Efficient Sorting Algorithms

Mark Matthews PhD

Summary

- We've come to the end of our sorting journey
- QuickSort - the best sorting algorithm we have?
- Sorting summary

QuickSort

Quicksort

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

Quicksort.

- Used in many standard libraries
- Very robust sort for sorting many types of inputs
- Recursive algorithm
- Divide & Conquer Strategy

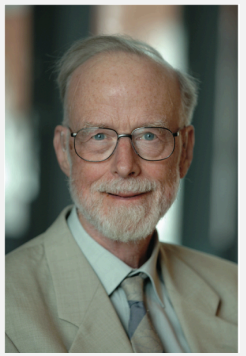
Quicksort

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.
  - [but couldn't explain his algorithm or implement it!]
- Learned Algol 60 (and recursion).
- Implemented quicksort.
- Recursion done after work done
- MergeSort work done before recursion



Tony Hoare
1980 Turing Award

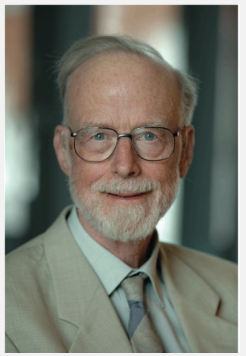
ALGORITHM 64
QUICKSORT
C. A. R. HOARE
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
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

- Invented quicksort to translate Russian into English.
  - [but couldn't explain his algorithm or implement it!]
- Learned Algol 60 (and recursion).
- Implemented quicksort.



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. "

Bob Sedgewick

- Refined and popularized quicksort.
- Analyzed quicksort.

Bob Sedgewick

Programming Techniques

S. L. Graham, R. L. Rivest Editors

Implementing Quicksort Programs

Robert Sedgewick
Brown University

This paper is a practical study of how to implement the Quicksort sorting algorithm and its best variants on real computers, including how to apply various code optimization techniques. A detailed implementation combining the most effective improvements to Quicksort is given, along with a discussion of how to implement it in assembly language. Analytic results describing the performance of the programs are summarized. A variety of special situations are considered from a practical standpoint to illustrate Quicksort's wide applicability as an internal sorting method which requires negligible extra storage.

Key Words and Phrases: Quicksort, analysis of algorithms, code optimization, sorting
CR Categories: 4.0, 4.6, 5.25, 5.31, 5.5

Acta Informatica 7, 327—355 (1977)

The Analysis of Quicksort Programs*

Robert Sedgewick

Received January 19, 1976

Summary. The Quicksort sorting algorithm and its best variants are presented and analyzed. Results are derived which make it possible to obtain exact formulas describing the total expected running time of particular implementations on real computers of Quicksort and an improvement called the median-of-three modification. Detailed analysis of the effect of an implementation technique called loop unwrapping is presented. The paper is intended not only to present results of direct practical utility, but also to illustrate the intriguing mathematics which arises in the complete analysis of this important algorithm.

QuickSort Basic Steps

Partition and Order

1. Pick an element for your pivot
2. Put all elements less than to its left
3. Put all elements greater than to right
4. Repeat

Demo

Quicksort

Basic plan.

input

Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E

Quicksort

Basic plan.

- Shuffle the array.

shuffle

Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E

Quicksort

Basic plan.

- Shuffle the array.

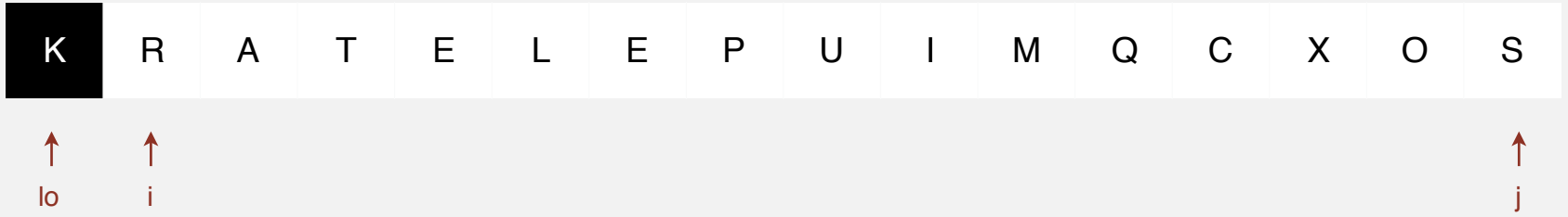
shuffle

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

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].



