Algorithms ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

2.3 QUICKSORT

- *quicksort*
- *practical improvements*
- *defeating quicksort (optional)*

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.

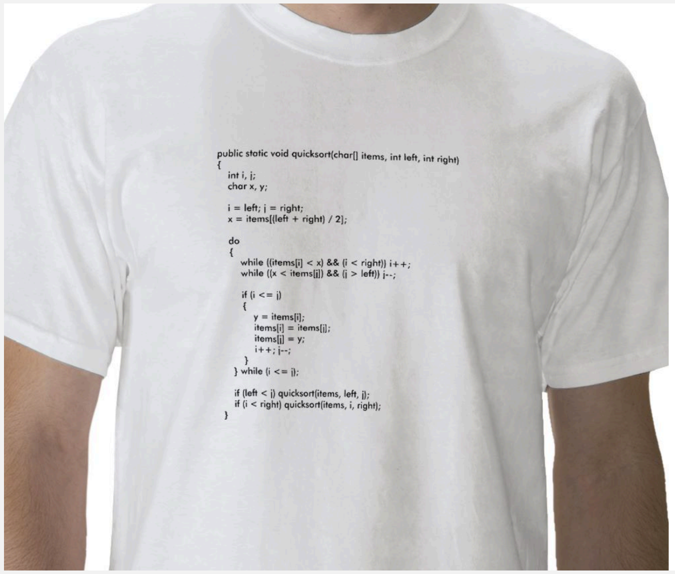
Mergesort. ← last lecture

- 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 t-shirt



http://algs4.cs.princeton.edu

2.3 QUICKSORT

- *quicksort*
- *practical improvements*
- *defeating quicksort (optional)*

Partitioning - the heart of Quicksort

Partitioning.

- To Partition an array a[] on element x=a[i] is to rearrange it such that
  - x moves to position j (may be the same as i)
  - All entries to the left of x are <= x.
  - All entries to the right of x are >= x.

5	550	10	4	10	9	330

A.

4	5	9	10	10	330	550

B.

5	4	9	10	10	550	330

C.

5	9	10	4	10	550	330

D.

5	9	10	4	10	550	330

pollEv.com/jhug text to **37607**

Q: Which partitions are valid?
A: [103370]
A, B: [103400]
A, B, C: [103407]
A, B, C, D: [103424]
NONE: [103425]

On board

- Partition based sorting (a.k.a. quicksort)?
- How do we partition?

Partition!

2	4	5	3	1	6	7	8

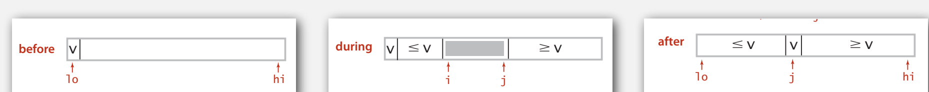
1	2	5	3	4	[redacted]		

pollEv.com/jhug text to **37607**

Q: How many total compares were made while partitioning on 2?
A. 3 [104701]
B. 4 [104734]
C. 5 [104736]
D. 6 [104757]
E. 7 [104765]

Quicksort: Java code for partitioning

```
private static int partition(Comparable[] a, int lo, int hi)
{
   int i = lo, j = hi+1;
   while (i < j)
   {
      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
}
```



