ID1020: Quicksort

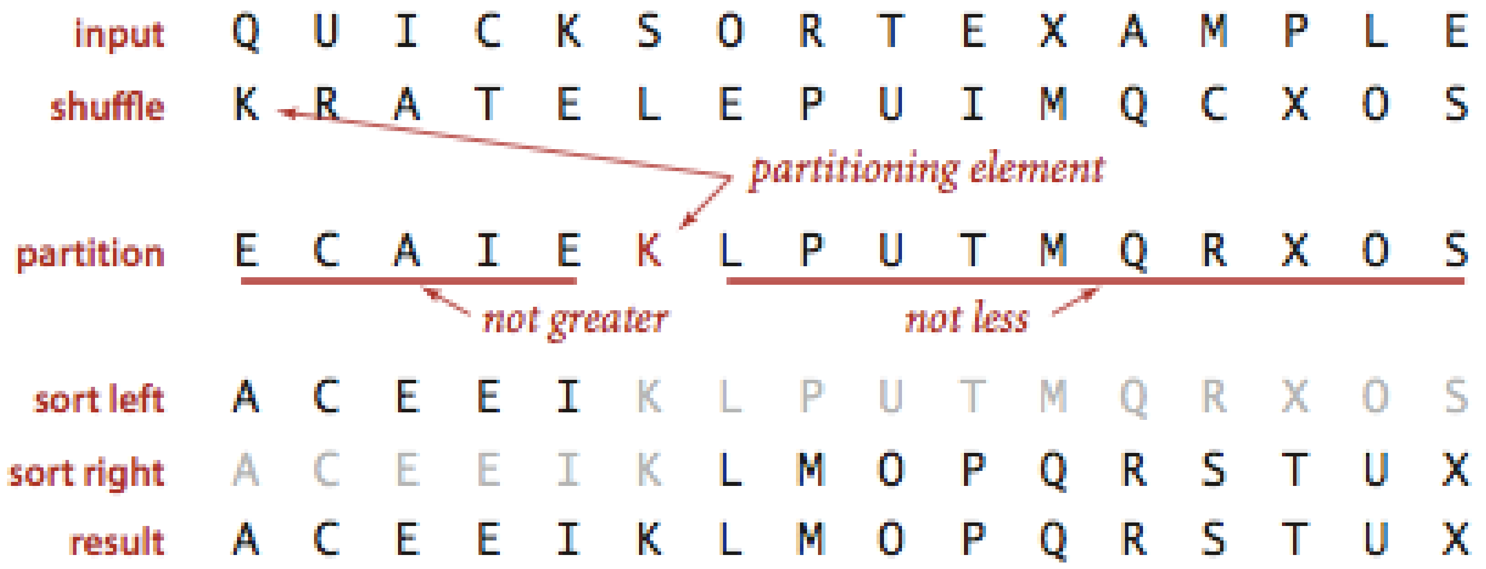
Dr. Per Brand
pbrand@kth.se

kap 2.3

Slides adapted from Algoritms 4th Edition, Sedgewick.

Quicksort

- Fundamental method.
  - Shuffle the array!!!
  - Partition the array for some element `j` so that
    - element `a[j]` is in the right slot
    - No larger element is to the left of `j`
    - No smaller element is to the right of `j`
  - Sort the subarray recursively.

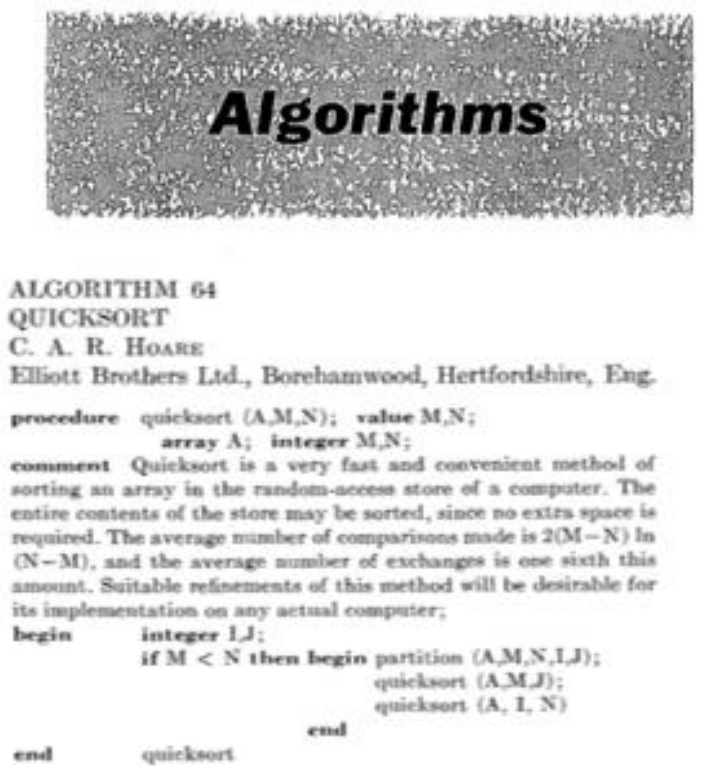


input Q U I C K S O R T E X A M P L E
shuffle K R A T E L E P U I M Q C X O S
partitioning element
partition E C A I E K L P U T M Q R X O S
not greater
not less
sort left A C E E I K L P U T M Q R X O S
sort right A C E E I K L M O P Q R S T U X
result A C E E I K L M O P Q R S T U X

Quicksort overview

Tony Hoare

- Hoare invented quicksort to translate from Russian to English
- Implemented for the first time in Algol 60 (using recursion).

