

2.3 QUICKSORT

Two classic sorting algorithms

Critical components in the world's computational infrastructure.

- Full scientific understanding of their properties has enabled us to develop them into practical system sorts.
- Quicksort honored as one of top 10 algorithms of 20th century in science and engineering.

Mergesort.

← last lecture

- Java sort for objects.
- Perl, C++ stable sort, Python stable sort, Firefox JavaScript, ...

Quicksort.

← this lecture

- Java sort for primitive types.
- C qsort, Unix, Visual C++, Python, Matlab, Chrome JavaScript, ...

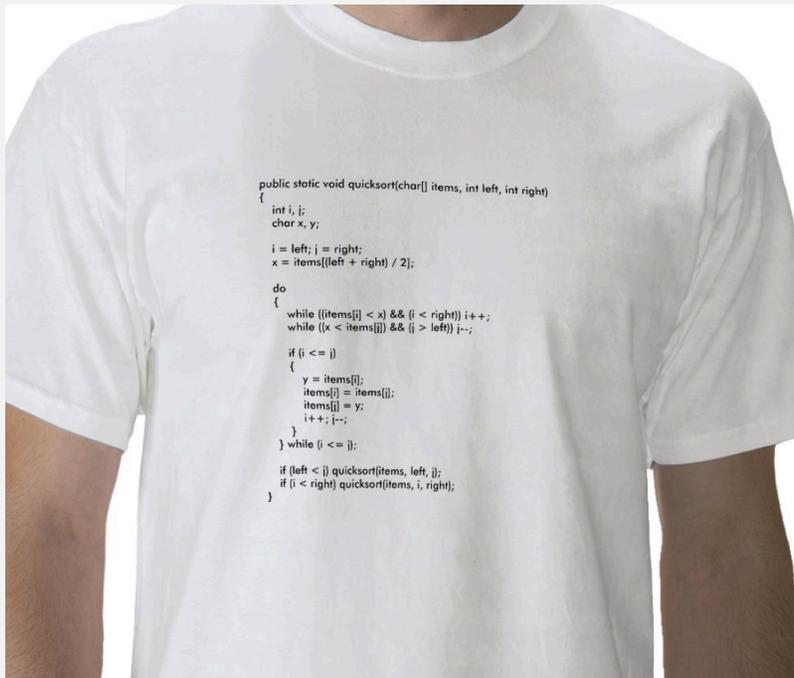


- ▶ quicksort
- ▶ selection
- ▶ duplicate keys
- ▶ system sorts

Algorithms, 4th Edition · Robert Sedgwick and Kevin Wayne · Copyright © 2002–2011 · October 4, 2011 4:54:01 AM

2

Quicksort t-shirt



3

- ▶ quicksort
- ▶ selection
- ▶ duplicate keys
- ▶ system sorts

4

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 piece recursively.



Sir Charles Antony Richard Hoare
1980 Turing Award

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

5

Quicksort partitioning demo

Quicksort partitioning

Basic plan.

- Scan i from left for an item that belongs on the right.
- Scan j from right for an item that belongs on the left.
- Exchange $a[i]$ and $a[j]$.
- Repeat until pointers cross.

	i	j	v	$a[i]$
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	K	E C A I E K L P U T M Q R X O S
result	6	5	K	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)

7

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]))           find item on left to swap
            if (i == hi) break;

        while (less(a[lo], a[--j]))           find item on right to swap
            if (j == lo) break;

        if (i >= j) break;                    check if pointers cross
        exch(a, i, j);                          swap
    }