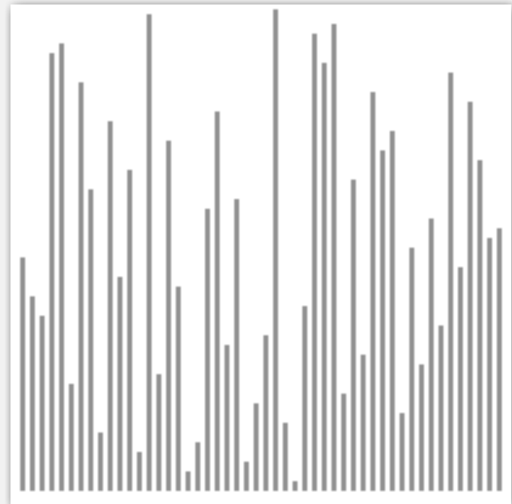
Quicksort: Java implementation

```
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 animation

50 random items



algorithm position
in order
current subarray
not in order

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

Compare analysis

On board

- Best case
- 9/10ths case
- Worst case
- Average case recurrence relation

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: best-case analysis

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

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

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

partitioning probability

- Multiply both sides by N and collect terms:

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

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

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

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

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

Quicksort: average-case analysis

- Repeatedly apply above equation:

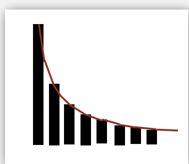
$$\begin{aligned}\frac{C_N}{N+1} &= \frac{C_{N-1}}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \frac{C_{N-3}}{N-2} + \frac{2}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \ldots + \frac{2}{N+1}\end{aligned}$$

previous equation

substitute previous equation

- Approximate sum by an integral:

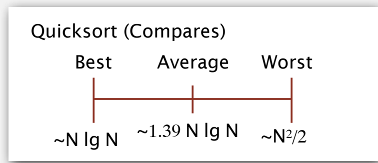
$$\begin{aligned}C_N &= 2(N+1)\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots \frac{1}{N+1}\right) \\ &\sim 2(N+1)\int_3^{N+1} \frac{1}{x}\,dx\end{aligned}$$



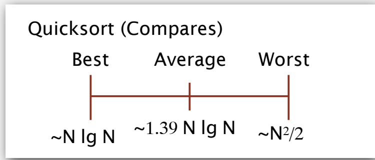
- Finally, the desired result:

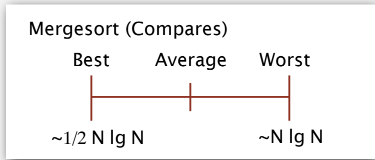
$$C_N \sim 2(N+1)\ln N \approx 1.39 N \lg N$$

Sounds of sorting



Quicksort performance





Preserving randomness. Shuffling provides probabilistic guarantee of average case behavior.

- More compares than Mergesort.

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: 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.

Preserving randomness. Shuffling is needed for performance guarantee.

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

Equal Keys.

pollEv.com/jhug text to **37607**

Q: What is the result of partitioning:

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

Correct Answer: The middle one!

- 8 total compares
- 3 swaps involving non-pivot As
  - {1, 6}, {2, 5}, {3, 4}
- 1 swap involving the pivot {1, 4}

Equal Keys.

pollEv.com/jhug text to **37607**

Q: What is the result of partitioning if we do not stop on equal values?