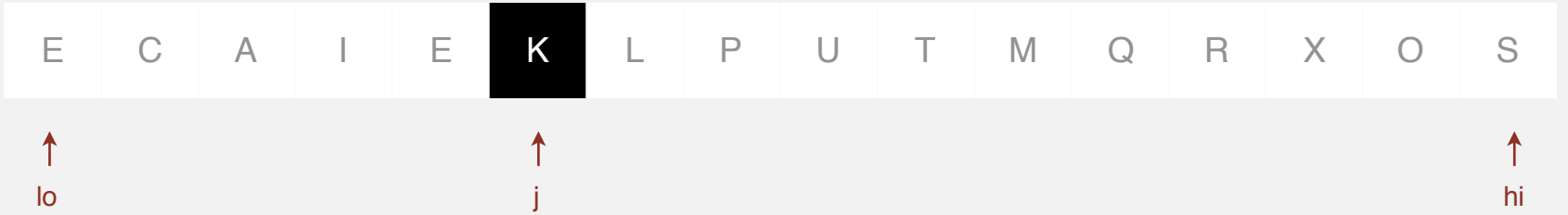
Quicksort partitioning demo

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].

When pointers cross.

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



partitioned!

Quicksort

Basic plan.

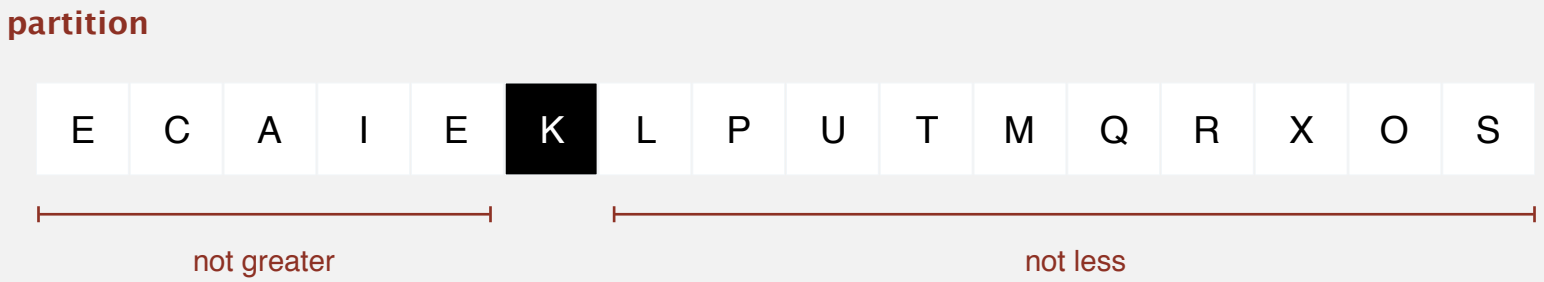
- Shuffle the array.
- Partition so that, for some j
  - entry a[j] is in place
  - no larger entry to the left of j
  - no smaller entry to the right of j

partition

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



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.

sort the left subarray

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.

sort the left subarray

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.

sort the right subarray

A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S

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.

sort the right subarray

A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

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.

sorted array

A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort: pseudocode

```
/* low  --> Starting index,  high  --> Ending index
*/
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Partition: pseudocode

```
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller
element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