    exch(a, lo, j);                            swap with partitioning item
    return j;                                   return index of item now known to be in place
}
```



8

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)

9

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

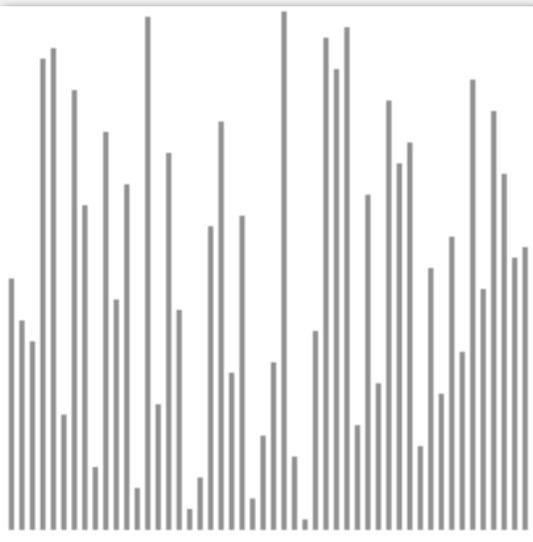
no partition for subarrays of size 1

Quicksort trace (array contents after each partition)

10

Quicksort animation

50 random items



▲ algorithm position
 in order
 — current subarray
 not in order

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

11

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 a bit trickier than it might seem.

Staying in bounds. The $(j == lo)$ test is redundant (why?), but the $(i == hi)$ test is not.

Preserving randomness. Shuffling is needed for performance guarantee.

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

12

Quicksort: empirical analysis

Running time estimates:

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

computer	insertion sort (N^2)			mergesort ($N \log N$)			quicksort ($N \log N$)		
	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.

13

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	0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2	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	4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6	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	I	J	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	8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10	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	12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14	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

14

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	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

15

Quicksort: average-case analysis

Proposition. The average number of compares C_N to quicksort an array of N distinct keys is $\sim 2N \ln N$ (and the number of exchanges is $\sim \frac{1}{3} N \ln N$).

Pf 1. C_N satisfies the recurrence $C_0 = C_1 = 0$ and for $N \geq 2$:

$$C_N = \underbrace{(N+1)}_{\text{partitioning}} + \underbrace{\frac{C_0 + C_1 + \dots + C_{N-1}}{N}}_{\text{left}} + \underbrace{\frac{C_{N-1} + C_{N-2} + \dots + C_0}{N}}_{\text{right}} \underbrace{\leftarrow}_{\text{partitioning probability}}$$

- Multiply both sides by N and collect terms:

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

- Subtract this from the same equation for $N-1$:

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

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

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

16

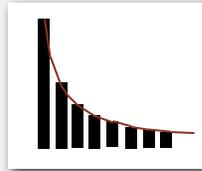
Quicksort: average-case analysis

- Repeatedly apply above equation:

$$\begin{aligned}
 \frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\
 &\stackrel{\text{previous equation}}{=} \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \quad \leftarrow \text{substitute previous equation} \\
 &= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\
 &= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \dots + \frac{2}{N+1}
 \end{aligned}$$

- Approximate sum by an integral:

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



- Finally, the desired result:

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

17

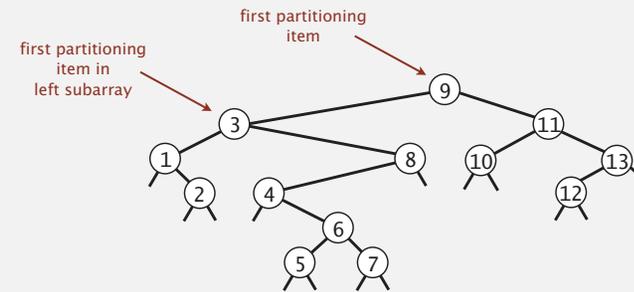
Quicksort: average-case analysis

Proposition. The average number of compares C_N to quicksort an array of N distinct keys is $\sim 2N \ln N$ (and the number of exchanges is $\sim \frac{1}{3} N \ln N$).

Pf 2. Consider BST representation of keys 1 to N .

shuffle

9	10	2	5	8	7	6	1	11	12	13	3	4
---	----	---	---	---	---	---	---	----	----	----	---	---



18

Quicksort: average-case analysis

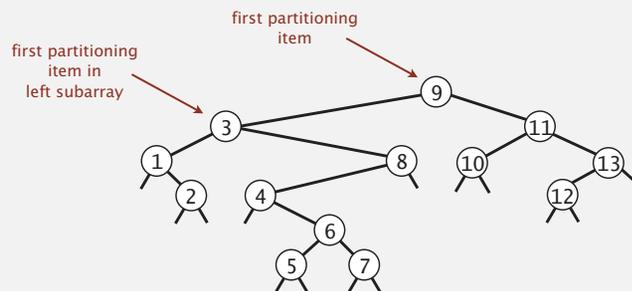
Proposition. The average number of compares C_N to quicksort an array of N distinct keys is $\sim 2N \ln N$ (and the number of exchanges is $\sim \frac{1}{3} N \ln N$).

Pf 2. Consider BST representation of keys 1 to N .

- A key is compared only with its ancestors and descendants.
- Probability i and j are compared equals $2 / |j - i + 1|$.

3 and 6 are compared
(when 3 is partition)

1 and 6 are not compared
(because 3 is partition)



19

Quicksort: average-case analysis

Proposition. The average number of compares C_N to quicksort an array of N distinct keys is $\sim 2N \ln N$ (and the number of exchanges is $\sim \frac{1}{3} N \ln N$).

Pf 2. Consider BST representation of keys 1 to N .

- A key is compared only with its ancestors and descendants.
- Probability i and j are compared equals $2 / |j - i + 1|$.

$$\begin{aligned}
 \text{Expected number of compares} &= \sum_{i=1}^N \sum_{j=i+1}^N \frac{2}{j-i+1} = 2 \sum_{i=1}^N \sum_{j=2}^{N-i+1} \frac{1}{j} \\
 &\stackrel{\text{all pairs } i \text{ and } j}{\leq} 2N \sum_{j=1}^N \frac{1}{j} \\
 &\sim 2N \int_{x=1}^N \frac{1}{x} dx \\
 &= 2N \ln N
 \end{aligned}$$

20

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.

- $N + (N - 1) + (N - 2) + \dots + 1 \sim \frac{1}{2}N^2$.
- More likely that your computer is struck by lightning bolt.

Average case. Number of compares is $\sim 1.39 N \lg N$.

- 39% more compares than mergesort.
- **But** faster than mergesort in practice because of less data movement.

Random shuffle.

- Probabilistic guarantee against worst case.
- Basis for math model that can be validated with experiments.

Caveat emptor. Many textbook implementations go **quadratic** if array

- Is sorted or reverse sorted.
- Has many duplicates (even if randomized!)

21

Quicksort: practical improvements

Insertion sort small subarrays.

- Even quicksort has too much overhead for tiny subarrays.
- Cutoff to insertion sort for ≈ 10 items.
- Note: could delay insertion sort until one pass at end.