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

Correct Answer: The top one!

- At least 12 total compares
  - Precise number depends on code
- 1 swap involving the pivot and itself

Total running time?

- Order of growth: N^2

Quicksort: summary of performance characteristics

Worst case. Number of compares is quadratic.

- $N + (N - 1) + (N - 2) + \ldots + 1 \sim \frac{1}{2} N^2$.
- 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!)

Quicksort properties

Proposition. Quicksort is an in-place sorting algorithm.

Pf.

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

can guarantee logarithmic depth by recurring on smaller subarray before larger subarray

Proposition. Quicksort is not stable.

Pf.

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

COS226: Quicksort vs. Mergesort

algorithm	Mergesort	Quicksort
Recursion	Before doing work	Do work first
Deterministic	Yes	No
Compares (worst)	~ N lg N	~N^2 / 2
Compares (average)	[redacted]	~ 1.39 N lg N
Exchanges (average)	N/A	~ 0.23 N lg N
Stable	Yes	No
Memory Use	N	In-place
Overall Performance	Worse	Better

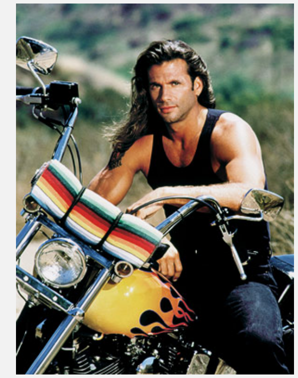
COS226: Quicksort vs. Mergesort

Mergesort	Quicksort
	

Beauty → Motivation → Effort → Comprehension → Beauty

COS226: Quicksort vs. Mergesort

> *"Mathematics, rightly viewed, possesses not only truth, but supreme beauty — a beauty cold and austere, like that of sculpture, without appeal to any part of our weaker nature, without the gorgeous trappings of painting or music, yet sublimely pure, and capable of a stern perfection such as only the greatest art can show. The true spirit of delight, the exaltation, the sense of being more than Man, which is the touchstone of the highest excellence, is to be found in mathematics as surely as poetry"* — *Bertrand Russell (The Study of Mathematics. 1919)*

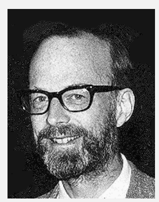


Beauty → Motivation → Effort → Comprehension → Beauty

Quicksort Inventor.

Tony Hoare.

- QuickSort invented in 1960 at age 26
  - Used to help with machine translation project
- Also invented the **null-pointer**
- 4 honorary doctorates
- 1 real doctorate
- Knight



Sir Charles Antony Richard Hoare
1980 Turing Award

> *" I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. 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. "* — *Tony Hoare (2009)*

http://algs4.cs.princeton.edu

2.3 Quicksort

- quicksort
- *practical improvements*
- defeating quicksort (optional)

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);
}
```

