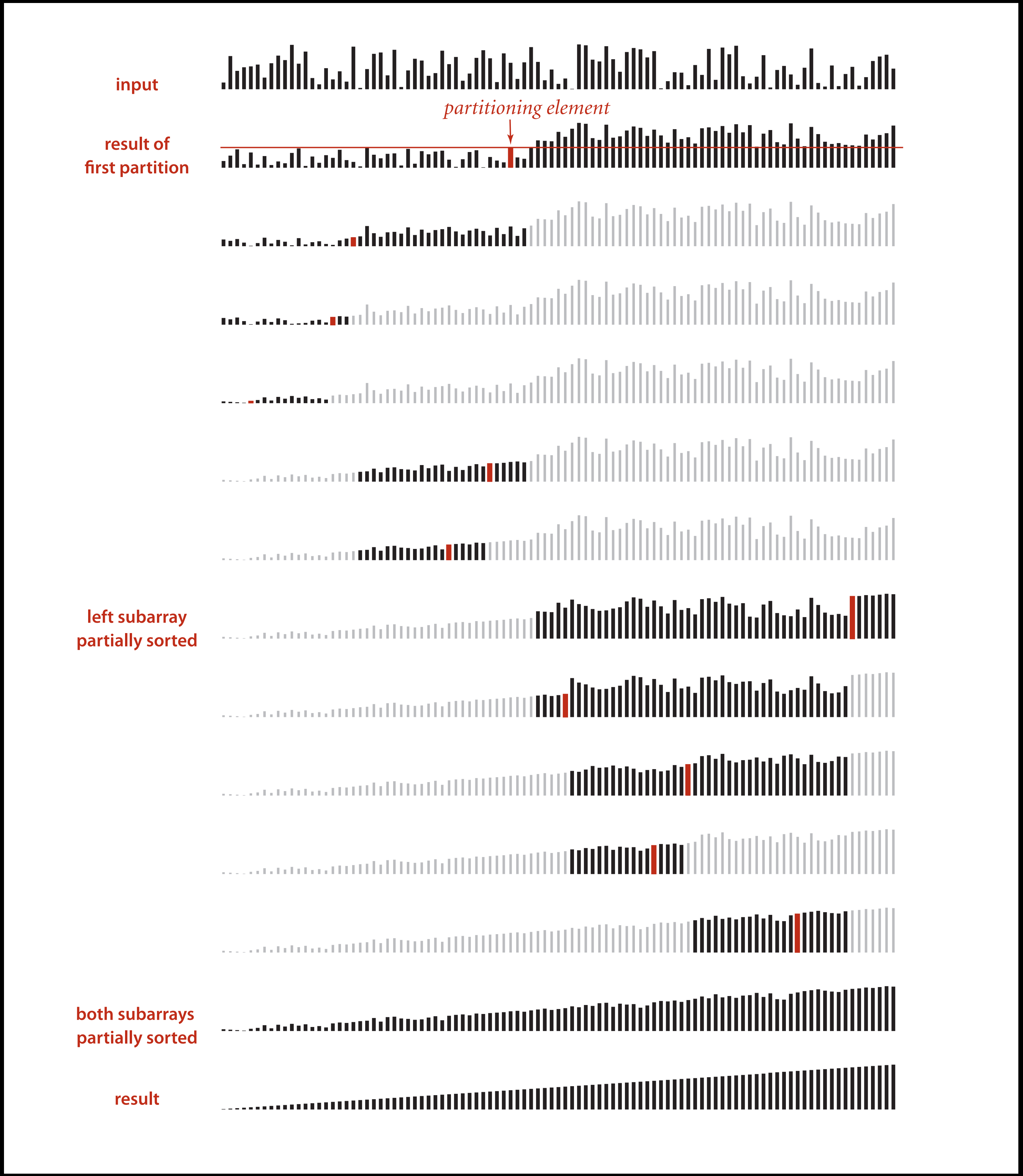
- Best choice of pivot item = median.
- Estimate true median by taking median of sample.
- Median-of-3 (random) items.

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;

   int median = medianOf3(a, lo, lo + (hi - lo)/2, hi);
   swap(a, lo, median);

   int j = partition(a, lo, hi);
   sort(a, lo, j-1);
   sort(a, j+1, hi);
}
```

# Quicksort with median-of-3 and cutoff to insertion sort: visualization



# QUICK SELECT

# Selection

Goal. Given an array of $N$ items, find the $k^{th}$ smallest item.

Ex. Min $(k = 0)$, max $(k = N - 1)$, median $(k = N/2)$.

Applications.

- Order statistics.
- Find the "top $k$."

Use theory as a guide.

- Easy $N \log N$ upper bound. How?
- Easy $N$ upper bound for $k = 1, 2, 3$. How?
- Easy $N$ lower bound. Why?

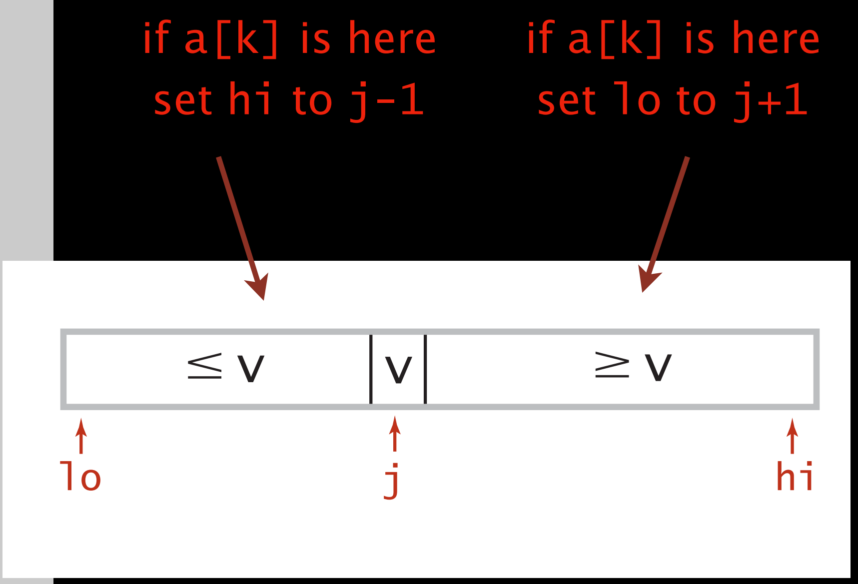
# Quick-select

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

```
public static Comparable select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
       int j = partition(a, lo, hi);
       if      (j < k) lo = j + 1;
       else if (j > k) hi = j - 1;
       else            return a[k];
    }
    return a[k];
}
```



# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

select element of rank k = 5

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 21 | 28 | 65 | 39 | 59 | 56 | 22 | 95 | 12 | 90 | 53 | 32 | 77 | 33 |

k = 5

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partition on leftmost entry

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 | 21 | 28 | 65 | 39 | 59 | 56 | 22 | 95 | 12 | 90 | 53 | 32 | 77 | 33 |

k = 5

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partitioned array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 22 | 21 | 28 | 33 | 39 | 32 | 12 | 50 | 95 | 56 | 90 | 53 | 59 | 77 | 65 |

k = 5

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

can safely ignore right subarray

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 22 | 21 | 28 | 33 | 39 | 32 | 12 | 50 | 95 | 56 | 90 | 53 | 59 | 77 | 65 |

k = 5

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partition on leftmost entry

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 22 | 21 | 28 | 33 | 39 | 32 | 12 | 50 | 95 | 56 | 90 | 53 | 59 | 77 | 65 |

k = 5

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

**partitioned array**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 21 | 22 | 33 | 39 | 32 | 28 | 50 | 95 | 56 | 90 | 53 | 59 | 77 | 65 |

**k = 5**

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

can safely ignore left subarray

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 21 | 22 | 33 | 39 | 32 | 28 | 50 | 95 | 56 | 90 | 53 | 59 | 77 | 65 |

k = 5

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partition on leftmost entry

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 21 | 22 | 33 | 39 | 32 | 28 | 50 | 95 | 56 | 90 | 53 | 59 | 77 | 65 |

k = 5

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

partitioned array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 21 | 22 | 32 | 28 | 33 | 39 | 50 | 95 | 56 | 90 | 53 | 59 | 77 | 65 |

k = 5

# Quick-select demo

Partition array so that:

- Entry `a[j]` is in place.
- No larger entry to the left of `j`.
- No smaller entry to the right of `j`.

Repeat in one subarray, depending on `j`; finished when `j` equals `k`.