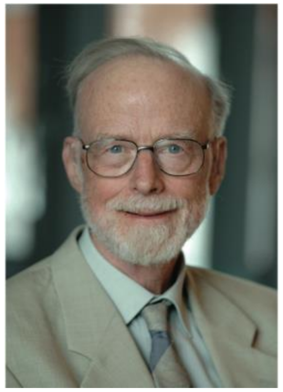


Algorithms

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
```

Communications of the ACM (July 1961)



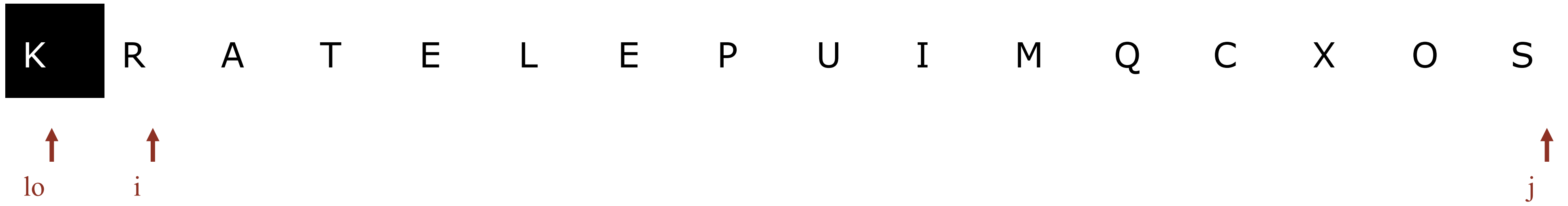
Tony Hoare

1980 Turing Award

“ 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. ”

Quicksort partitioning demo

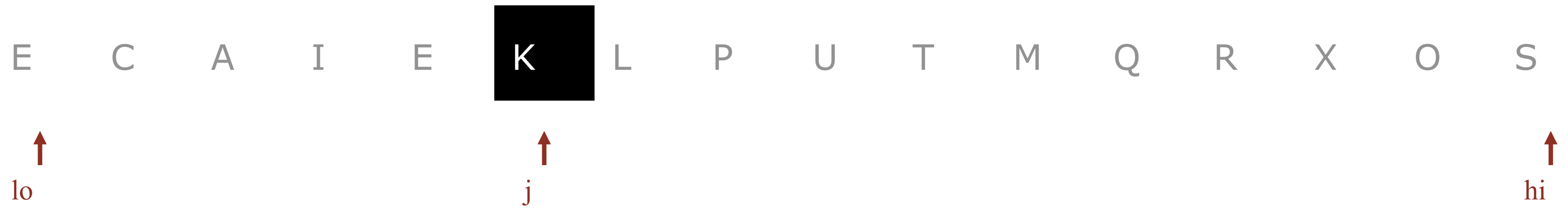
- Repeat until the scanning pointers i and j cross over.
  - Scan `i` from left to right `(a[i] < a[lo])`.
  - Scan from right to left as long `(a[j] > a[lo])`.
  - Swap `a[i]` with `a[j]`.



Quicksort partitioning demo

- Repeat until the scanning pointers i and j cross over.
  - Scan `i` from left to right `(a[i] < a[lo])`.
  - Scan from right to left as long `(a[j] > a[lo])`.
  - Swap `a[i]` with `a[j]`.

- When the scanning pointer cross
  - Swap `a[lo]` with `a[j]`.



partitioned!

Quicksort: Java code

```
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;
}
```

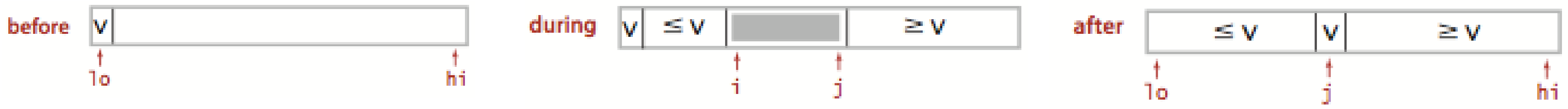
Find the element of the left that should be swapped

Find the element on the right that should be swapped

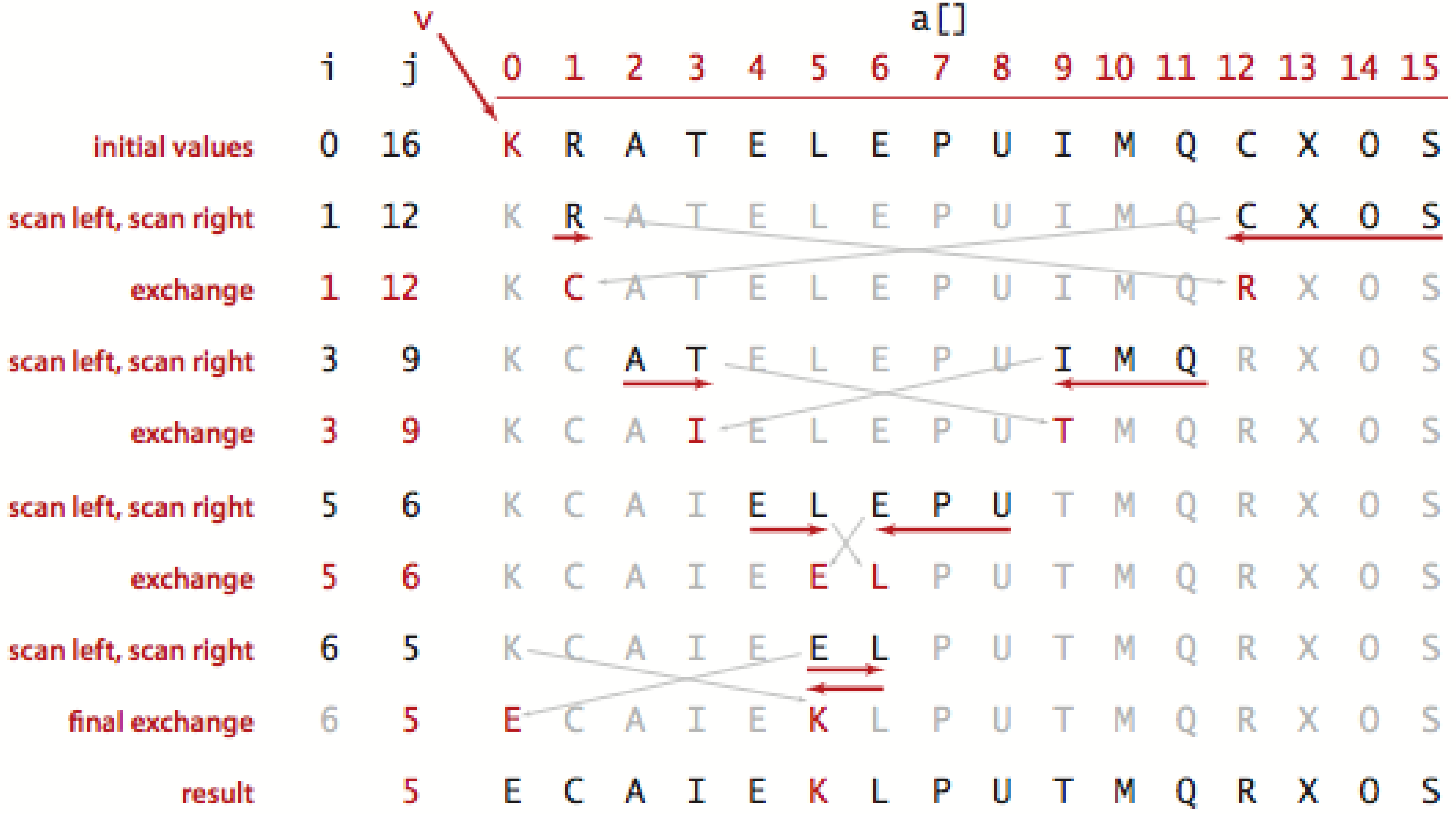
Check if the scanning pointers cross

swap

swap with partitioning element (pivot)
return index for boundary betweeen the two subarrays



Quicksort partitioning trace



	i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values	0	16	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
scan left, scan right	1	12	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
exchange	1	12	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
scan left, scan right	3	9	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S
exchange	3	9	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
scan left, scan right	5	6	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S
exchange	5	6	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
scan left, scan right	6	5	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S
final exchange	6	5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
result		5	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Partitioning trace (array contents before and after each exchange)

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);
  }
}
```

Shuffling!!