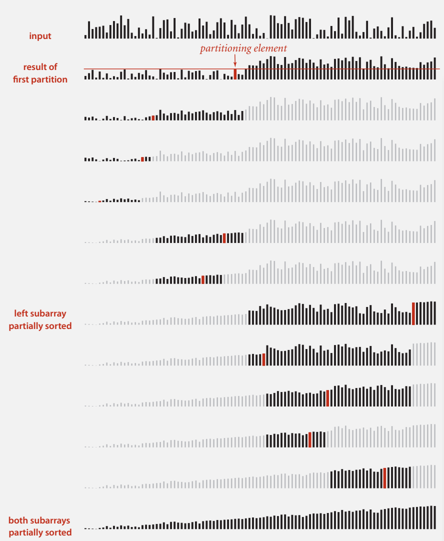

Quicksort: practical improvements

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);
}
```


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



pollEv.com/jhug text to **37607**

Q: Assume an array of length N and that we abort sorting for arrays less than length 10.

How many inversions remain at the last step shown (to left) in the worst case?

A. Θ(N^2) [104533]
B. Θ(cN^2) [104539]
c. Θ(N) [104545]
d. Θ(cN) [104554]

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.
- Place children in magical residential colleges.

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

Duplicate keys

Mergesort with duplicate keys. Between $\frac{1}{2} N \lg N$ and $N \lg N$ compares.

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

which is why ours does!
(but many textbook implementations do not)

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

Duplicate keys: the problem

Mistake. Put all items equal to the partitioning item on one side.
Consequence. $\sim \frac{1}{2} N^2$ compares when all keys equal.

B A A B A B B B C C D   A A A A A A A A A A A

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 D B C B   A A A A A A A A A A A

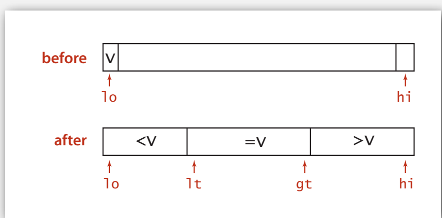
Desirable. Put all items equal to the partitioning item in place.

A A A B B B B B C D C   A A A A A A A A A A A

3-way partitioning

Goal. Partition array into 3 parts so that:

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



Dutch national flag problem. [Edsger Dijkstra]

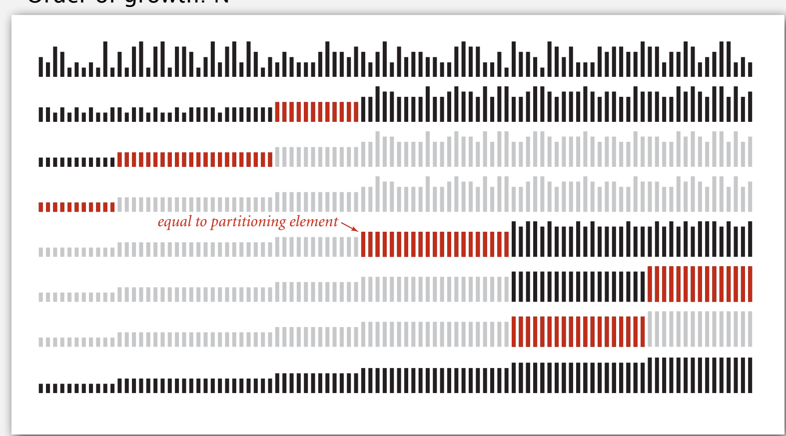
- Conventional wisdom until mid 1990s: not worth doing.
- New approach discovered when fixing mistake in C library `qsort()`.
- Now incorporated into `qsort()` and Java system sort.

3-way partitioning

Key Insight

Assumed constant

- Given an array of length N and k distinct items.
- How many times do you have to partition?
  - k, each taking Θ(N) time.
- Order of growth: N



Real world considerations

Introsort

- Detect when sort goes quadratic (recursion depth exceeds some level).
  - Switch to heapsort (or mergesort).
- Detect when subproblem is less than size 15 or so.
  - Insertion sort is faster for small arrays.

Handling almost sorted arrays

- Can optimize Quicksort to handle this. Or...
  - Timsort (fancy mergesort)
  - Smoothsort
  - Insertion Sort (if you're feeling lucky!)

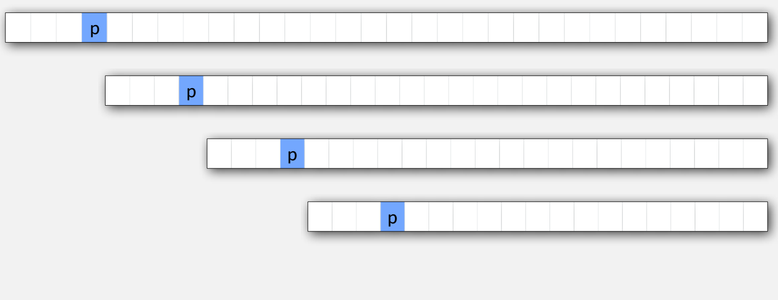
http://algs4.cs.princeton.edu

2.3 Quicksort

Defeating deterministic Quicksort

Goal

- Find a sequence of integers such that pivot(s) always ends up within a constant distance of the left edge.
  - Results in $O(N^2)$ runtime.



Defeating deterministic Quicksort

Evil Integer

- True form set at time of creation.
- Chosen form is chosen later.
  - Permanent choice.
  - Becomes **solid** after choice.