stop: partitioning item is at index k

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 21 | 22 | 32 | 28 | 33 | 39 | 50 | 95 | 56 | 90 | 53 | 59 | 77 | 65 |

k = 5

WHAT ABOUT THE DUPLICATE KEY PROBLEM FROM EARLIER

# Duplicate keys

Often, purpose of sort is to bring items with equal keys together.

- Sort population by age.
- Remove duplicates from mailing list.
- Sort job applicants by college attended.

Typical characteristics of such applications.

- Huge array.
- Small number of key values.

```
Chicago 09:25:52
Chicago 09:03:13
Chicago 09:21:05
Chicago 09:19:46
Chicago 09:19:32
Chicago 09:00:00
Chicago 09:35:21
Chicago 09:00:59
Houston 09:01:10
Houston 09:00:13
Phoenix 09:37:44
Phoenix 09:00:03
Phoenix 09:14:25
Seattle 09:10:25
Seattle 09:36:14
Seattle 09:22:43
Seattle 09:10:11
Seattle 09:22:54
```

key

Quicksort with duplicate keys. Algorithm can go quadratic unless partitioning stops on equal keys!

S T O P O N E Q U A L K E Y S

swap

if we don't stop on equal keys

if we stop on equal keys

Caveat. Some textbook (and commercial) implementations go quadratic when many duplicate keys.

# Don't stop scan on equal keys

Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] <= a[lo])`.
- Scan `j` from right to left so long as `(a[j] => a[lo])`.
- Exchange `a[i]` with `a[j]`.

When pointers cross.

- Exchange `a[lo]` with `a[j]`.

**What is the result of partitioning the following array?**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| lo | i | | | | | | | | | | | | | | j |

**What is the result of partitioning the following array?**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

**A.**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

**B.**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

**C.**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

**What is the result of partitioning the following array?**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

**A.**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

**B.**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

**C.**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

# What if we stop on equal keys

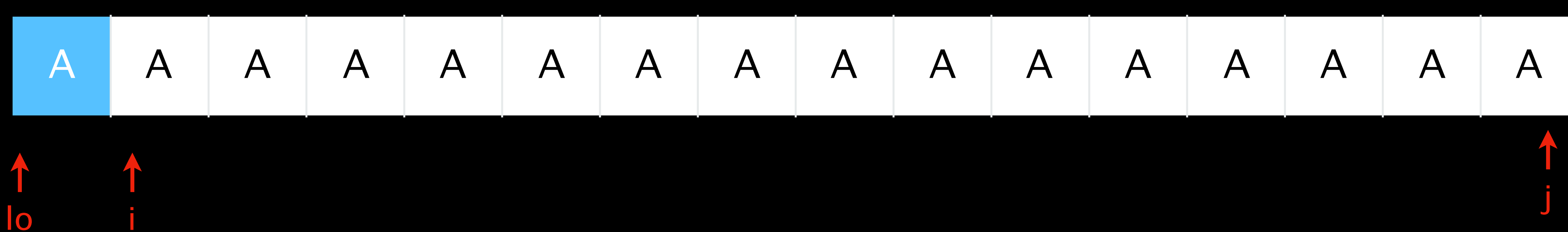
Repeat until `i` and `j` pointers cross.

- Scan `i` from left to right so long as `(a[i] < a[lo])`.
- Scan `j` from right to left so long as `(a[j] > a[lo])`.
- Exchange `a[i]` with `a[j]`.

When pointers cross.

- Exchange `a[lo]` with `a[j]`.

**What is the result of partitioning the following array?**



# Partitioning an array with all equal keys

| | | a[ ] | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| | | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 1 | 15 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 1 | 15 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 2 | 14 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 2 | 14 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 3 | 13 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 3 | 13 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 4 | 12 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 4 | 12 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 5 | 11 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 5 | 11 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 6 | 10 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 6 | 10 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 7 | 9 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| 7 | 9 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| | 8 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |
| | 8 | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A | A |

# Duplicate keys: the problem

Recommended. Stop scans on items equal to the partitioning item.
Consequence. $\sim N \lg N$ compares when all keys equal.

B A A B A B C C B C B        A A A A A A A A A A A

Mistake. Don't stop scans on items equal to the partitioning item.
Consequence. $\sim \frac{1}{2} N^2$ compares when all keys equal.

B A A B A B B B C C C        A A A A A A A A A A A

Desirable. Put all items equal to the partitioning item in place.

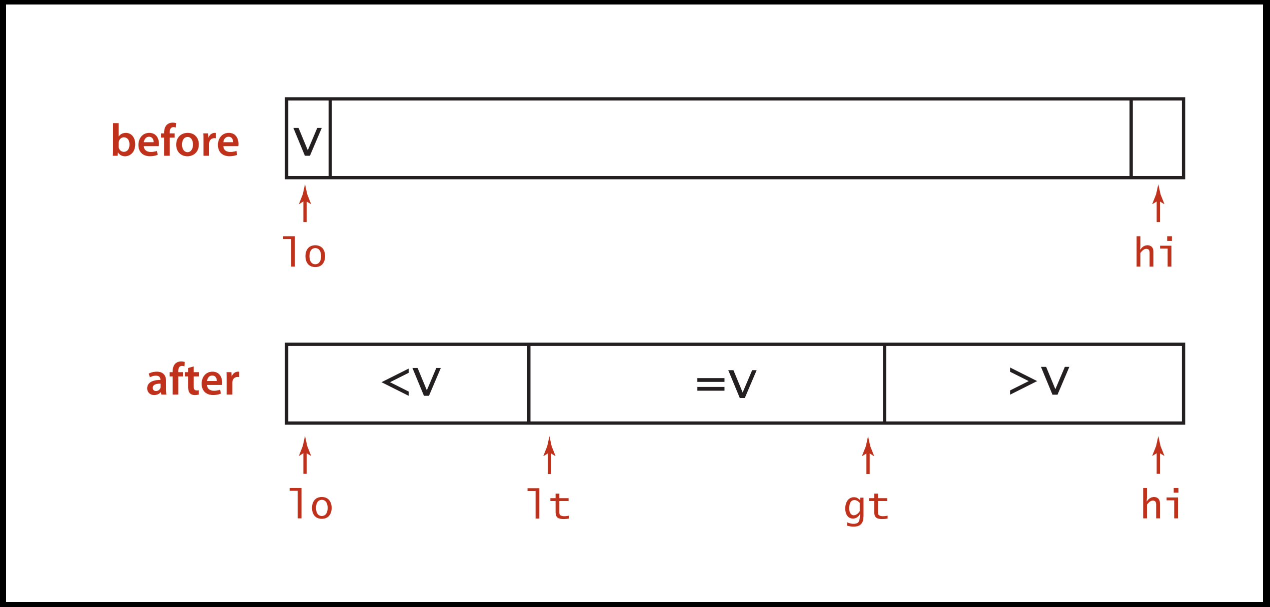
A A A B B B B B C C C        A A A A A A A A A A A

# THREE WAY PARTITIONS

# 3-way partitioning

Goal. Partition array into three parts so that:

- Entries between `lt` and `gt` equal to the partition item.
- No larger entries to left of `lt`.
- No smaller entries to right of `gt`.

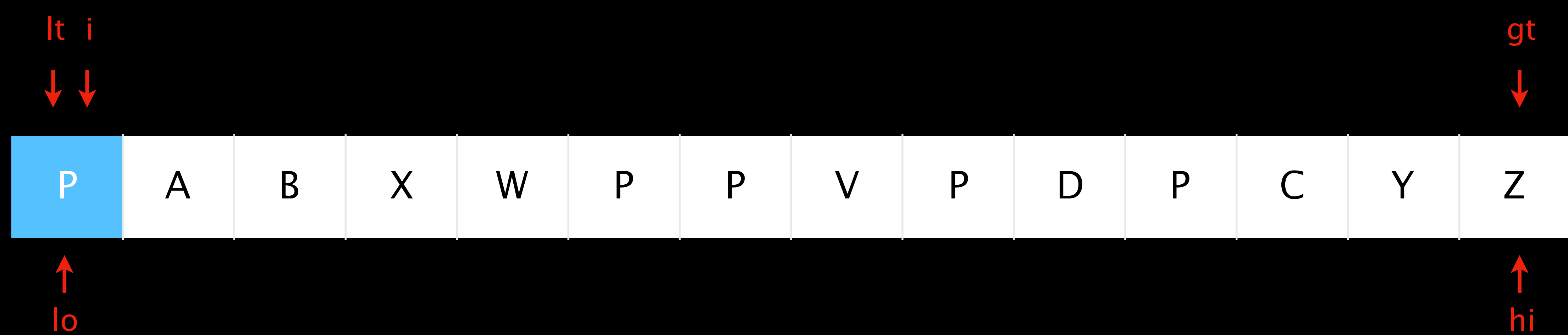


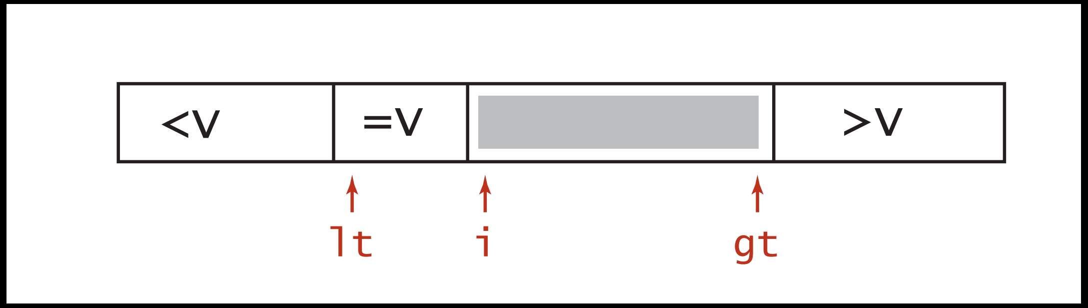

Dutch national [Edsger Dijkstra]

- Conventional wisdom until mid 1990s: not worth doing.