```
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);
}
```

22

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 (slightly fewer)
~ 12/35 $N \ln N$ exchanges (slightly more)

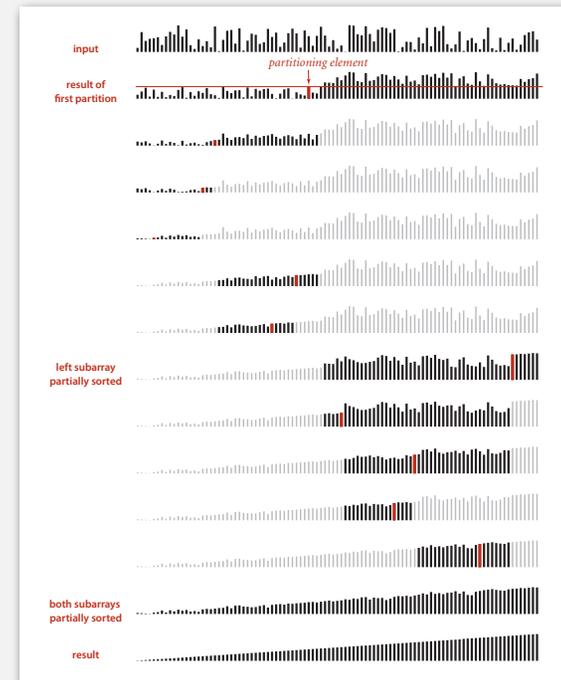
```
private static void sort(Comparable[] a, int lo, int hi)
{
    if (hi <= lo) return;

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

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

23

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



24

- ▶ quicksort
- ▶ selection
- ▶ duplicate keys
- ▶ system sorts

Selection

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

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

Which is true?

- $\Omega(N \log N)$ lower bound? ← is selection as hard as sorting?
- $O(N)$ upper bound? ← is there a linear-time algorithm for each k ?

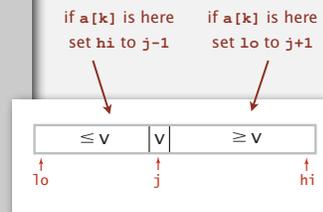
Quick-select

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: mathematical analysis

Proposition. Quick-select takes **linear** time on average.

Pf sketch.