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: implementation

- Partitioning is in-place. Note that if we use an auxillary array that the code is simpler and that the algorithm can be made stable.
- To stop the iteration. Controlling when the scanning pointers cross is vital
- Duplicate keys. When there are duplicate keys, it is (counter-intuivately) att to stop scanning on keys that are equal to the partitioning element.
- Shuffling needed. Otherwise worst case input partitions into sets where one set is size 1, e.g., in order or reverse order
- An equivalent solution. Choose the partitioning element randomly in all subarrays.

Quicksort: empirical analys (1961)

- Runtime measurements and estimates:
  - Algol 60 implementation.
  - National-Elliott 405 computer.

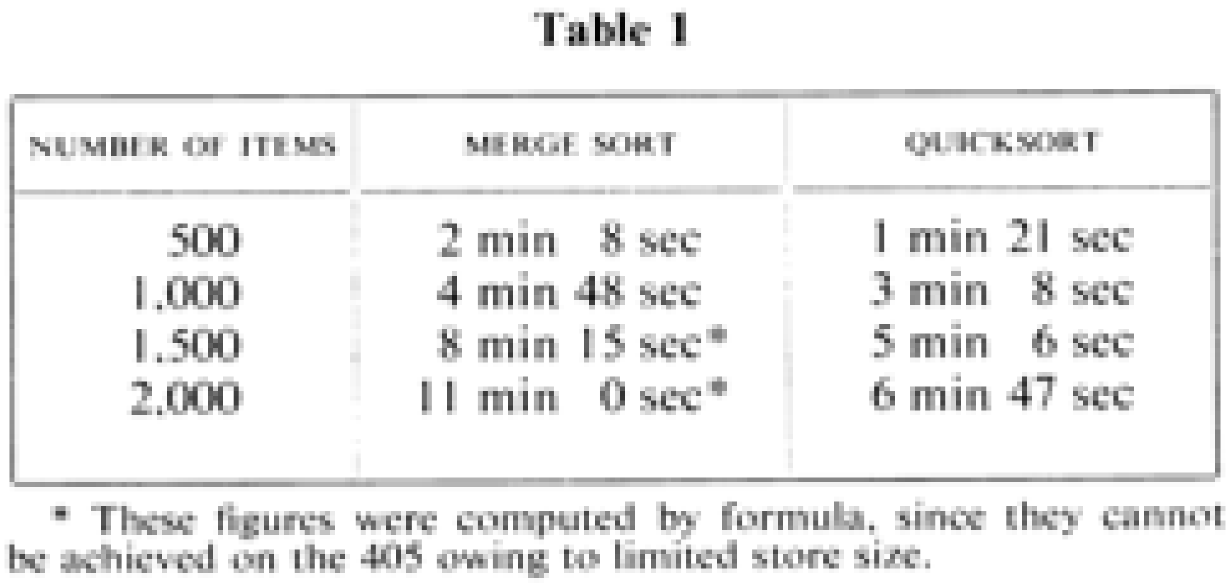


Table 1

NUMBER OF ITEMS	MERGE SORT	QUICKSORT
500	2 min 8 sec	1 min 21 sec
1,000	4 min 48 sec	3 min 8 sec
1,500	8 min 15 sec*	5 min 6 sec
2,000	11 min 0 sec*	6 min 47 sec

* These figures were computed by formula, since they cannot be achieved on the 405 owing to limited store size.

sorting N 6-word items with 1-word keys

Elliott 405 magnetic disc (16K words)

Quicksort: empirical analys

- Running times:
  - Home PC executes 10^8 comparisons/second.
  - Supercomputer executes 10^{12} comparisons/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 better than supercomputers.
- Lesson. Better algorithms better than good algorithms.

Quicksort: best-case analysis

- Best case. Number of comparisons $\sim N \lg N$.

			a[ ]														
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
input värden			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
blanda värden			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 comparisons $\sim \frac{1}{2} N^2$.

			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
No 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

Quicksort: average case analysis

- **Proof.** Average number of comparisons C_N to run quicksort on an array av N distinct keys is $\sim 2N \ln N$ (number of exchanges $\sim \frac{1}{3} N \ln N$).

- **Proof** C_N satisfies the recurrence relation $C_0 = C_1 = 0$ and for $N \geq 2$:

$$C_N = \overset{\text{partitioning}}{\overset{\downarrow}{(N+1)}} + \left(\frac{C_0 + C_{N-1}}{N}\right) + \left(\frac{\overset{\text{left}}{\overset{\downarrow}{C_1}} + \overset{\text{right}}{\overset{\downarrow}{C_{N-2}}}}{\underset{\nwarrow \text{ partitionings probability}}{N}}\right) + \ldots + \left(\frac{C_{N-1} + C_0}{N}\right)$$

  - Muliply both sides by N and collect terms:

$$NC_N = N(N+1) + 2(C_0 + C_1 + \ldots + C_{N-1})$$

  - Ta bort från denna ekvation samma ekvation för $N-1$ fallet:

$$NC_N - (N-1)C_{N-1} = 2N + 2C_{N-1}$$

  - Collect terms and divide by $N(N+1)$:

$$\frac{C_N}{N+1} = \frac{C_{N-1}}{N} + \frac{2}{N+1}$$

Quicksort: average-case analys

- Repeated application

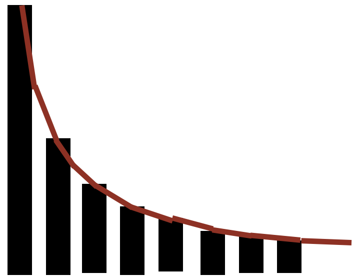
Previous equation

$$\begin{aligned}\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}\end{aligned}$$

Substitute

- Approximate sum with intergral:

$$\begin{aligned}C_N &= 2(N+1)\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots \frac{1}{N+1}\right) \\ &\sim 2(N+1)\int_3^{N+1} \frac{1}{x}\,dx\end{aligned}$$



- Final result:

$$C_N \sim 2(N+1)\ln N \approx 1.39 N \lg N$$

Quicksort: summary of performance

- Quicksort is a randomized algorithm.
  - Provides a probabilistic quarantee.
  - Runtime depends on the shuffling outcome.