- Now incorporated into C library `qsort()` and Java 6 system sort.

# Dijkstra 3-way partitioning demo

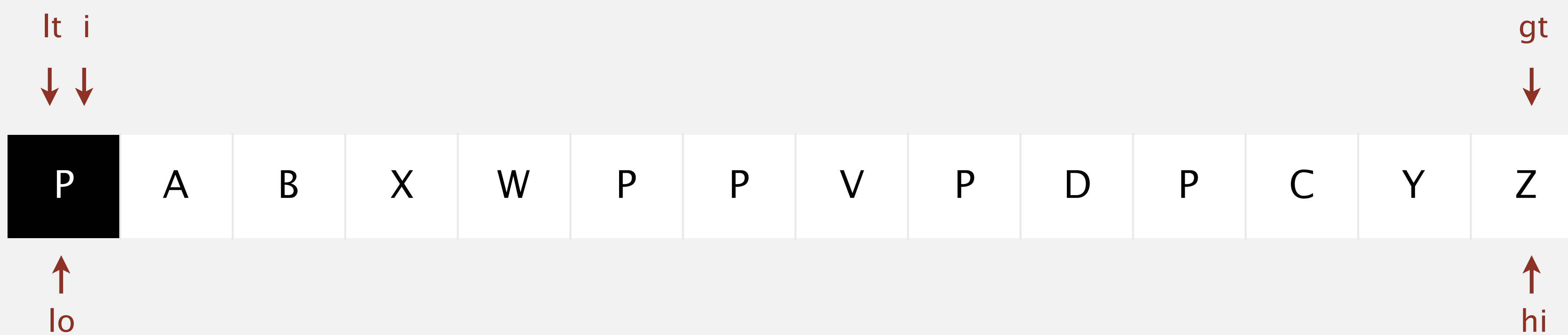
- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and i
  - `(put items small than the pivot to the left of lt)`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(put items greater than the pivot to the right of gt)`
  - `(a[i] == v)`: increment `I`
  - `(make region bigger)`



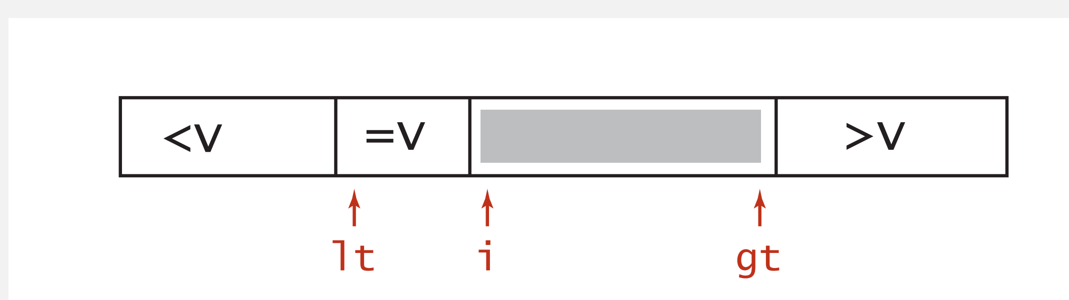



# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

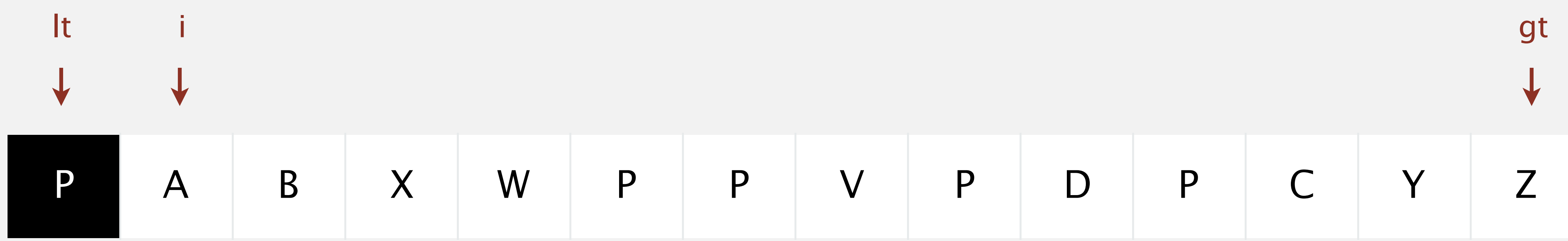



**invariant**

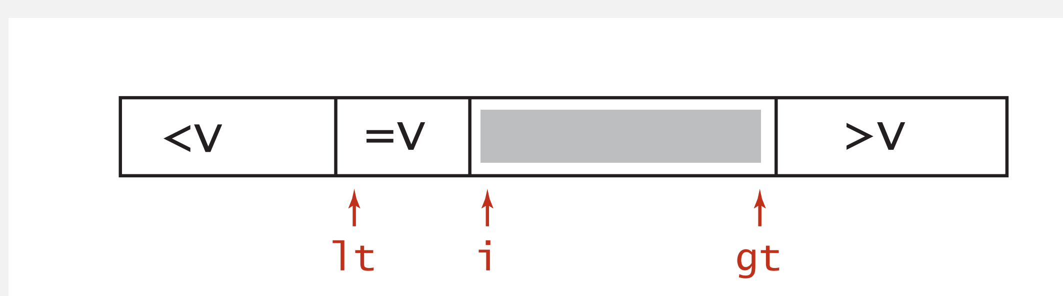


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