- Intuitively, each partitioning step splits array approximately in half:
 $N + N/2 + N/4 + \dots + 1 \sim 2N$ compares.
- Formal analysis similar to quicksort analysis yields:

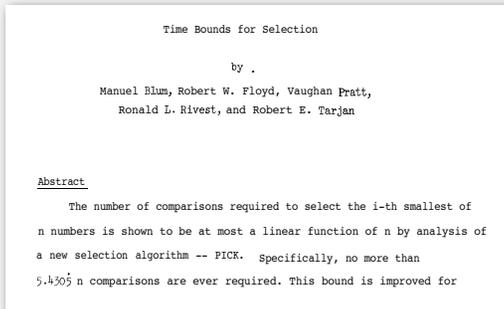
$$C_N = 2N + k \ln(N/k) + (N-k) \ln(N/(N-k))$$

(2 + 2 ln 2) N to find the median

Remark. Quick-select uses $\sim \frac{1}{2} N^2$ compares in the worst case, but (as with quicksort) the random shuffle provides a probabilistic guarantee.

Theoretical context for selection

Proposition. [Blum, Floyd, Pratt, Rivest, Tarjan, 1973] There exists a compare-based selection algorithm whose worst-case running time is linear.



Remark. But, constants are too high \Rightarrow not used in practice.

Use theory as a guide.

- Still worthwhile to seek **practical** linear-time (worst-case) algorithm.
- Until one is discovered, use quick-select if you don't need a full sort.

29

Generic methods

In our `select()` implementation, client needs a cast.

```
Double[] a = new Double[N];
for (int i = 0; i < N; i++)
    a[i] = StdRandom.uniform();
Double median = (Double) Quick.select(a, N/2);
```

unsafe cast
required in client

The compiler complains.

```
% javac Quick.java
Note: Quick.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Q. How to fix?

30

Generic methods

Pedantic (safe) version. Compiles cleanly, no cast needed in client.

```
public class QuickPedantic
{
    public static <Key extends Comparable<Key>> Key select(Key[] a, int k)
    { /* as before */ }

    public static <Key extends Comparable<Key>> void sort(Key[] a)
    { /* as before */ }

    private static <Key extends Comparable<Key>> int partition(Key[] a, int lo, int hi)
    { /* as before */ }

    private static <Key extends Comparable<Key>> boolean less(Key v, Key w)
    { /* as before */ }

    private static <Key extends Comparable<Key>> void exch(Key[] a, int i, int j)
    { Key swap = a[i]; a[i] = a[j]; a[j] = swap; }
}
```

generic type variable
(value inferred from argument a[])

return type matches array type

can declare variables of generic type

<http://www.cs.princeton.edu/algs4/23quicksort/QuickPedantic.java.html>

Remark. Obnoxious code needed in system sort; not in this course (for brevity).

31

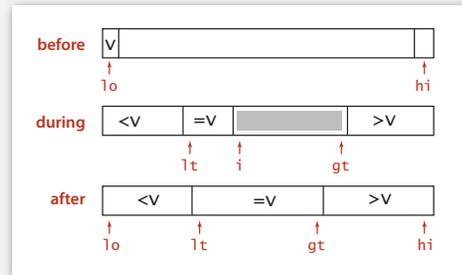
- quicksort
- selection
- duplicate keys
- system sorts

32

Dijkstra 3-way partitioning algorithm

3-way partitioning.

- Let v be partitioning item $a[lo]$.
- Scan i from left to right.
 - $a[i]$ less than v : exchange $a[lt]$ with $a[i]$ and increment both lt and i
 - $a[i]$ greater than v : exchange $a[gt]$ with $a[i]$ and decrement gt
 - $a[i]$ equal to v : increment i



Most of the right properties.

- In-place.
- Not much code.
- Linear time if keys are all equal.

37

Dijkstra's 3-way partitioning: demo

38

Dijkstra's 3-way partitioning: trace

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

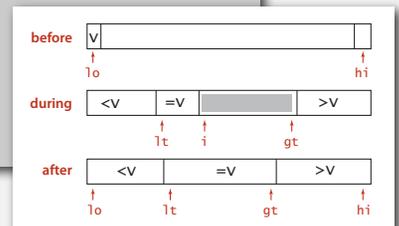
3-way partitioning trace (array contents after each loop iteration)

39

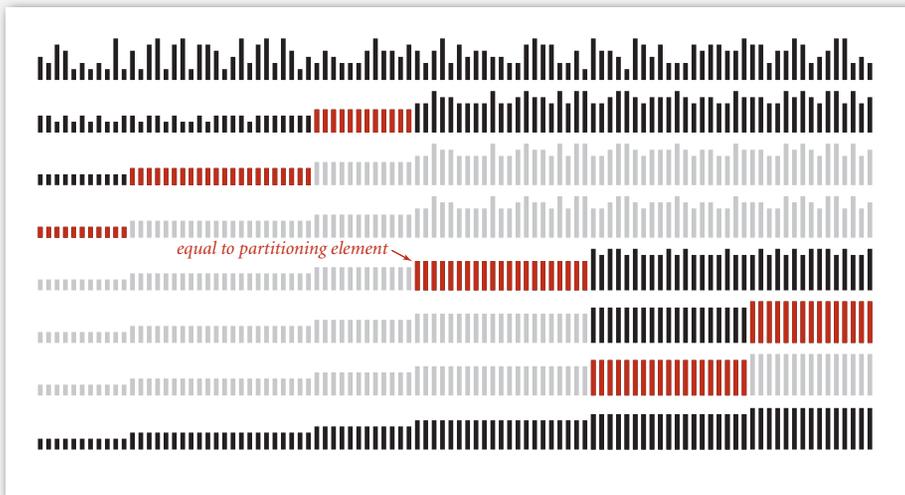
3-way quicksort: Java implementation

