



<http://algs4.cs.princeton.edu>

2.3 QUICKSORT

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

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?

Which is true?

- $N \log N$ lower bound?  is selection as hard as sorting?
- N upper bound?  is there a linear-time algorithm?

Quick-select

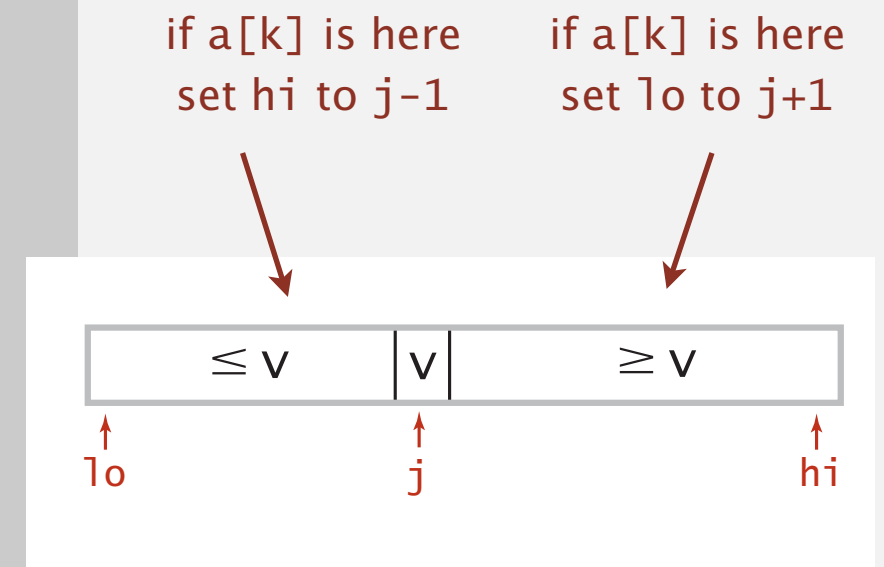
Partition array so that:

- 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

$k = 5$

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:

$$\begin{aligned} C_N &= 2N + 2k \ln(N/k) + 2(N-k) \ln(N/(N-k)) \\ &\leq (2 + 2 \ln 2) N \end{aligned}$$

- Ex: $(2 + 2 \ln 2) N \approx 3.38 N$ compares to find median ($k = N/2$).



<http://algs4.cs.princeton.edu>

2.3 QUICKSORT

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

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

War story (system sort in C)

A beautiful bug report. [Allan Wilks and Rick Becker, 1991]

We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

```
main (int argc, char**argv) {
    int n = atoi(argv[1]), i, x[100000];
    for (i = 0; i < n; i++)
        x[i] = i;
    for ( ; i < 2*n; i++)
        x[i] = 2*n-i-1;
    qsort(x, 2*n, sizeof(int), intcmp);
}
```

Here are the timings on our machine:

```
$ time a.out 2000
real    5.85s
$ time a.out 4000
real    21.64s
$ time a.out 8000
real    85.11s
```

War story (system sort in C)

Bug. A `qsort()` call that should have taken seconds was taking minutes.



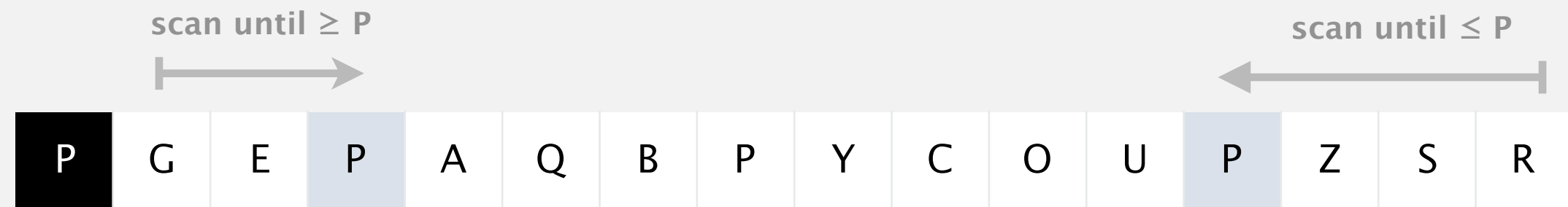
At the time, almost all `qsort()` implementations based on those in:

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

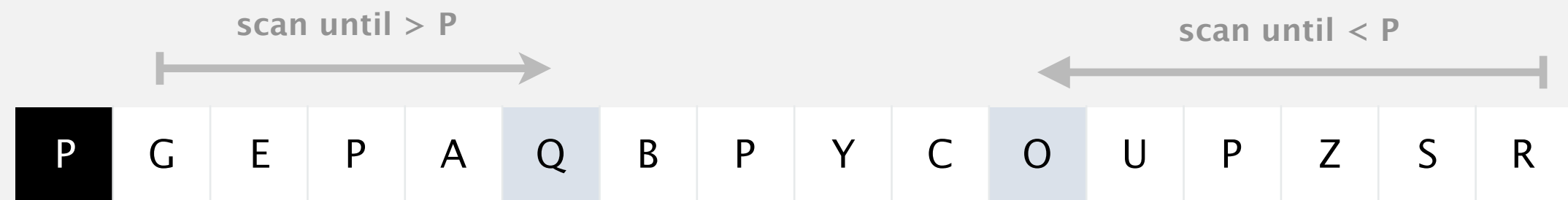


Duplicate keys: stop on equal keys

Our partitioning subroutine stops both scans on equal keys.



Q. Why not continue scans on equal keys?



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: partitioning strategies

Bad. Don't stop scans on equal keys.

[$\sim \frac{1}{2} N^2$ 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

Good. Stop scans on equal keys.

[$\sim N \lg N$ 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

Better. Put all equal keys in place. How?

[$\sim N$ compares when all keys equal]

A A A **B B B B B** C C C

A A A A A A A A A A A




<http://algs4.cs.princeton.edu>

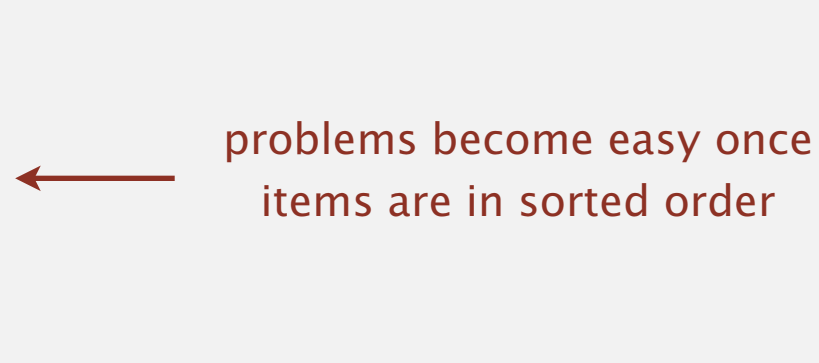
2.3 QUICKSORT

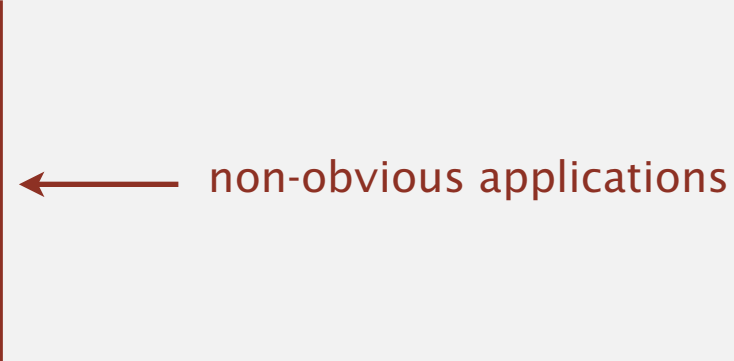
- ▶ *quicksort*
- ▶ *selection*
- ▶ *duplicate keys*
- ▶ *system sorts*

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.
- 
- A vertical red line is positioned to the right of the first four list items. A horizontal red arrow points from the text 'obvious applications' to this line.

- Find the median.
 - Identify statistical outliers.
 - Binary search in a database.
 - Find duplicates in a mailing list.
- 
- A vertical red line is positioned to the right of the last four list items. A horizontal red arrow points from the text 'problems become easy once items are in sorted order' to this line.