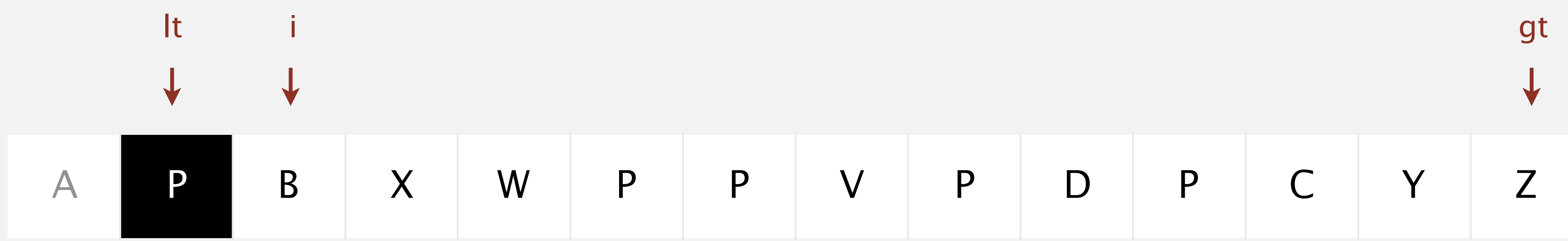



**invariant**

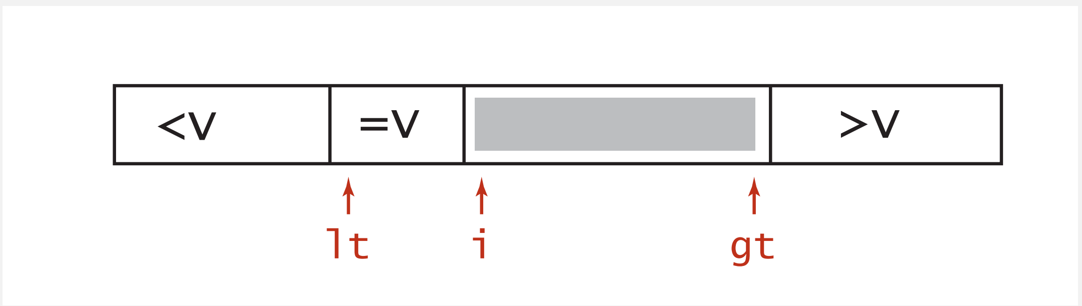


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

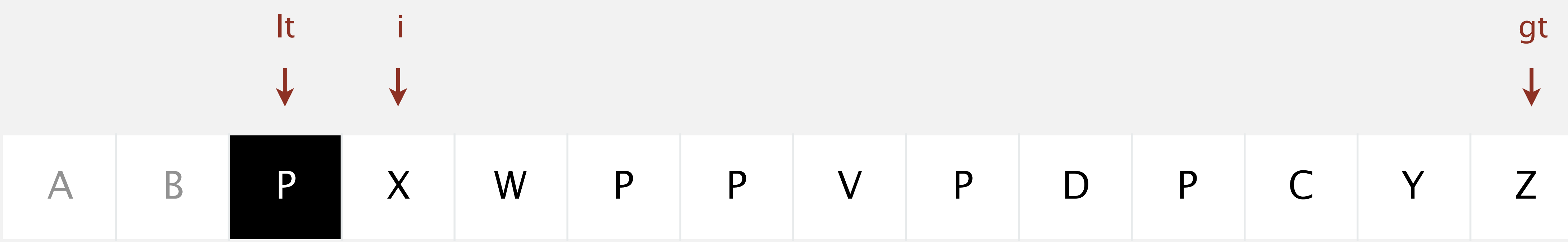



**invariant**

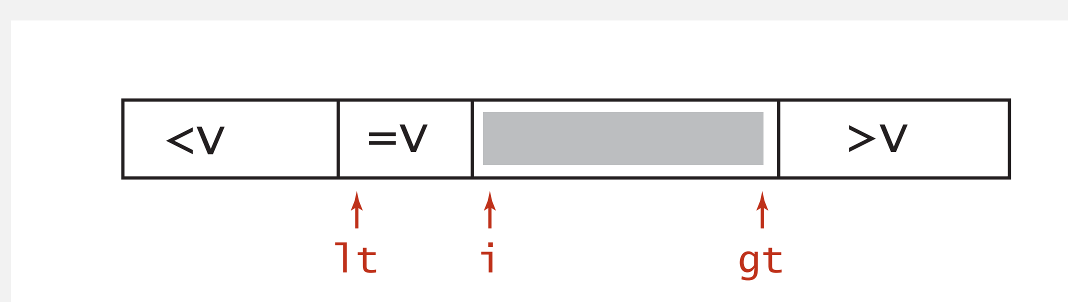


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

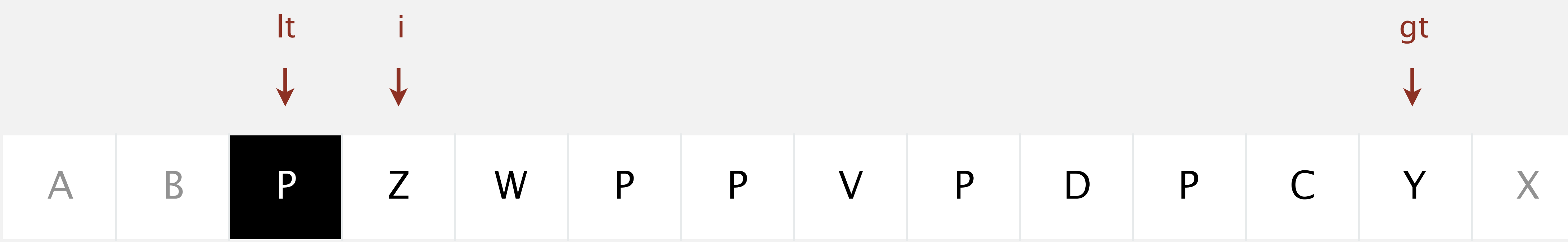



**invariant**

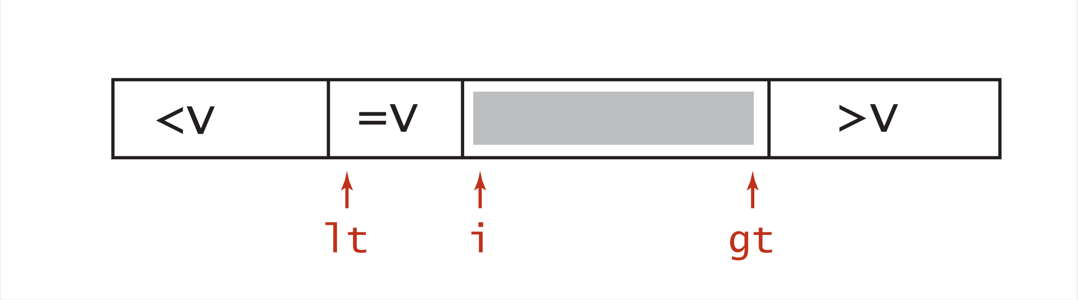


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

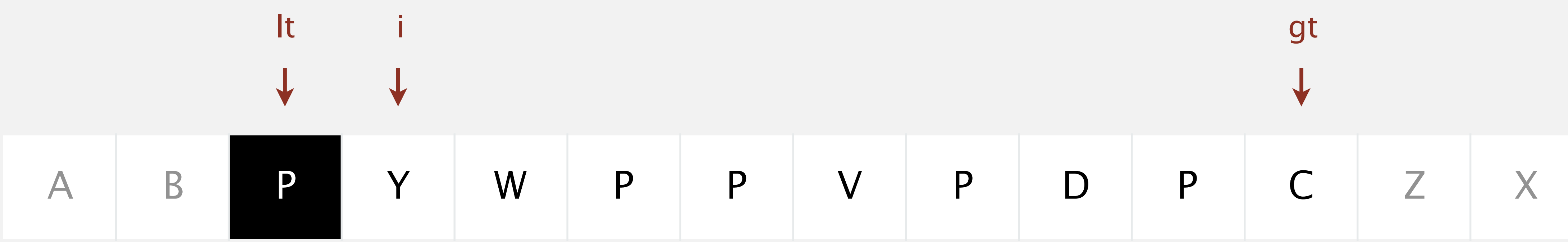



**invariant**

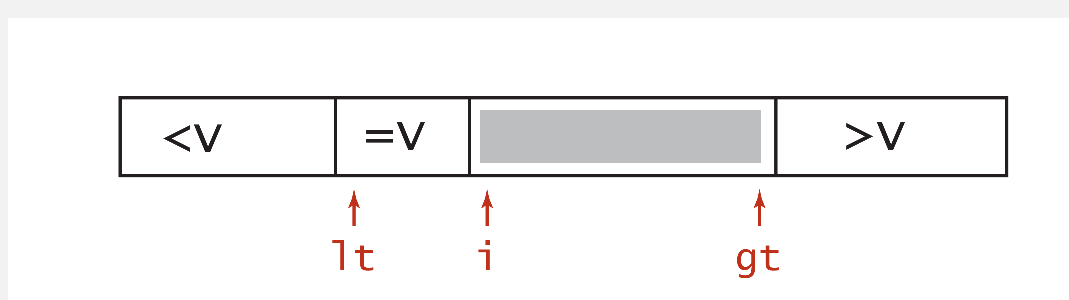


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

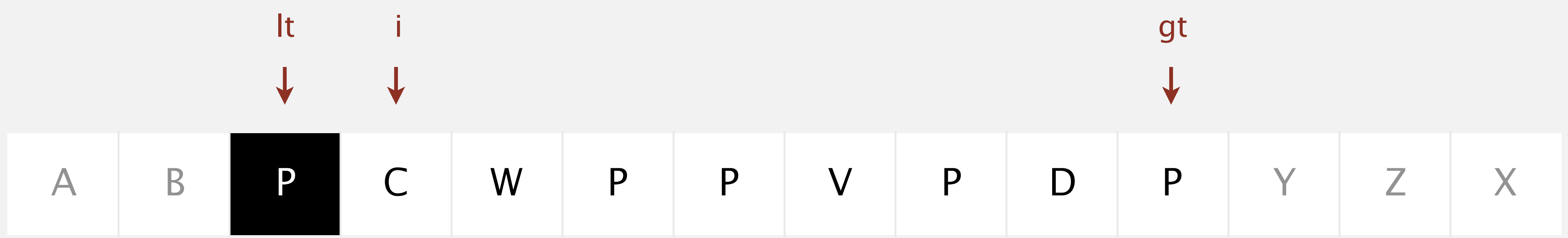


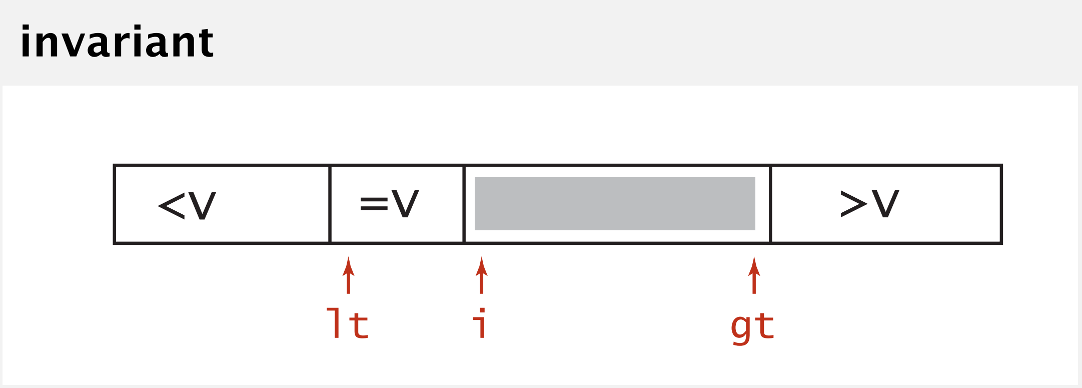