```
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);
}
}
```



40



Sorting lower bound. If there are n distinct keys and the i^{th} one occurs x_i times, any compare-based sorting algorithm must use at least

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

compares in the worst case.

$N \lg N$ when all distinct;
linear when only a constant number of distinct keys

proportional to lower bound

Proposition. [Sedgwick-Bentley, 1997]

Quicksort with 3-way partitioning is entropy-optimal.

Pf. [beyond scope of course]

Bottom line. Randomized quicksort with 3-way partitioning reduces running time from linearithmic to linear in broad class of applications.

- ▶ selection
- ▶ duplicate keys
- ▶ comparators
- ▶ system sorts

Sorting applications

Sorting algorithms are essential in a broad variety of applications:

- Sort a list of names.
 - Organize an MP3 library.
 - Display Google PageRank results.
 - List RSS feed in reverse chronological order.
- obvious applications
-
- Find the median.
 - Find the closest pair.
 - Binary search in a database.
 - Identify statistical outliers.
 - Find duplicates in a mailing list.
- problems become easy once items are in sorted order
-
- Data compression.
 - Computer graphics.
 - Computational biology.
 - Supply chain management.
 - Load balancing on a parallel computer.
- non-obvious applications

Every system needs (and has) a system sort!

Java uses both mergesort and quicksort.

- `Arrays.sort()` sorts an array of `Comparable` or of any primitive type.
- Uses tuned quicksort for primitive types; tuned mergesort for objects.

```
import java.util.Arrays;