- Average case. Number of comparisons~ 1.39 $N \lg N$.
  - 39% more comparisons than mergesort .
  - But faster than mergesort in practice – less copying of data.

- Best case. Number of comparisons is ~ $N \lg N$.

- Worst case. Number of comarisons~ ½ N^2.

Quicksort properties

- Theorem. Quicksort is an in-place sorting algorithm.
- Proof.
  - Partitioning: constant extra space.
  - Depth of recursion: logarithmic extra space (with high probability).

- Theorem. Quicksort is not stable.
- Proof by counterexample.

i	j	0	1	2	3
		B_1	C_1	C_2	A_1
1	3	B_1	C_1	C_2	A_1
1	3	B_1	A_1	C_2	C_1
0	1	A_1	B_1	C_2	C_1

Quicksort: improvement

- Insertion sort on small subarray.
  - Even quicksort has too much overhead for small subarrays.
  - Cutoff to insertion sort for less than 10 elements.

```
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: improvement

- Partititioning element = median of small sample .
  - Best choice = median. Why ?
  - Approximate median by calculating median of a sample.
  - Median-of 3 random chosen elements.

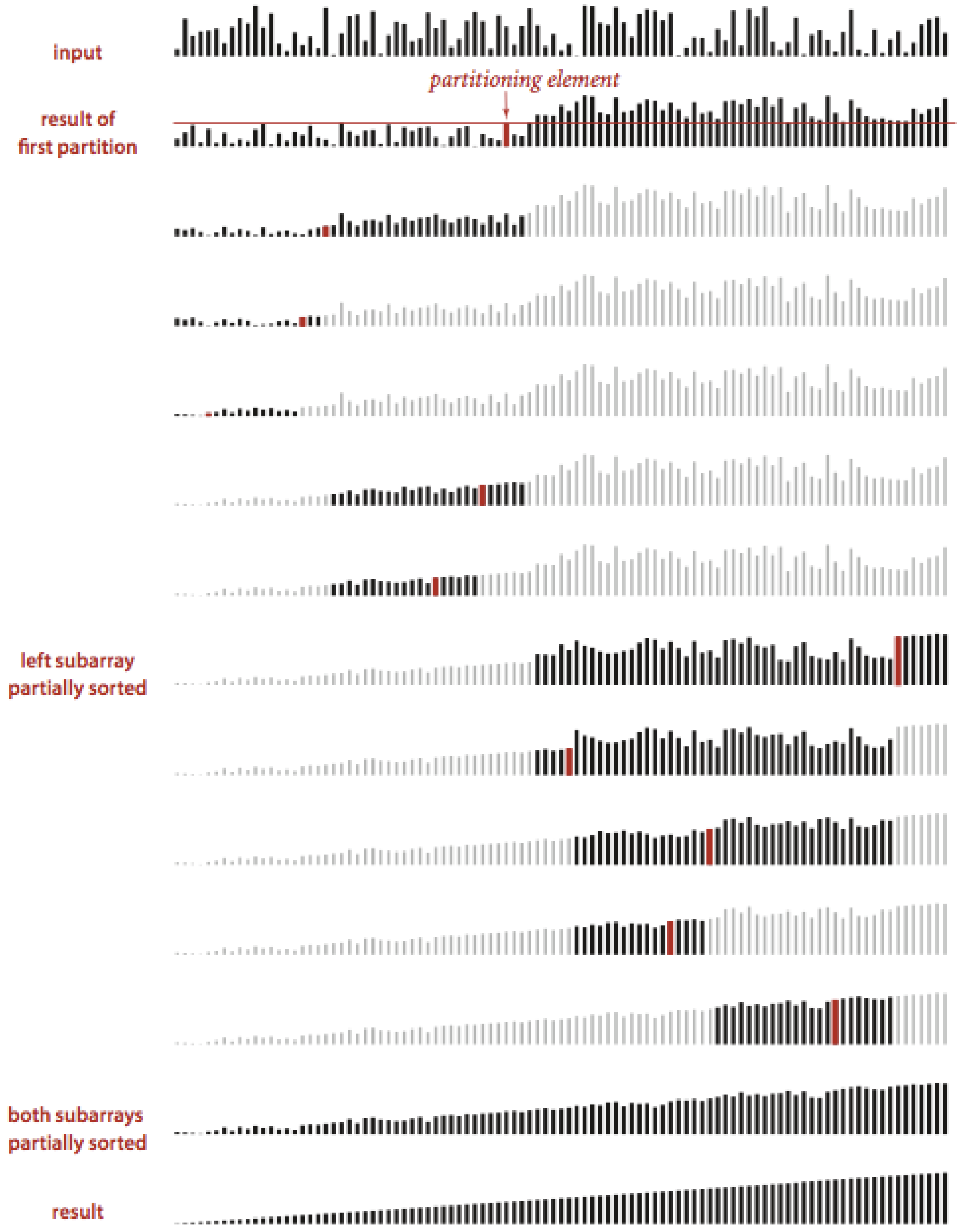
~ 12/7   N ln N comparisoons (14% mindre)
~ 12/35 N ln N exchanges (3% mer)