**invariant**



# Dijkstra 3-way partitioning demo

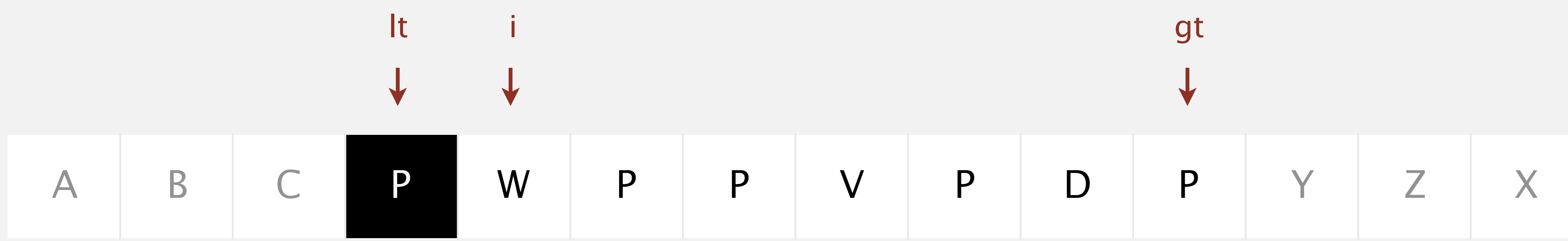
- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i] < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i] > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`



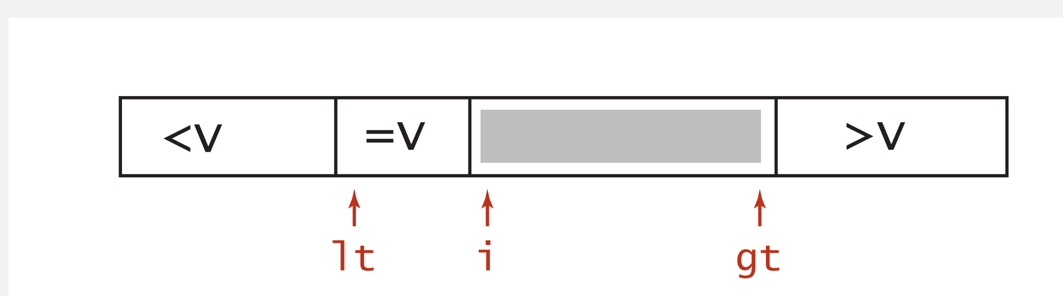



# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

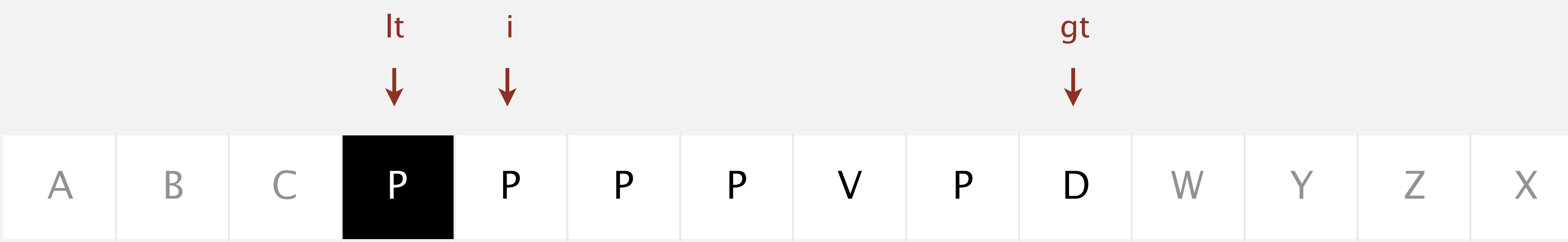



**invariant**

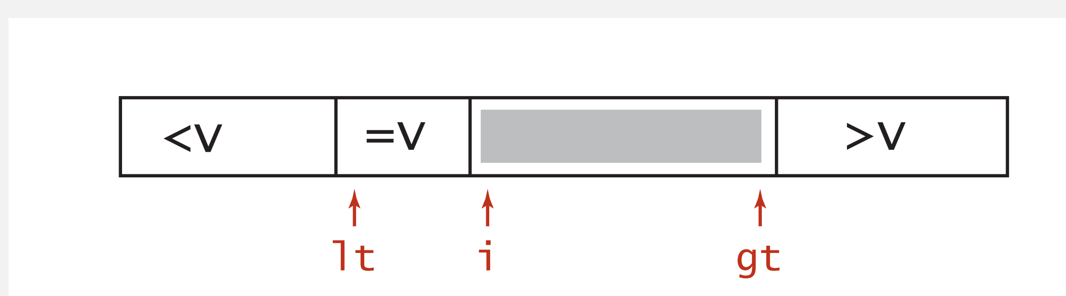


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i] < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i] > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

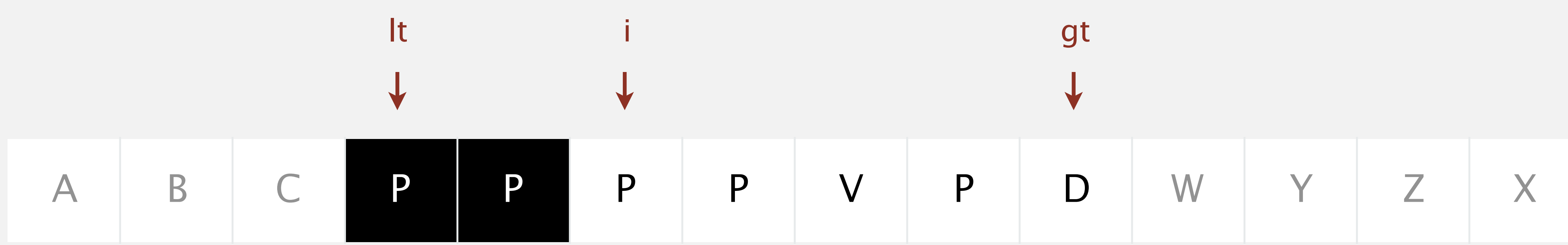