public class StringSort
{
    public static void main(String[] args)
    {
        String[] a = StdIn.readAll().split("\\s+");
        Arrays.sort(a);
        for (int i = 0; i < N; i++)
            StdOut.println(a[i]);
    }
}
```

Q. Why use different algorithms, depending on type?

45

AT&T Bell Labs (1991). Allan Wilks and Rick Becker discovered that a `qsort()` call that should have taken a few minutes was consuming hours of CPU time.



At the time, almost all `qsort()` implementations based on those in:

- Version 7 Unix (1979): quadratic time to sort organ-pipe arrays.
- BSD Unix (1983): quadratic time to sort random arrays of 0s and 1s.

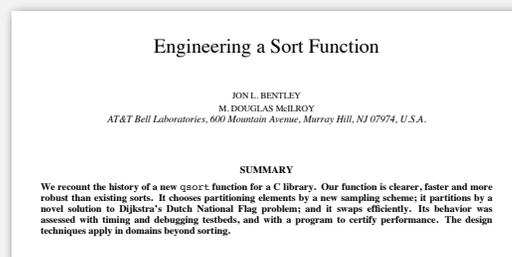


46

Engineering a system sort

Basic algorithm = quicksort.

- Cutoff to insertion sort for small subarrays.
- Partitioning scheme: Bentley-McIlroy 3-way partitioning. [ahead]
- Partitioning item.
 - small arrays: middle entry
 - medium arrays: median of 3
 - large arrays: Tukey's ninther [next slide]



Now widely used. C, C++, Java,

47

Tukey's ninther

Tukey's ninther. Median of the median of 3 samples, each of 3 entries.

- Approximates the median of 9.
- Uses at most 12 compares.



nine evenly spaced entries	R	L	A	P	M	C	G	A	X	Z	K	R	B	R	J	J	E
groups of 3	R	A	M	G	X	K	B	J	E								
medians	M	K	E														
ninther	K																

Q. Why use Tukey's ninther?

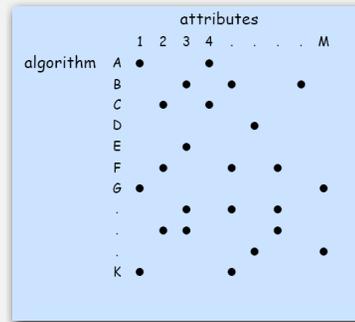
A. Better partitioning than random shuffle and less costly.

48

System sort: Which algorithm to use?

Applications have diverse attributes.

- Stable?
- Parallel?
- Deterministic?
- Keys all distinct?
- Multiple key types?
- Linked list or arrays?
- Large or small items?
- Is your array randomly ordered?
- Need guaranteed performance?



many more combinations of attributes than algorithms

Elementary sort may be method of choice for some combination.
Cannot cover **all** combinations of attributes.

- Q. Is the system sort good enough?
A. Usually.

Sorting summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	N exchanges
insertion	x	x	$N^2 / 2$	$N^2 / 4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
merge		x	$N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee, stable
quick	x		$N^2 / 2$	$2 N \ln N$	$N \lg N$	$N \log N$ probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	$2 N \ln N$	N	improves quicksort in presence of duplicate keys
???	x	x	$N \lg N$	$N \lg N$	$N \lg N$	holy sorting grail

Which sorting algorithm?

lifo	find	data	data	data	data	hash	data
fifo	fifo	fifo	fifo	exch	fifo	fifo	exch
data	data	find	find	fifo	lifo	data	fifo
type	exch	hash	hash	find	type	link	find
hash	hash	heap	heap	hash	hash	leaf	hash
heap	heap	lifo	lifo	heap	heap	heap	heap
sort	less	link	link	leaf	link	exch	leaf
link	left	list	list	sort	node	link	left
list	leaf	push	push	less	find	lifo	less
push	lifo	root	root	lifo	list	left	lifo
find	push	sort	sort	link	push	find	link
root	root	type	type	list	root	path	list
leaf	list	leaf	leaf	sort	leaf	list	next
tree	tree	left	tree	tree	null	next	node
null	null	node	null	null	path	less	null
path	path	null	path	path	tree	root	path
node	node	path	node	node	exch	sink	push
left	link	tree	left	type	left	swim	root
less	sort	exch	less	root	less	null	sink
exch	type	less	exch	push	node	sort	sort
sink	sink	next	sink	sink	next	type	swap
swim	swim	sink	swim	swim	sink	tree	swim
next	next	swap	next	next	swap	push	tree
swap	swap	swim	swap	swap	swim	swap	type

original ? ? ? ? ? ? sorted

Which sorting algorithm?

lifo	find	data	data	data	data	hash	data
fifo	fifo	fifo	fifo	exch	fifo	fifo	exch
data	data	find	find	fifo	lifo	data	fifo
type	exch	hash	hash	find	type	link	find
hash	hash	heap	heap	hash	hash	leaf	hash
heap	heap	lifo	lifo	heap	heap	heap	heap
sort	less	link	link	leaf	link	exch	leaf
link	left	list	list	left	sort	node	left
list	leaf	push	push	less	find	lifo	less
push	lifo	root	root	lifo	list	left	lifo
find	push	sort	sort	link	push	find	link
root	root	type	type	list	root	path	list
leaf	list	leaf	leaf	sort	leaf	list	next
tree	tree	left	tree	tree	null	next	node
null	null	node	null	null	path	less	null
path	path	null	path	path	tree	root	path
node	node	path	node	node	exch	sink	push
left	link	tree	left	type	left	swim	root
less	sort	exch	less	root	less	null	sink
exch	type	less	exch	push	node	sort	sort
sink	sink	next	sink	sink	next	type	swap
swim	swim	sink	swim	swim	sink	tree	swim
next	next	swap	next	next	swap	push	tree
swap	swap	swim	swap	swap	swim	swap	type

original quicksort mergesort insertion selection merge BU shellsort sorted