```
class EvilInteger {
   int trueForm;
   int chosenForm;
   boolean solid;

   EvilInteger(int value) {
      trueForm = value;
   }
}
```

True	0	1	2	3	4
Chosen			10	7	
Solid	False	False	True	True	False

Comparing Evil Integers (three cases)

I. Both are solid.

- Compare with chosen value.

True	0	1	2	3	4
Chosen			10	7	
Solid	False	False	True	True	False

Comparing Evil Integers (three cases)

I. Both are solid.

- Compare with chosen value.

A[2] < A[3]?

10 < 7?

False

True	0	1	2	3	4
Chosen			10	7	
Solid	False	False	True	True	False

Comparing Evil Integers (three cases)

I. Both are solid.

- Compare with chosen value.

II. One is solid, one is not.

- Solid one is considered less.
- The gooey one gains ‘**the mark**’.

A[3] < A[1]?

7 <  ?

True

True	0	1	2	3	4
Chosen			10	7	
Solid	False	False	True	True	False

Comparing Evil Integers (three cases)

I. Both are solid.

- Compare with chosen value

II. One is solid, one is not.

- Solid one is considered less.
- The gooey one gains ‘**the mark**’.

A[3] < A[1]?

7 <  ?

True

True	0	1	2	3	4
Chosen			10	7	
Solid	False	False	True	True	False

Comparing Evil Integers (three cases)

I. Both are solid.

- Compare with chosen value

II. One is solid, one is not.

- Solid one is considered less.
- The gooey one gains ‘**the mark**’.

A[3] < A[0]?

7 <  ?

True

True	0	1	2	3	4
Chosen			10	7	
Solid	False	False	True	True	False

Comparing Evil Integers (three cases)

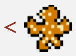
I. Both are solid.

- Compare with chosen value

II. One is solid, one is not.

- Solid one is considered less.
- The gooey one gains ‘**the mark**’.

A[0] < A[4]?

 <  ?

Must be bigger than 7 and 10

III. Neither is solid.

- One chooses a value and becomes solid.
  - Newly chosen value must be larger than all others.
  - If either Evil Integer is marked, it preferentially solidifies.
- Go to case II.

True	0	1	2	3	4
Chosen			10	7	
Solid	False	False	True	True	False

Comparing Evil Integers (three cases)

I. Both are solid.

- Compare with chosen value

II. One is solid, one is not.

- Solid one is considered less.
- The gooey one gains ‘**the mark**’.

A[0] < A[4]?

42 <  ?

True

III. Neither is solid.

- One chooses a value and becomes solid.
  - Newly chosen value must be larger than all others.
  - If either Evil Integer is marked, it preferentially solidifies.
- Go to case II.

True	0	1	2	3	4
Chosen	42		10	7	
Solid	True	False	True	True	False

Comparing Evil Integers (three cases)

I. Both are solid.

- Compare with chosen value

II. One is solid, one is not.

- Solid one is considered less.
- The gooey one gains ‘**the mark**’.

III. Neither is solid.

- One chooses a value and becomes solid.
  - Newly chosen value must be larger than all others.
  - If either Evil Integer is marked, it preferentially solidifies.
- Go to case II.

True	0	1	2	3	4
Chosen	42		10	7	
Solid	True	False	True	True	False

Comparing Evil Integers

Observations

- Evil integers that solidify first have smallest values.
- Evil integers tell a consistent story (not obvious!).
  - Obey all the normal properties of an inequality.
  - Example: If E0 < E1 at some point, E0 will always be less than E1.

Reminder

- Goal: Find a sequence of integers that causes Quicksort to go quadratic.

True	0	1	2	3	4
Chosen	42		10	7	
Solid	True	False	True	True	False

Quicksorting Evil Integers

Using first element as pivot.

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C																			

L G

Quicksorting Evil Integers

Using first element as pivot.

- A[L] < p?

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C																			

L G

Quicksorting Evil Integers