**invariant**

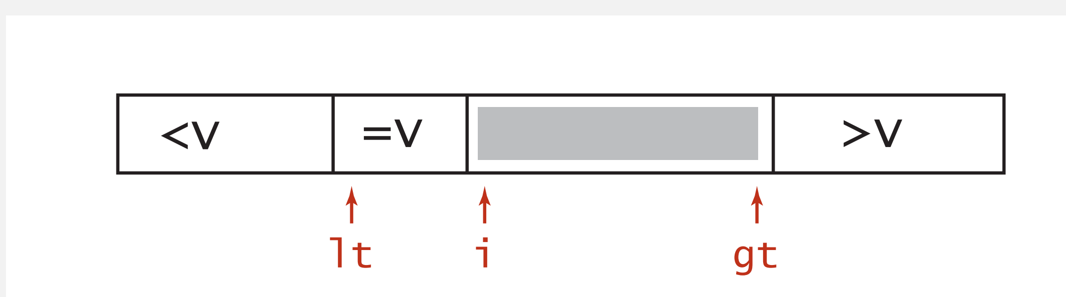


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i] < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i] > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`



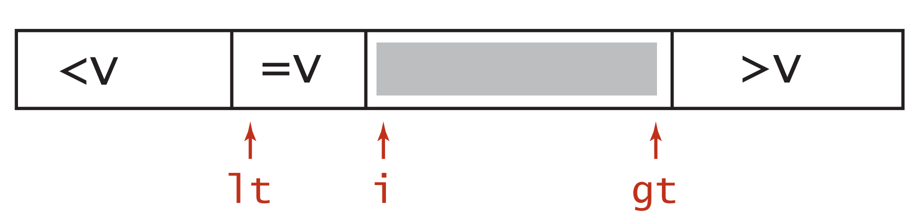

**invariant**



# Dijkstra 3-way partitioning demo

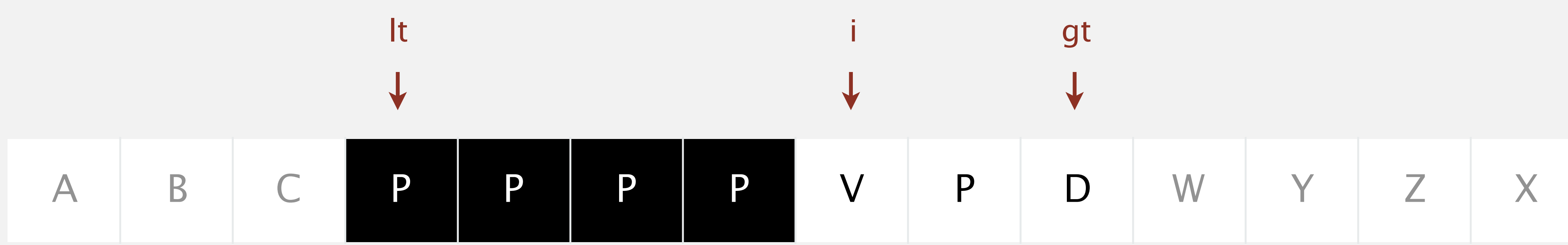
- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`


**invariant**

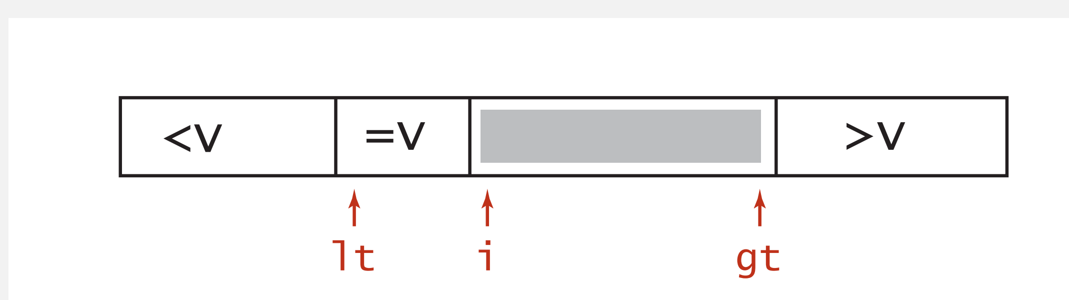


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

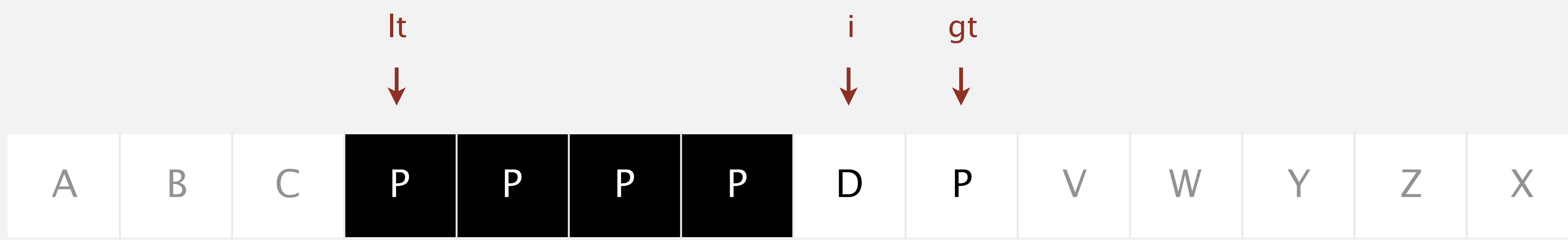



**invariant**

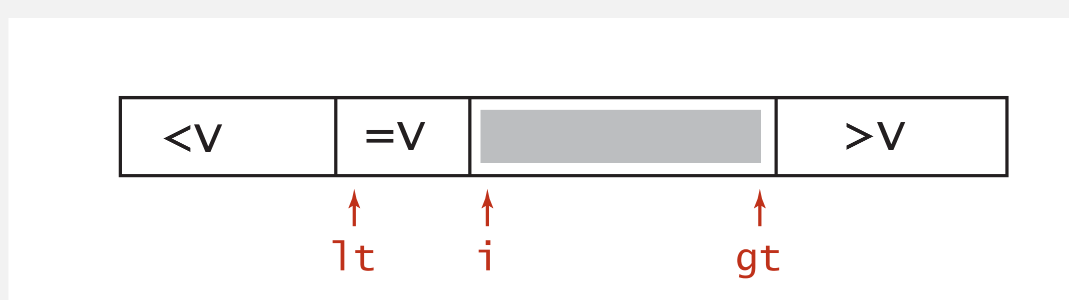


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