```
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);
}
```

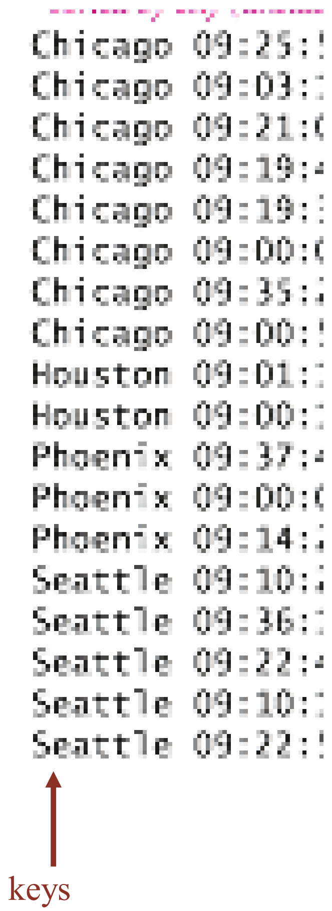
Quicksort with median-of-3 and cutoff to insertion sort: visualization



Duplicate keys

- Many duplicate keys is a common case .
  - E.g. Sorting people by age.
  - Removing duplicates from e-mail lists
  - Sorting job applicants after grades.

- Properties.
  - Very large array.
  - A limited number of distinct keys.

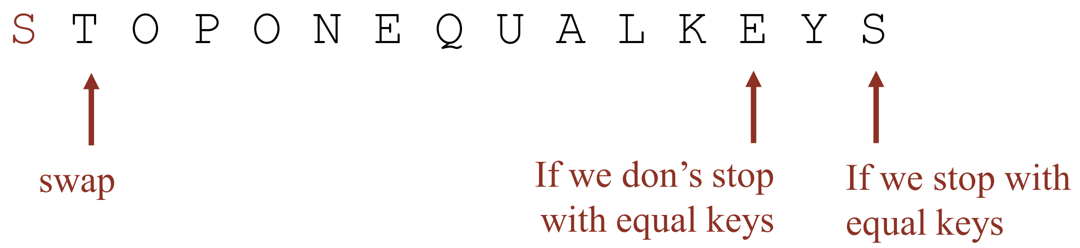


```
Chicago 09:25:
Chicago 09:03:
Chicago 09:21:
Chicago 09:19:
Chicago 09:19:
Chicago 09:00:
Chicago 09:35:
Chicago 09:00:
Houston 09:01:
Houston 09:00:
Phoenix 09:37:
Phoenix 09:00:
Phoenix 09:14:
Seattle 09:10:
Seattle 09:36:
Seattle 09:22:
Seattle 09:10:
Seattle 09:22:
```

Quicksort with duplicate keys

Duplicate keys

- Quicksort with duplicate keys. Algorithm can take quadratic time if partitioning does not stop when comparing values with equal keys!


S T O P O N E Q U A L K E Y S
swap
If we don’s stop with equal keys
If we stop with equal keys

- Some implementations (both in books and in use) take quadratic time when the input contains many elements with the same key.
- In the 1990s the problem was detected in a standard library in C.

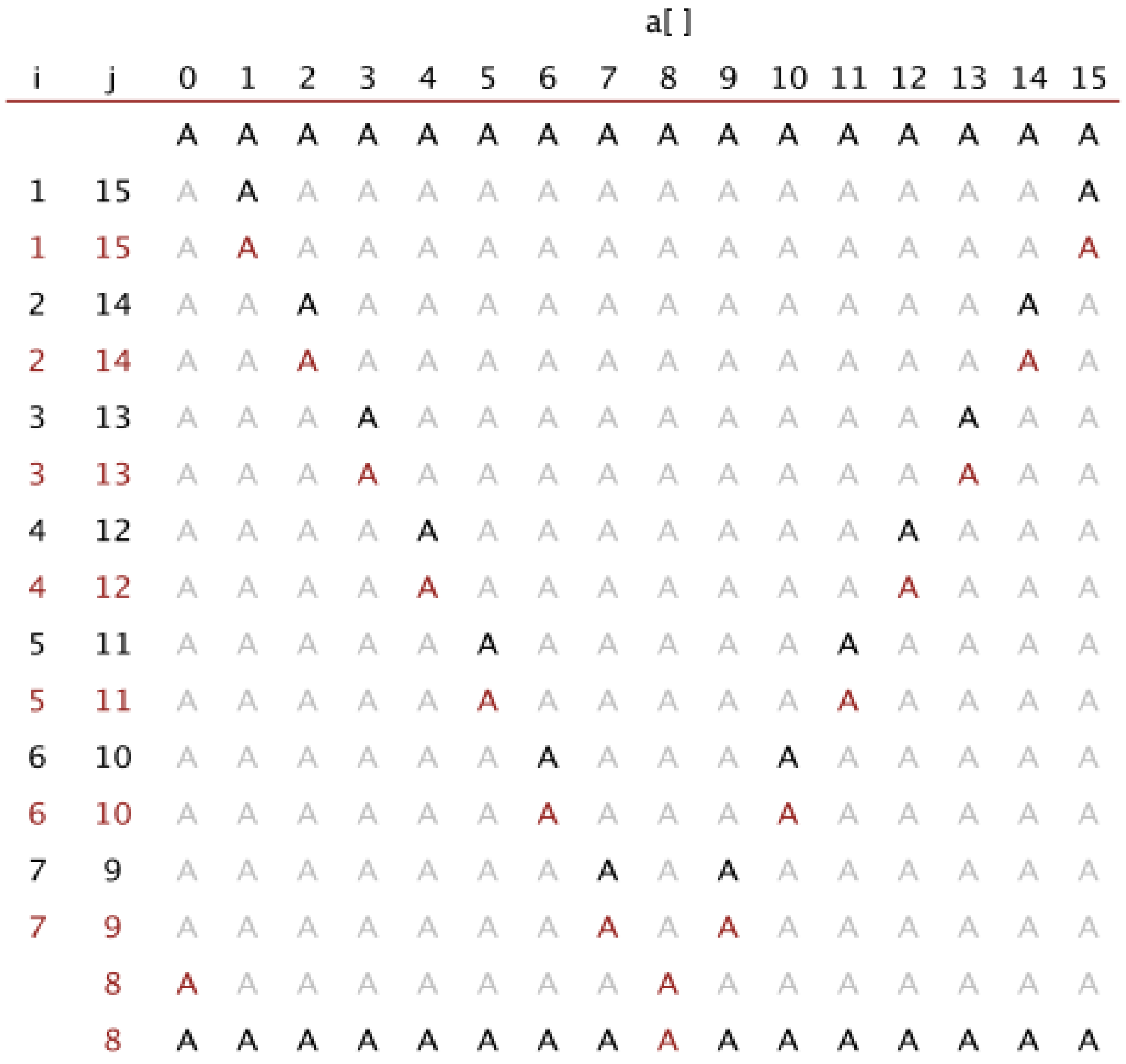
Partitioning with duplicates

- Where does the partitioning element end up if we sort the array below with Quicksort?

$\boxed{A}$ A A A A A A A A A A A A A A A

- A. $\boxed{A}$ 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 $\boxed{A}$
- C. A A A A A A A A $\boxed{A}$ A A A A A A A

Partitioning with duplicates



		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

The problem with duplicate keys

- Mistake. Continue scanning when elements are reached with the same key as the partitioning element.
- Result. $\sim \frac{1}{2} N^2$ comparisons when all keys are equal.
- .