Using first element as pivot.

- A[L] < p?
  - 23 <  ?
  - Yes.

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C		23																	

L G

Quicksorting Evil Integers

Using first element as pivot.

- A[L] < p?
  - 23 <  ?
  - Yes.

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C		23																	

L G

Quicksorting Evil Integers

Using first element as pivot.

- A[L] < p?
  - 23 <  ?
  - Yes.
- A[L] < p?

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C		23																	

L G

Quicksorting Evil Integers

Using first element as pivot.

- A[L] < p?
  - 23 <  ?
  - Yes.
- A[L] < p?
  -  < 24?
  - No.

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	23																	

L G

Quicksorting Evil Integers

Using first element as pivot.

- A[G] > p?

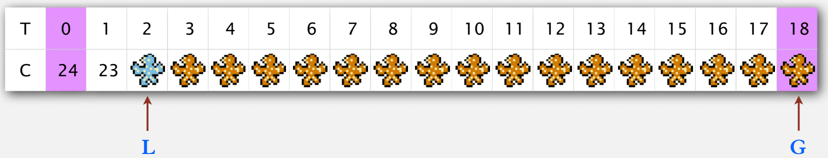
T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	23																	

L G

Quicksorting Evil Integers

Using first element as pivot.

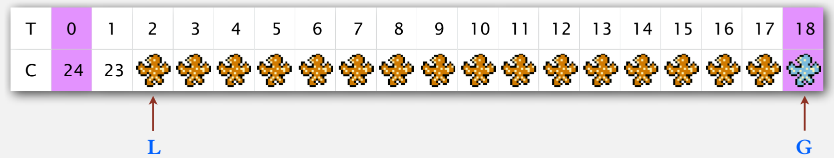
- A[G] > p?
  -  > 24?
  - Yes!



Quicksorting Evil Integers

Using first element as pivot.

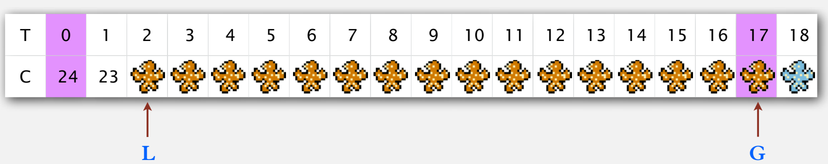
- A[G] > p?
  -  > 24?
  - Yes!



Quicksorting Evil Integers

Using first element as pivot.

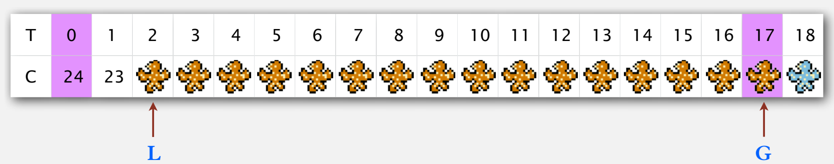
- A[G] > p?
  -  > 24?
  - Yes!
- A[G] > p?



Quicksorting Evil Integers

Using first element as pivot.

- A[G] > p?
  -  > 24?
  - Yes!
- A[G] > p?
  -  > 24?
  - Yes!



Quicksorting Evil Integers

Using first element as pivot.

- A[G] > p?
  -  > 24?
  - Yes!
- A[G] > p?
  -  > 24?
  - Yes!

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	23																	

L (index 2) G (index 17)

Quicksorting Evil Integers

Using first element as pivot.

- A[G] > p?
  -  > 24?
  - Yes!
- A[G] > p?
  -  > 24?
  - Yes!
- A[G] > p?

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	23																	

G L

Quicksorting Evil Integers

Using first element as pivot.

- A[G] > p?
  -  > 24?
  - Yes!
- A[G] > p?
  -  > 24?
  - Yes!
- A[G] > p?
  - 23 > 24?
  - No!

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	23																	

G L

Quicksorting Evil Integers

Using first element as pivot.

- L and G have both stopped.
  - L > G, so don't swap A[L] and A[G].

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	23																	

G L

Quicksorting Evil Integers

Using first element as pivot.

- L and G have both stopped.
  - L > G, so don't swap A[L] and A[G].

- Swap pivot (24) and A[G]

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	23																	

G L

Quicksorting Evil Integers

Using first element as pivot.

- L and G have both stopped.
  - L > G, so don't swap A[L] and A[G].

- Swap pivot (24) and A[G]

T	1	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	23	24																	

Subproblem of size N-2

Quicksorting Evil Integers

Using median of 3 as pivot.

- Median identification.
  - Involves compares.

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C																			

Quicksorting Evil Integers

Using median of 3 as pivot.

- Median identification.
  - Involves compares.
  - 23 < 24 <

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	23																		24

Quicksorting Evil Integers

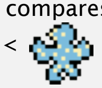
Using median of 3 as pivot.

- Median identification.
  - Involves compares.
  - 23 < 24 < 
- Swap median (24) into pivot position.

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	23																		24

Quicksorting Evil Integers

Using median of 3 as pivot.

- Median identification.
  - Involves compares.
  - 23 < 24 < 
- Swap median (24) into pivot position.