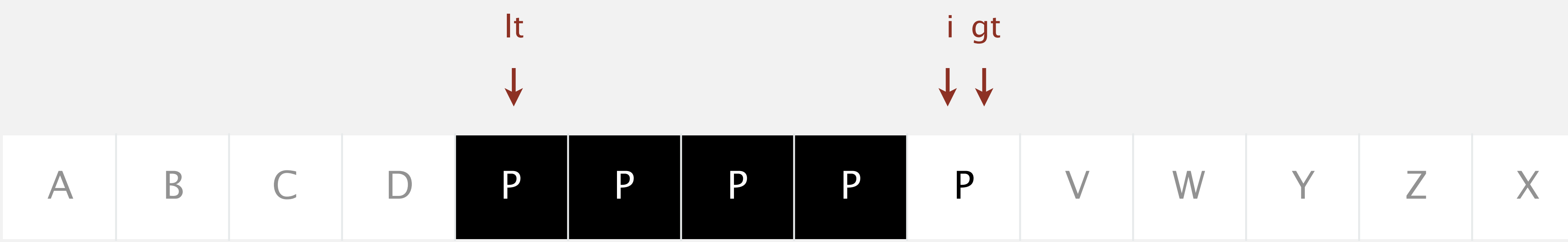



**invariant**

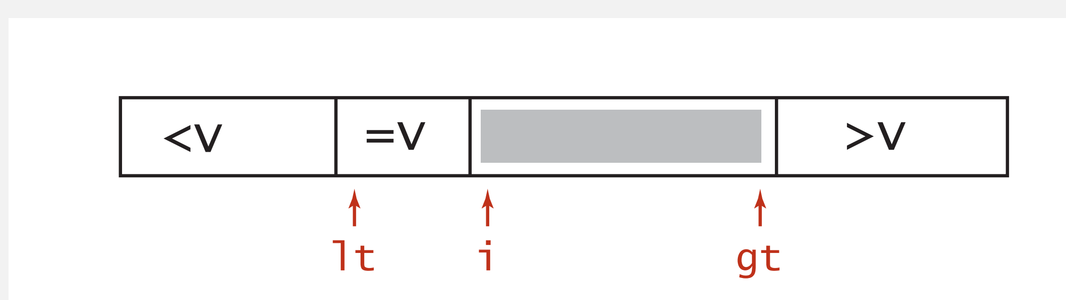


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

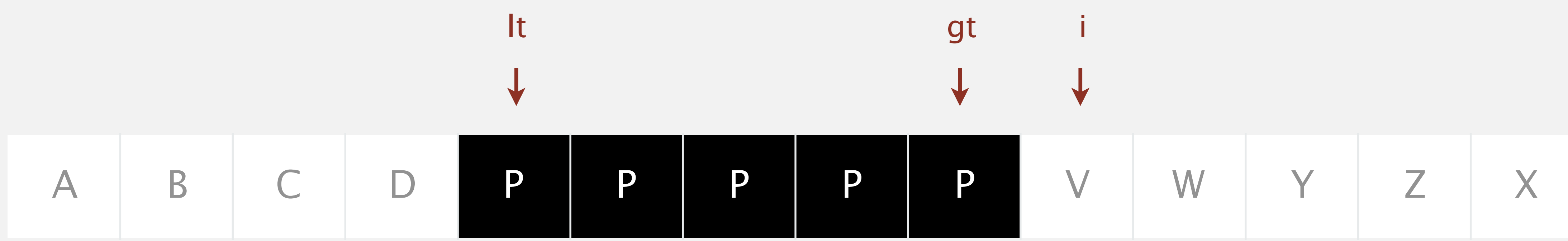



**invariant**

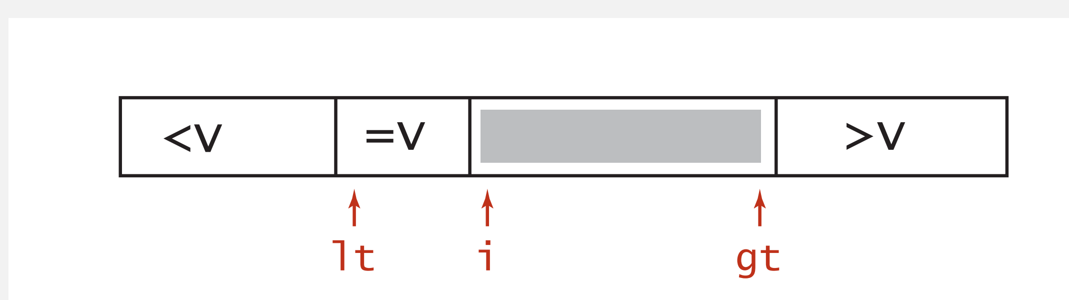


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

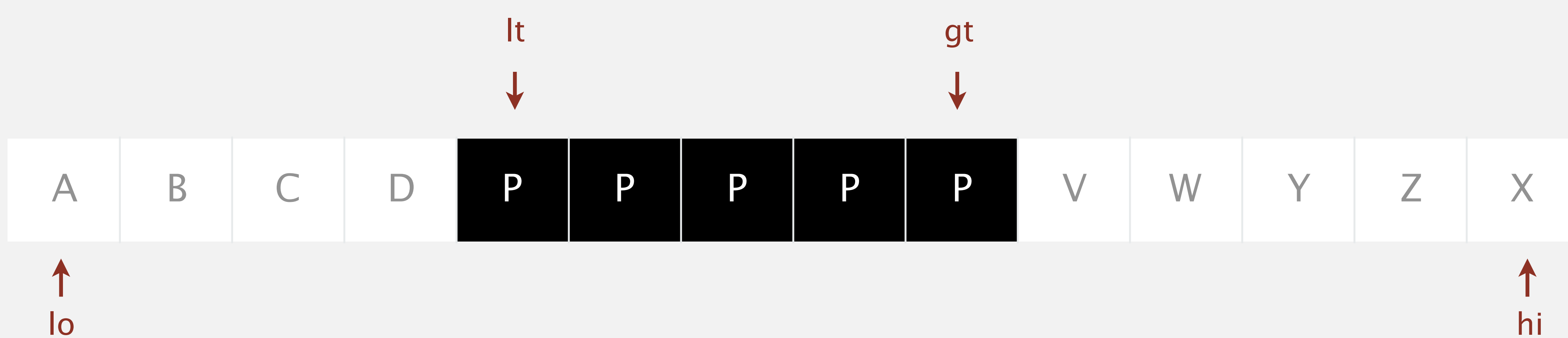



**invariant**

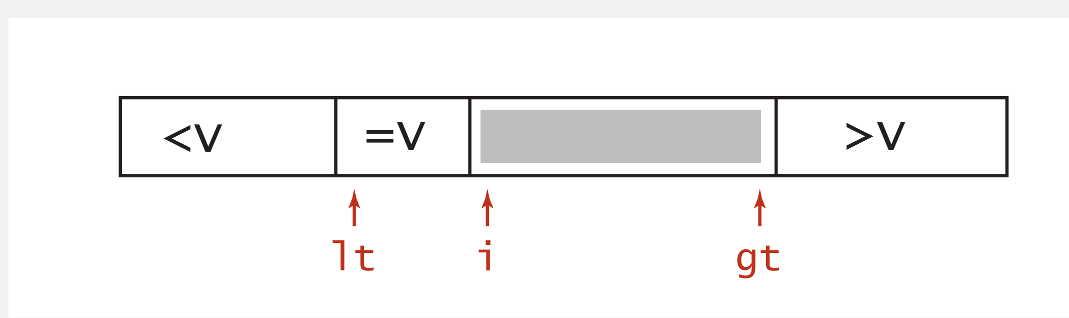


# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`

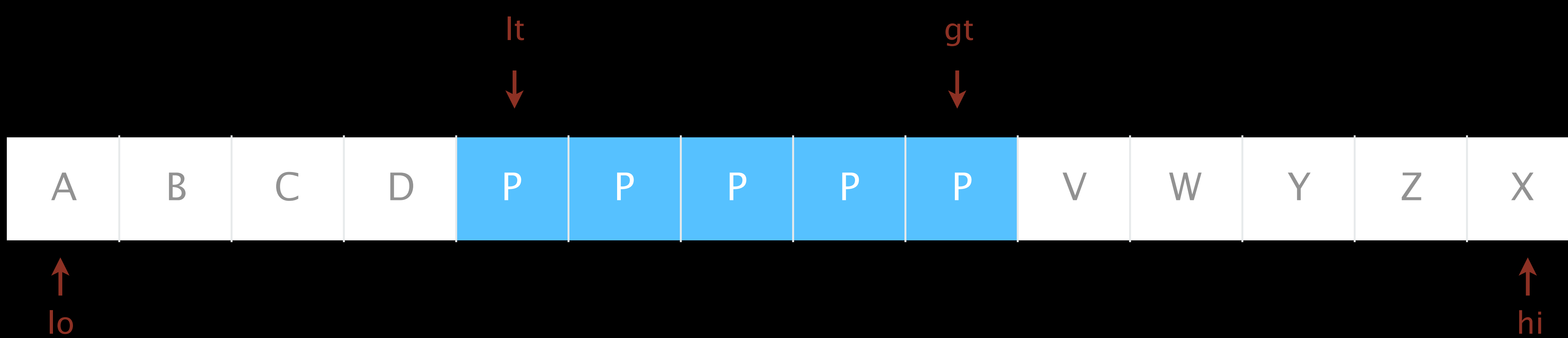


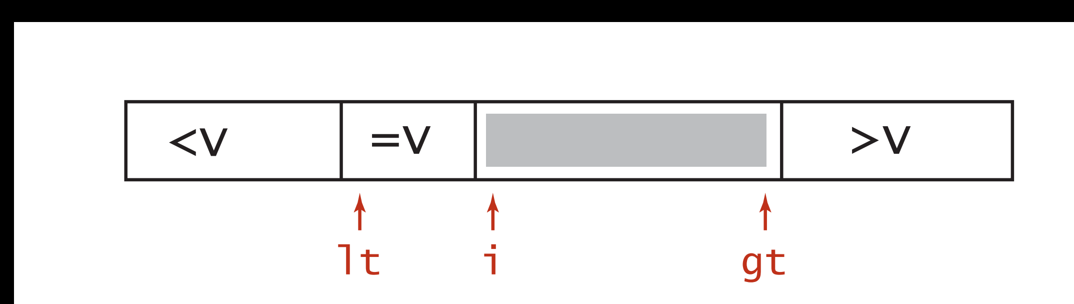

**invariant**



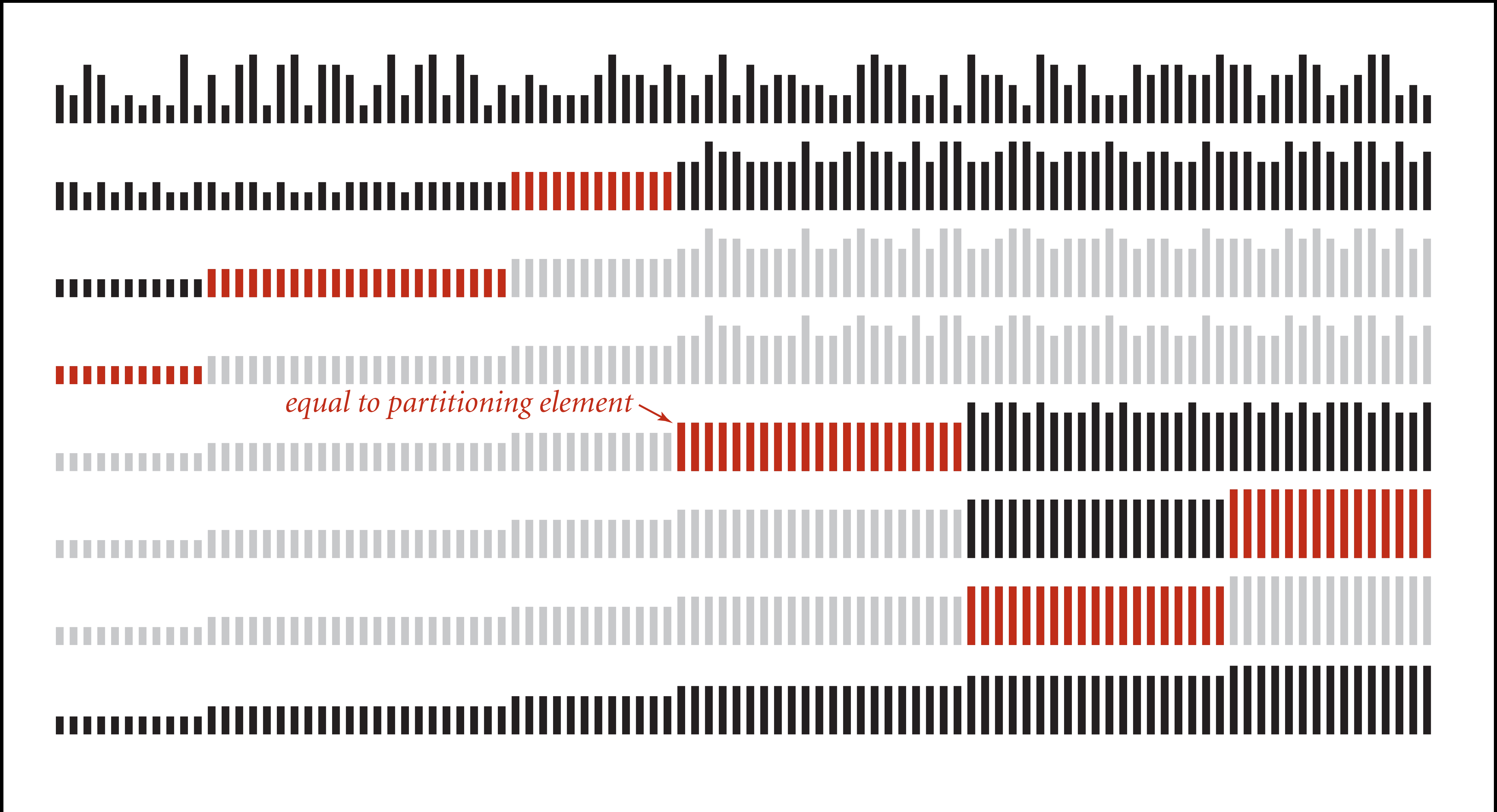
# Dijkstra 3-way partitioning demo

- Let `v` be partitioning item `a[lo]`.