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

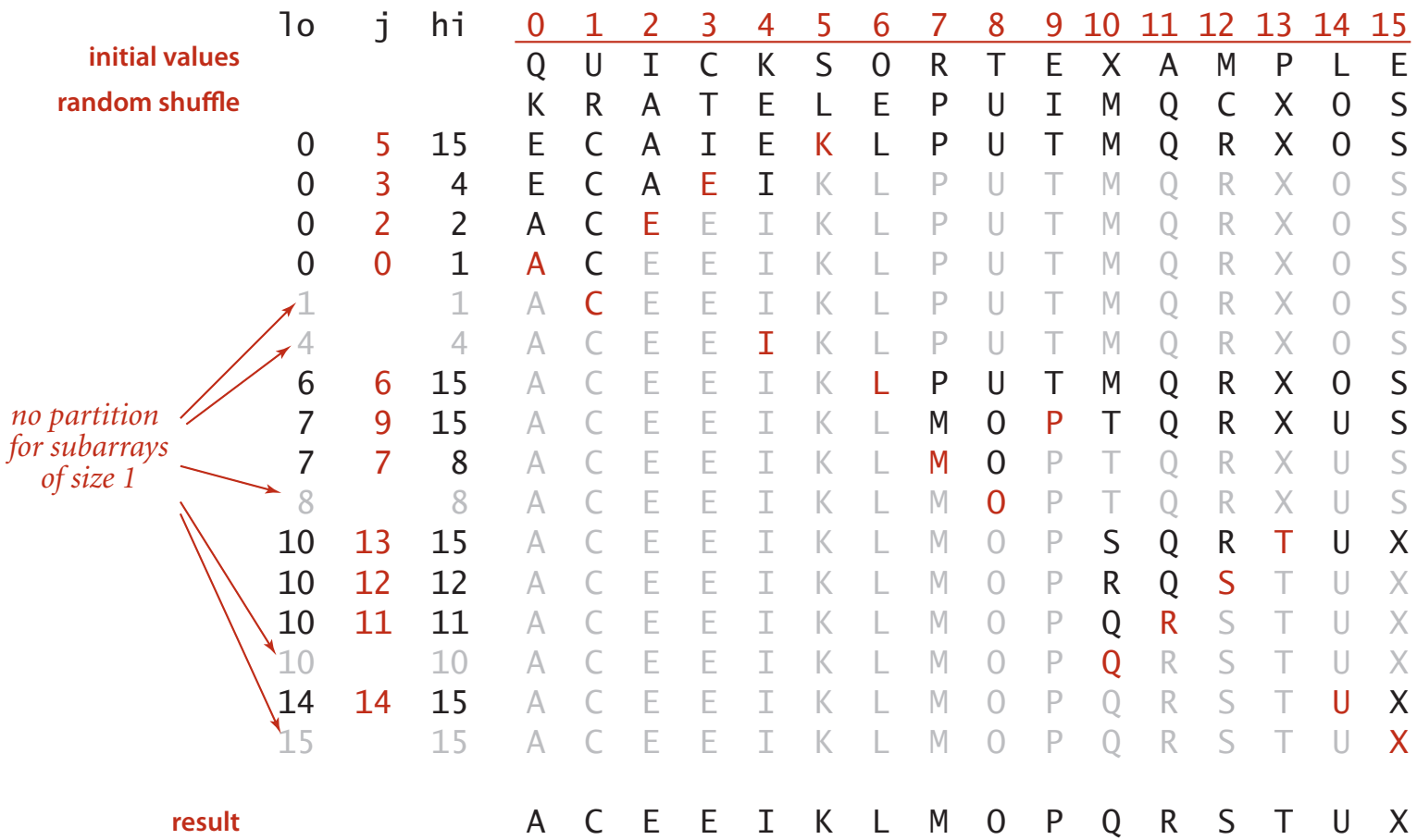
```
public class Quick
{
   private static int partition(Comparable[] a, int lo, int hi)
   {  /* see previous slide */  }

   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

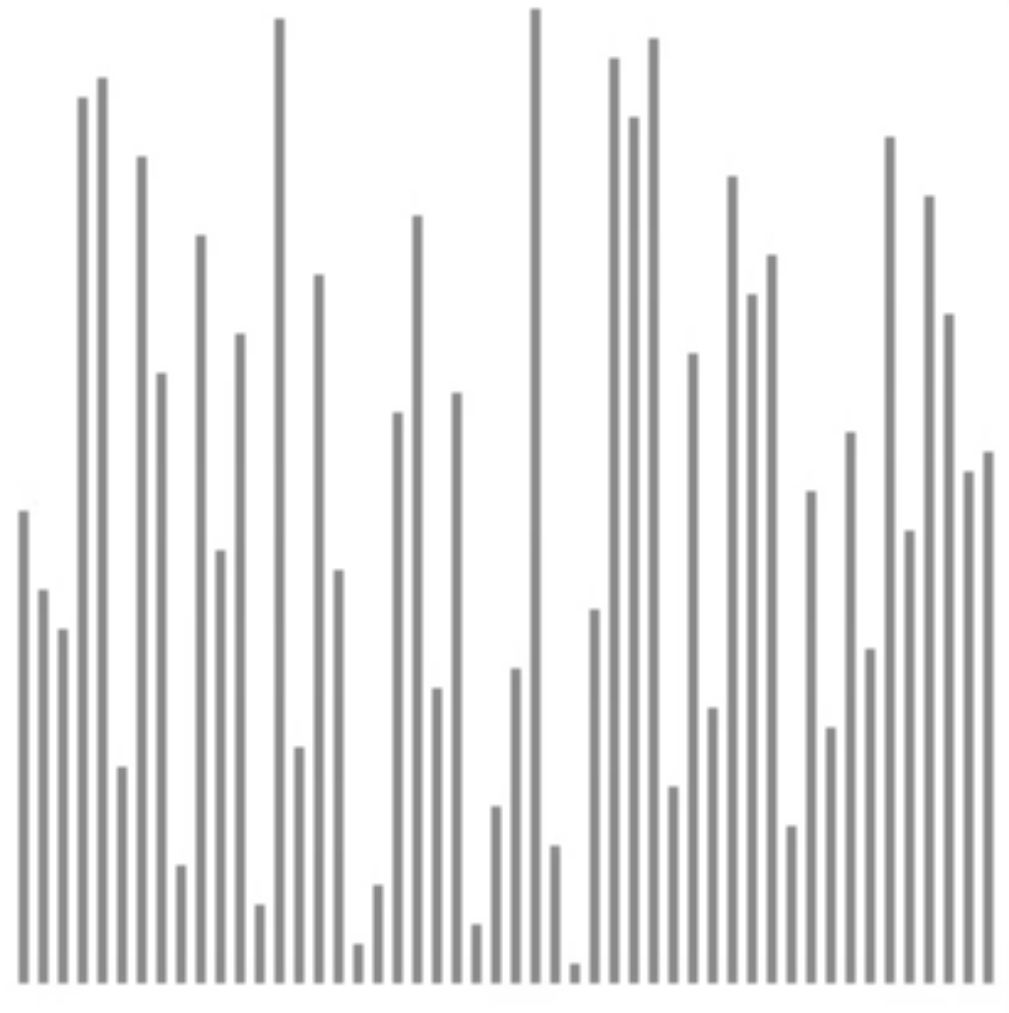
Quicksort trace



Quicksort trace (array contents after each partition)

Quicksort animation

50 random items



algorithm position

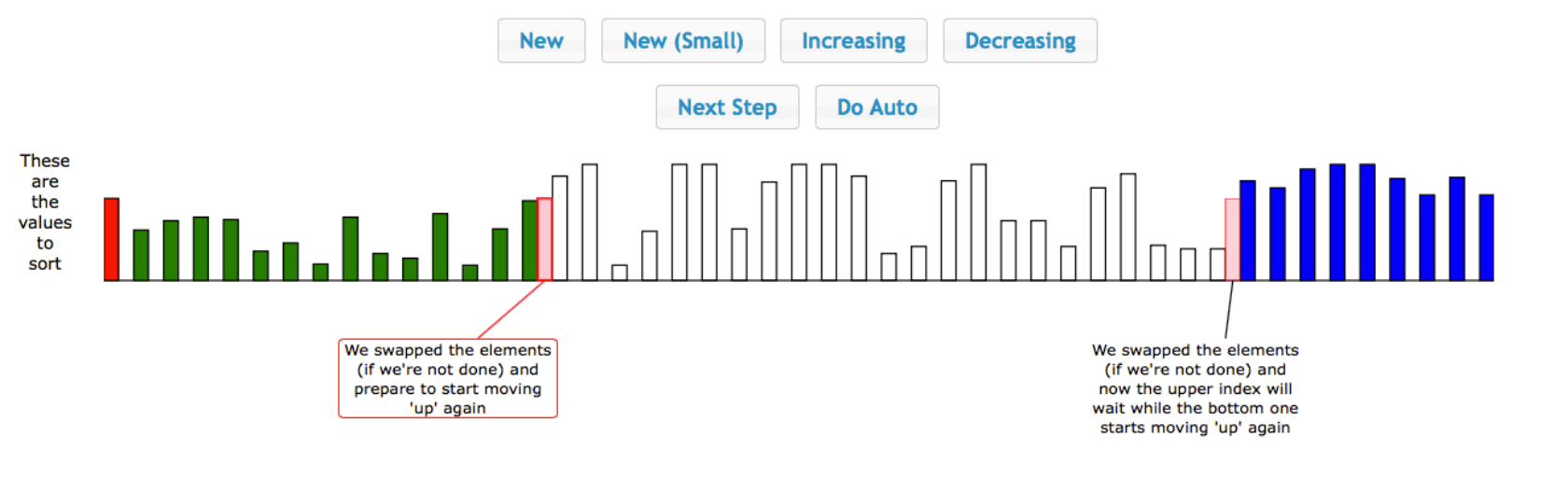
in order

current subarray

not in order

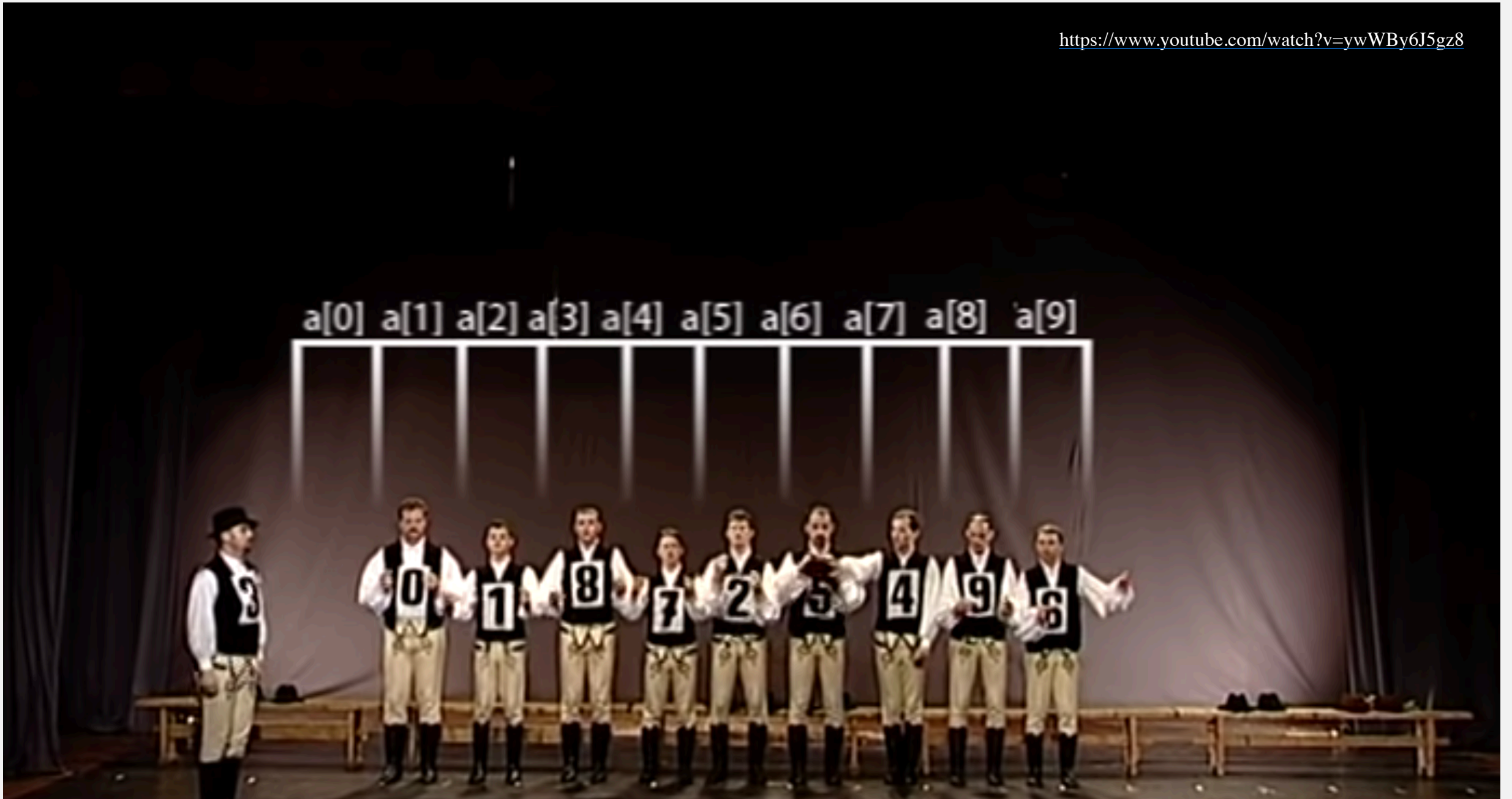
http://www.sorting-algorithms.com/quick-sort

The music of quicksort partitioning (by Brad Lyon)



https://learnforeverlearn.com/pivot_music/

QuickSort Dancing



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: implementation details

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