```
B A A B A B C C B C B          A A A A A A A A A A A
```

- The right way. Stop the scan when reaching element with the same key as the partititioning element.
- Result. $\sim N \lg N$ compares when all keys equal

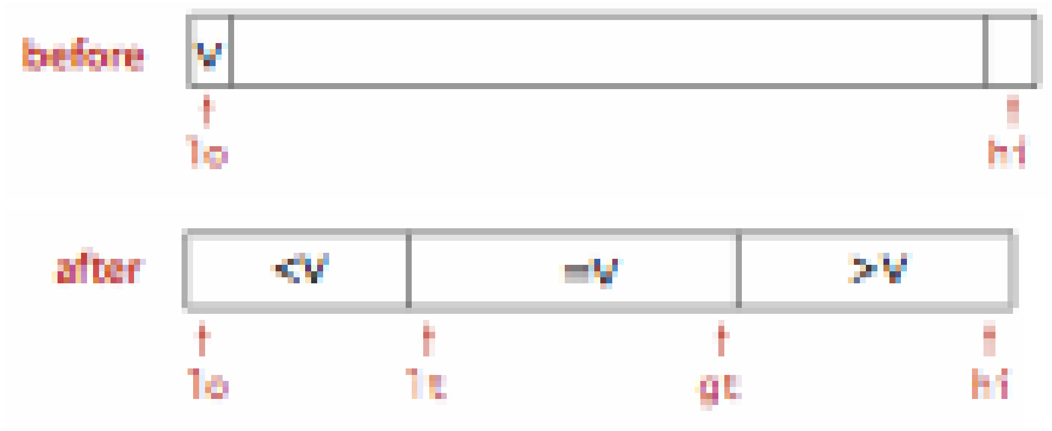
```
B A A B A B B B C C C          A A A A A A A A A A A
```

- Desirable. Put all elements that are equal to the partitioning element in place