- Data compression.
 - Computer graphics.
 - Computational biology.
 - Load balancing on a parallel computer.
- 
- A vertical red line is positioned to the right of the last four list items. A horizontal red arrow points from the text 'non-obvious applications' to this line.

. . .

System sort in Java 7

`Arrays.sort()`.

- Has one method for objects that are Comparable.
- Has an overloaded method for each primitive type.
- Has an overloaded method for use with a Comparator.
- Has overloaded methods for sorting subarrays.



Algorithms.

- Dual-pivot quicksort for primitive types.
- Timsort for reference types.

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

Bottom line. Use the system sort!

Ineffective sorts

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):  
  IF LENGTH(LIST) < 2:  
    RETURN LIST  
  PIVOT = INT(LENGTH(LIST) / 2)  
  A = HALFHEARTEDMERGESORT(LIST[:PIVOT])  
  B = HALFHEARTEDMERGESORT(LIST[PIVOT:])  
  // UMMMMM  
  RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):  
  // AN OPTIMIZED BOGOSORT  
  // RUNS IN O(N LOG N)  
  FOR N FROM 1 TO LOG(LENGTH(LIST)):  
    SHUFFLE(LIST):  
    IF ISSORTED(LIST):  
      RETURN LIST  
  RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):  
  OK SO YOU CHOOSE A PIVOT  
  THEN DIVIDE THE LIST IN HALF  
  FOR EACH HALF:  
    CHECK TO SEE IF IT'S SORTED  
    NO, WAIT, IT DOESN'T MATTER  
    COMPARE EACH ELEMENT TO THE PIVOT  
    THE BIGGER ONES GO IN A NEW LIST  
    THE EQUAL ONES GO INTO, UH  
    THE SECOND LIST FROM BEFORE  
    HANG ON, LET ME NAME THE LISTS  
    THIS IS LIST A  
    THE NEW ONE IS LIST B  
    PUT THE BIG ONES INTO LIST B  
    NOW TAKE THE SECOND LIST  
    CALL IT LIST, UH, A2  
    WHICH ONE WAS THE PIVOT IN?  
    SCRATCH ALL THAT  
    IT JUST RECURSIVELY CALLS ITSELF  
    UNTIL BOTH LISTS ARE EMPTY  
    RIGHT?  
    NOT EMPTY, BUT YOU KNOW WHAT I MEAN  
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):  
  IF ISSORTED(LIST):  
    RETURN LIST  
  FOR N FROM 1 TO 10000:  
    PIVOT = RANDOM(0, LENGTH(LIST))  
    LIST = LIST[PIVOT:] + LIST[:PIVOT]  
    IF ISSORTED(LIST):  
      RETURN LIST  
  IF ISSORTED(LIST):  
    RETURN LIST  
  IF ISSORTED(LIST): // THIS CAN'T BE HAPPENING  
    RETURN LIST  
  IF ISSORTED(LIST): // COME ON COME ON  
    RETURN LIST  
  // OH JEEZ  
  // I'M GONNA BE IN SO MUCH TROUBLE  
  LIST = [ ]  
  SYSTEM("SHUTDOWN -H +5")  
  SYSTEM("RM -RF ./")  
  SYSTEM("RM -RF ~/*")  
  SYSTEM("RM -RF /")  
  SYSTEM("RD /S /Q C:\*") // PORTABILITY  
  RETURN [1, 2, 3, 4, 5]
```

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
shell	✓		$N \log_3 N$?	$c N^{3/2}$	tight code; subquadratic
merge		✓	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
timsort		✓	N	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
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



<http://algs4.cs.princeton.edu>

1.4 ANALYSIS OF ALGORITHMS

- ▶ *introduction*
- ▶ *observations*
- ▶ *mathematical models*
- ▶ *order-of-growth classifications*
- ▶ *theory of algorithms*

Commonly-used notations in the theory of algorithms

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10 N^2$	$10 N^2$ $10 N^2 + 22 N \log N$ $10 N^2 + 2 N + 37$	provide approximate model
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$	develop lower bounds