- Scan `i` from left to right.
  - `(a[i]  < v)`: exchange `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: exchange `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`



# 3-way quicksort: visual trace



# A beautiful mailing list post (Yaroslavskiy, September 2011)

**Replacement of quicksort in java.util.Arrays with new dual-pivot quicksort**

Hello All,

I'd like to share with you new Dual-Pivot Quicksort which is faster than the known implementations (theoretically and experimental). I'd like to propose to replace the JDK's Quicksort implementation by new one.

...

The new Dual-Pivot Quicksort uses *two* pivots elements in this manner:

1. Pick an elements P1, P2, called pivots from the array.
2. Assume that P1 <= P2, otherwise swap it.
3. Reorder the array into three parts: those less than the smaller pivot, those larger than the larger pivot, and in between are those elements between (or equal to) the two pivots.
4. Recursively sort the sub-arrays.

The invariant of the Dual-Pivot Quicksort is:

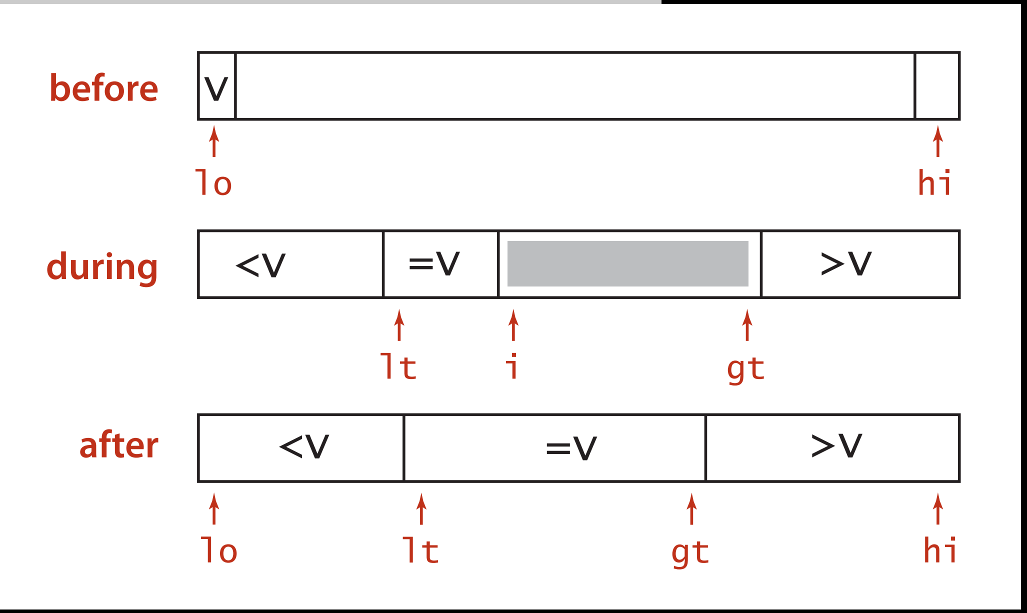
```
[ < P1 | P1 <= & <= P2 } > P2 ]
```

...

# 3-way quicksort: Java implementation

```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int lt = lo, gt = hi;
   Comparable v = a[lo];
   int i = lo;
   while (i <= gt)
   {
      int cmp = a[i].compareTo(v);
      if      (cmp < 0) exch(a, lt++, i++);
      else if (cmp > 0) exch(a, i, gt--);
      else              i++;
   }

   sort(a, lo, lt - 1);
   sort(a, gt + 1, hi);
}
```



# Sorting summary

| | inplace? | stable? | best | average | worst | remarks |
|---|---|---|---|---|---|---|
| selection | ✔ | | $\frac{1}{2} N^2$ | $\frac{1}{2} N^2$ | $\frac{1}{2} N^2$ | $N$ exchanges |
| insertion | ✔ | ✔ | $N$ | $\frac{1}{4} N^2$ | $\frac{1}{2} N^2$ | use for small $N$ or partially ordered |
| merge | | ✔ | $\frac{1}{2} N \lg N$ | $N \lg N$ | $N \lg N$ | $N \log N$ guarantee; stable |
| quick | ✔ | | $N \lg N$ | $2 N \ln N$ | $\frac{1}{2} N^2$ | $N \log N$ probabilistic guarantee; fastest in practice |
| 3-way quick | ✔ | | $N$ | $2 N \ln N$ | $\frac{1}{2} N^2$ | improves quicksort when duplicate keys |
| ? | ✔ | ✔ | $N$ | $N \lg N$ | $N \lg N$ | holy sorting grail |

# Dijkstra's 3-way partitioning: trace

| | | | v | | | | | a[] | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| lt | i | gt | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 0 | 0 | 11 | R | B | W | W | R | W | B | R | R | W | B | R |
| 0 | 1 | 11 | R | B | W | W | R | W | B | R | R | W | B | R |
| 1 | 2 | 11 | B | R | W | W | R | W | B | R | R | W | B | R |
| 1 | 2 | 10 | B | R | R | W | R | W | B | R | R | W | B | W |
| 1 | 3 | 10 | B | R | R | W | R | W | B | R | R | W | B | W |
| 1 | 3 | 9 | B | R | R | B | R | W | B | R | R | W | W | W |
| 2 | 4 | 9 | B | B | R | R | R | W | B | R | R | W | W | W |
| 2 | 5 | 9 | B | B | R | R | R | W | B | R | R | W | W | W |
| 2 | 5 | 8 | B | B | R | R | R | W | B | R | R | W | W | W |
| 2 | 5 | 7 | B | B | R | R | R | R | B | R | W | W | W | W |
| 2 | 6 | 7 | B | B | R | R | R | R | B | R | W | W | W | W |
| 3 | 7 | 7 | B | B | B | R | R | R | R | R | W | W | W | W |
| 3 | 8 | 7 | B | B | B | R | R | R | R | R | W | W | W | W |
| 3 | 8 | 7 | B | B | B | R | R | R | R | R | W | W | W | W |

**3-way partitioning trace (array contents after each loop iteration)**

# REFERENCES

- Robert Sedgewich & Kevin Wayne

    Section 3