```
A A A B B B B B C C C          A A A A A A A A A A A
```

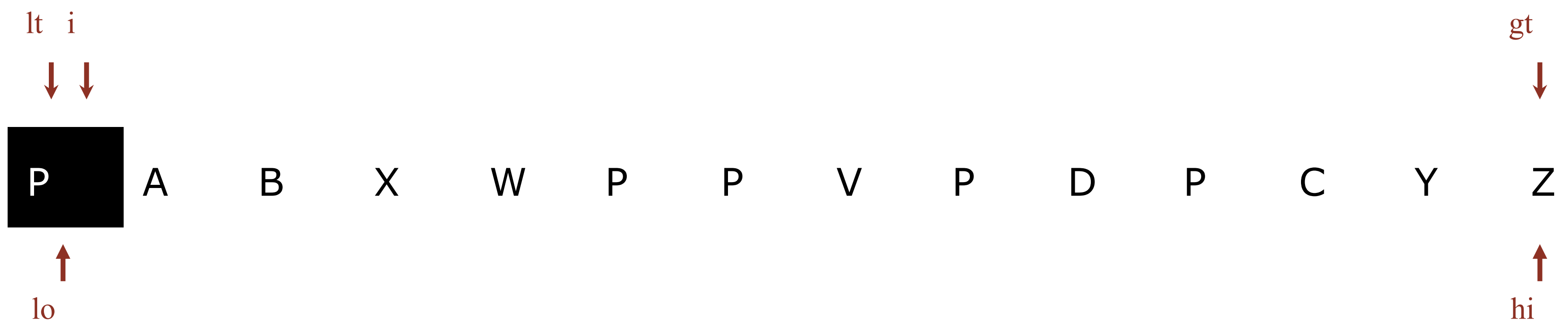
3-way partitioning

- Goal: Partition the array:n into three sets:
  - Equal elements between `lt` **and** `gt`
  - Smaller elements to the left of `lt`.
  - Larger elements to the right of `gt`.
-

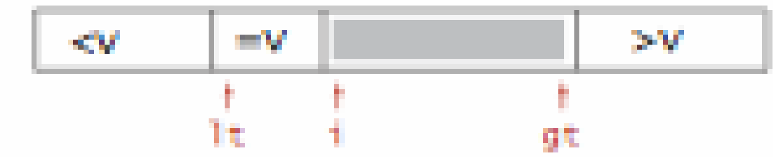


Dijkstra 3-vägs partitionering

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

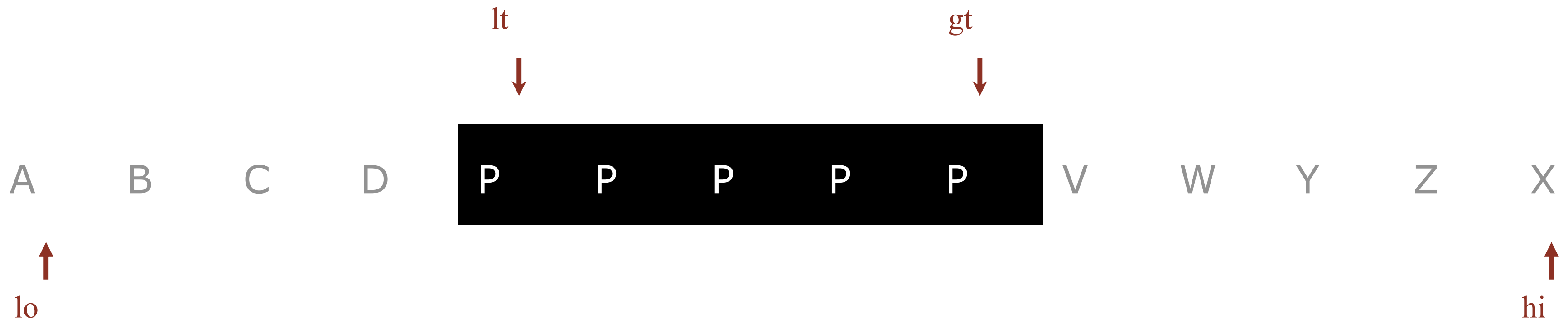


invariant

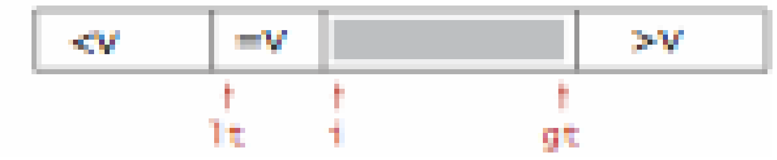


Dijkstra 3-way partitioning demo

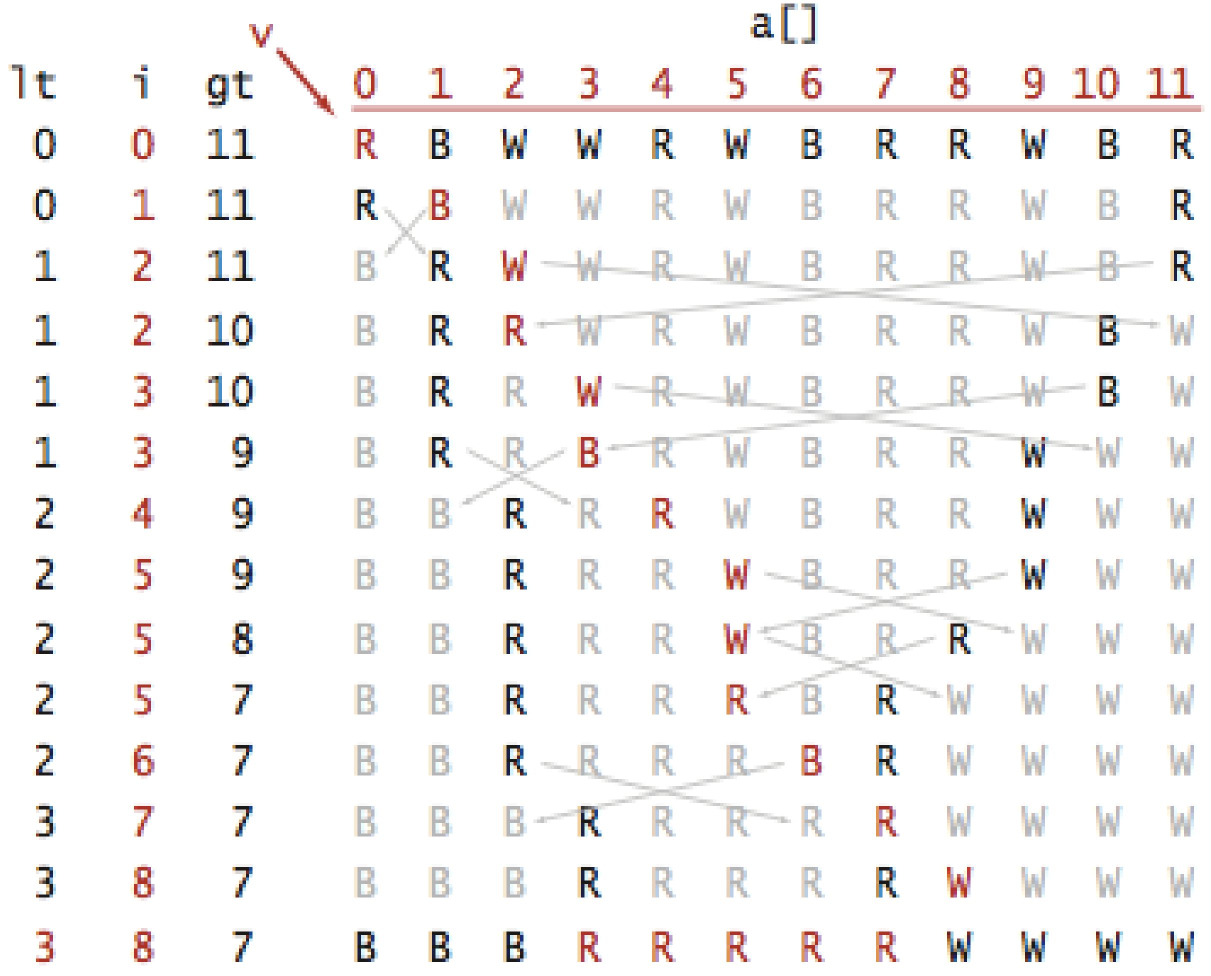
- Let `v` be the partitioning element in `a[lo]`.
- Scan with `i` from left to right.
  - `(a[i]  < v)`: swap `a[lt]` with `a[i]`; increment both `lt` and `i`
  - `(a[i]  > v)`: swap `a[gt]` with `a[i]`; decrement `gt`
  - `(a[i] == v)`: increment `i`



invariant



Dijkstra's 3-way partitioning: trace



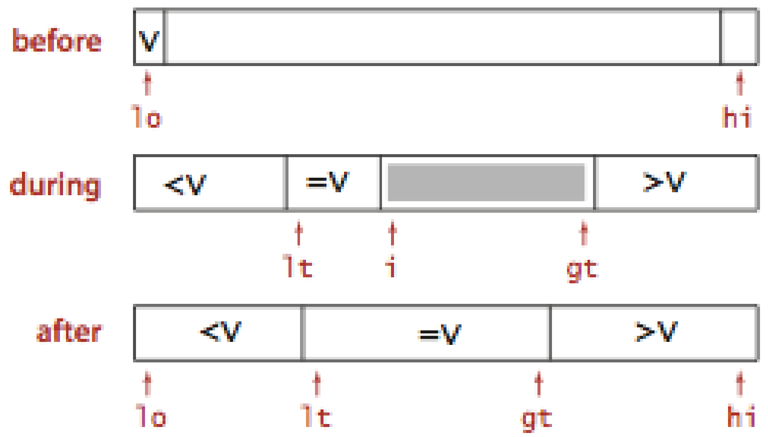
lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
0	0	11	R	B	W	W	R	W	B	R	R	W	B	R
0	1	11	R	B	W	W	R	W	B	R	R	W	B	R
1	2	11	B	R	W	W	R	W	B	R	R	W	B	R
1	2	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	10	B	R	R	W	R	W	B	R	R	W	B	W
1	3	9	B	R	R	B	R	W	B	R	R	W	W	W
2	4	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	9	B	B	R	R	R	W	B	R	R	W	W	W
2	5	8	B	B	R	R	R	W	B	R	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)

3-way quicksort: Java implementation

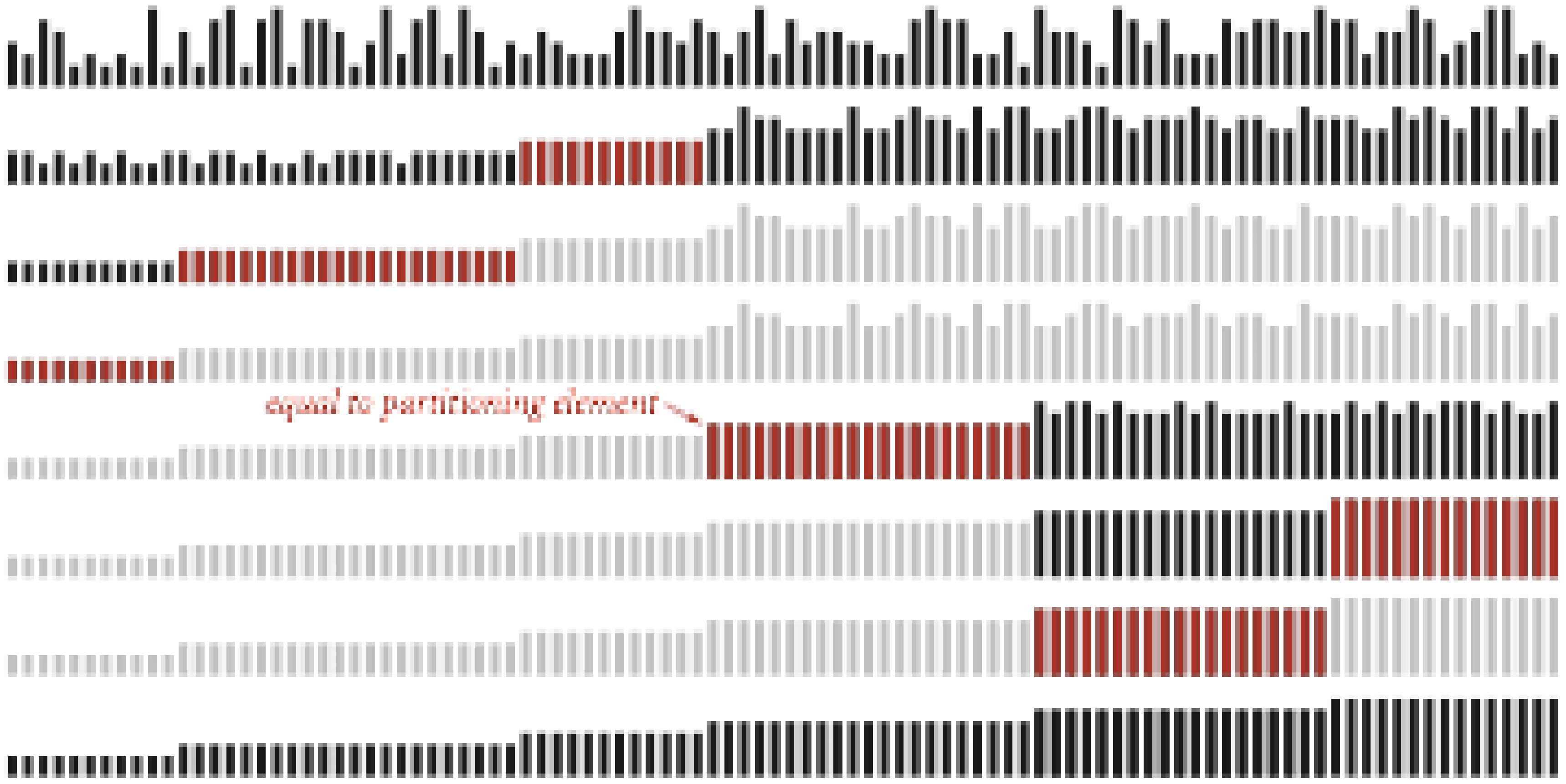
```
private static void sort(Comparable[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int lt = lo, gt = hi;
   Comparable v = a[lo];
   int i = lo;
   while (i <= gt)
   {
      int cmp = a[i].compareTo(v);
      if       (cmp < 0) exch(a, lt++, i++);
      else if (cmp > 0) exch(a, i, gt--);
 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