Terminating the loop. Testing whether the pointers cross is trickier than it might seem.

Equal keys. When duplicates are present, it is (counter-intuitively) better to stop scans on keys equal to the partitioning item's key.

Preserving randomness. Shuffling is needed for performance guarantee.

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

Quicksort: empirical analysis (1961)

Running time estimates:

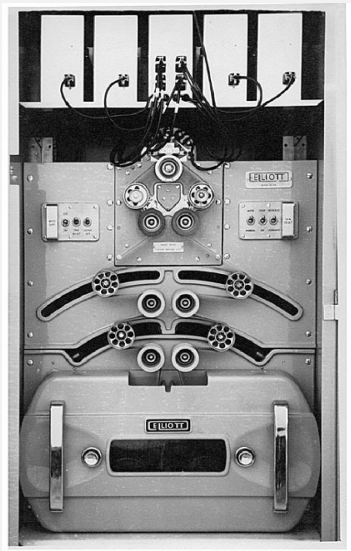
- Algol 60 implementation.
- National-Elliott 405 computer.

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 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

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

Quicksort: summary of performance characteristics

Quicksort is a (Las Vegas) randomized algorithm. https://en.wikipedia.org/wiki/Las_Vegas_algorithm

- Guaranteed to be correct.
- Running time depends on random shuffle.

Average case. Expected number of compares is $\sim 1.39\, N \lg N$.

- 39% more compares than mergesort.
- Faster than mergesort in practice because of less data movement.

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

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

[but more likely that lightning bolt strikes computer during execution]

QuickSort: summary

Quicksort is an in-place sorting algorithm.

- Partitioning: constant extra space.
- Depth of recursion: logarithmic extra space (with high probability).

Comparison sort?	Comparison
Time Complexity	O(N log N)
Space Complexity	Inplace
Internal or External?	Internal
Recursive / Non-recursive?	Recursive
Stable	Not stable

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items
- Can improve the running time by 10-20%