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



3-way partitioning overview

3-way quicksort: visual trace



Duplicate keys: lower bound

- Lower bound. If there are n distinct keys and the element i has x_i duplicates, then all comparison-based sorting algorithms require at least

$$\lg\left(\frac{N!}{x_1!\, x_2! \cdots x_n!}\right) \sim -\sum_{i=1}^{n} x_i \lg \frac{x_i}{N}$$

$N \lg N$ when all keys distinct;
Linear when there are constant number of distinct keys.

comparisons in worst case.

Proportional to lower bound

- Theorem. [Sedgewick-Bentley 1997]
- Quicksort with 3-way partitioning is entropy-optimal.
- Proof. [beyond scope of course]

- Conclusion. Quicksort with 3-way partitioning reduces running time form linearithmic to linear for many applications.

Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✓		½ N^2	½ N^2	½ N^2	N exchanges
insertion	✓	✓	N	¼ N^2	½ N^2	use for small N or partially ordered
shell	✓		$N \log_3 N$	?	$c N^{3/2}$	tight code; subquadratic
merge		✓	½ $N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
quicksort	✓		$N \lg N$	$2 N \ln N$	½ N^2	$N \log N$ probabilistic guarantee; fastest in practice
3-way quicksort	✓		N	$2 N \ln N$	½ N^2	improves quicksort when duplicate keys
?	✓	✓	N	$N \lg N$	$N \lg N$	holy sorting grail

Which sorting algorithm should one use?

- There are many to choose from:

elementary sorts	insertion sort, selection sort, bubblesort, shaker sort, ...
subquadratic sorts	quicksort, mergesort, heapsort, shellsort, samplesort, ...
system sorts	dual-pivot quicksort, timesort, introsort, ...
external sorts	Poly-phase mergesort, cascade-merge, psort, ...
radix sorts	MSD, LSD, 3-way radix quicksort, …
parallel sorts	bitonic sort, odd-even sort, smooth sort, GPUsort, ...

Which sorting algorithm should one use?

- Applications have different requirements:
  - Stable?
  - Parallel?
  - In-place?
  - Deterministic?
  - Duplicate keys?
  - Multiple key types?
  - Array or linked list?
  - Large or small elements?
  - If the array randomly ordered?
  - Worst or average case focus?

- Is ”system sort” good enough?
- Yes, usually.

System sorting i Java 7

- `Arrays.sort()`.
  - Has a method for objects that are `Comparable`.
  - Different method for primitive types.
  - Has a method for using a `Comparator`.

- Algorithm.
  - Dual-pivot (2 partitioning elements) quicksort for primitive types.
  - Timsort (mergesort variant) for reference types.

- Why are different algorithms used for primitives versus reference types?

Cases

- Best, worst, average, expected.
- Running time depends on
  - Property of input
    - For ex. Random, almost ordered, reverse ordered
  - Properties of algorithm
    - Including all internal decisions.
- Best case
  - Best input, best internal decisions.
- Worst case
  - Worst input, worst internal decisions

Expected contra average

- Average case
  - average input, average internal decisions.
- Expected case
  - worst input, average internal decisions

- Often the same, but sometimes not !!!
- How is with quicksort with and without shuffling?

Expected vs average

- Average case
  - average input, average internal decisions.
- Expected case
  - Worst input, average internal decisions
- Example: qsort without and with shuffling
- Average
  - N log N for both versions
- Expected
  - N log N for shuffling version
  - N^2 without shuffling.