```
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.

~ 12/7 N ln N compares (14% less)

~ 12/35 N ln N exchanges (3% more)

```
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);
}
```

Java system sorts

Arrays.sort().

- Has different methods for each primitive type
- Uses both quick sort and mergesort
- Uses quick sort for primitive types
- MergeSort for objects

Q. Why use different algorithms for primitive and reference types?

- judgement that if programmer is using objects then space might not be critical point (so extra space that mergesort uses not an issue)

- if programmer using primitive types then performance key

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]

...

http://mail.openjdk.java.net/pipermail/core-libs-dev/2009-September/002630.html

Which sorting algorithm to use?

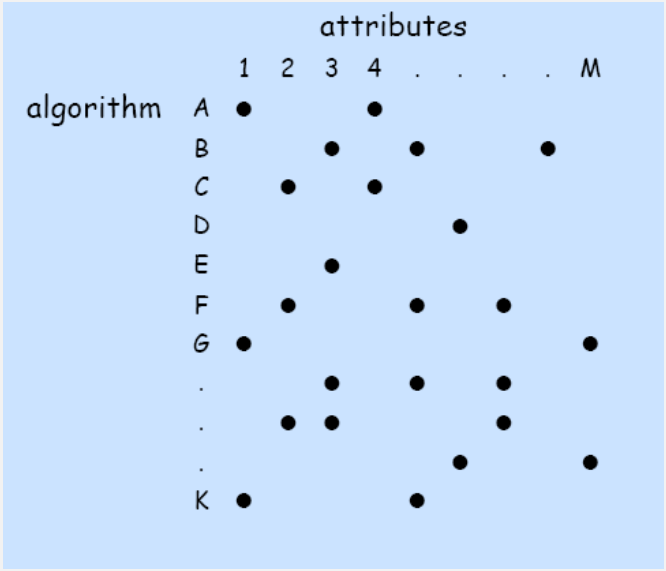
Many sorting algorithms to choose from:

sorts	algorithms
elementary sorts	insertion sort, selection sort, bubblesort, shaker sort, ...
subquadratic sorts	quicksort, mergesort, heapsort, shellsort, samplesort, ...
system sorts	dual-pivot quicksort, timsort, 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 to use?

Applications have diverse attributes.

- Stable?
- Parallel?
- In-place?
- Deterministic?
- Duplicate keys?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Randomly-ordered array?
- Guaranteed performance?



many more combinations of attributes than algorithms

Q. Is the system sort good enough?

A. Usually.

Sorting summary

	inplace?	stable?	best	average	worst	remarks
selection	✔		N^2	N^2	N^2	N exchanges
insertion	✔	✔	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✔		$N \log N$	?	$c\, N^{3/2}$	tight code; subquadratic
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
?	✔	✔	N	$N \lg N$	$N \lg N$	holy sorting grail