T	18	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	0
C	24																		23

Quicksorting Evil Integers

Using median of 3 as pivot.

- L will immediately stop ( is not less than 24)

T	18	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	0
C	24																		23

L G

Quicksorting Evil Integers

Using median of 3 as pivot.

- L will immediately stop ( is not less than 24)
- G will immediately stop (23 is not greater than 24)

T	18	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	0
C	24																		23

L G

Quicksorting Evil Integers

Using median of 3 as pivot.

- L will immediately stop (is not less than 24)
- G will immediately stop (23 is not greater than 24)
- Swap and 23

T	18	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	0
C	24																		23

L G

Quicksorting Evil Integers

Using median of 3 as pivot.

- L will immediately stop (is not less than 24)

T	18	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1
C	24	23																	

L G

Quicksorting Evil Integers

Using median of 3 as pivot.

- L will immediately stop (is not less than 24)
- G will go all the way to the beginning (are all bigger than 24)

T	18	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1
C	24	23																	

L G

Quicksorting Evil Integers

Using median of 3 as pivot.

- L and G have both stopped.
- L > G.
  - So don’t swap A[L] and A[G].

T	18	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1
C	24	23																	

G L

Quicksorting Evil Integers

Using median of 3 as pivot.

- L and G have both stopped.
- L > G.
  - So don't swap A[L] and A[G].

- Swap pivot (24) and A[G].

T	18	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1
C	24	23																	

G L

Quicksorting Evil Integers

Using median of 3 as pivot.

- L and G have both stopped.
- L > G.
  - So don't swap A[L] and A[G].

- Swap pivot (24) and A[G].

T	18	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1
C	23	24																	

Subproblem of size N-2

Quicksorting Evil Integers

What's going on here?

- Pivot is reused by quicksort during every compare.
  - Pivot solidifies earlier than almost all other elements.
- Let k be number of items used to decide pivot.
  - Evil Integer pivot is always within first k positions.

T	18	0	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	1
C	23	24																	

Subproblem of size N-2

Comparing Evil Integers

Observations

- Evil Integers cause Quicksort to go quadratic.

Reminder

- Goal: Find a sequence of integers that causes Quicksort to go quadratic.

Upon completion

T	9	0	2	10	4	11	6	12	8	13	3	14	7	15	5	16	1	17	18
C	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	

Extracting information from Evil Integers

Starting Evil Integer array

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C																			

After Quicksorting

T	9	0	2	10	4	11	6	12	8	13	3	14	7	15	5	16	1	17	18
C	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	

Just put them back where they started

- Sort Evil Integers by their true value

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	39	25	33	27	37	29	35	31	23	26	28	30	32	34	36	38	40	

Comparing Evil Integers

Starting Evil Integer array

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C																			

Upon completion

T	9	0	2	10	4	11	6	12	8	13	3	14	7	15	5	16	1	17	18
C	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	

Sneaky trick

- Sort EvilIntegers by true value:

T	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	24	39	25	33	27	37	29	35	31	23	26	28	30	32	34	36	38	40	

This sequence of non-evil integers causes median-of-3 Quicksort to go quadratic!

Non-deterministic Quicksort

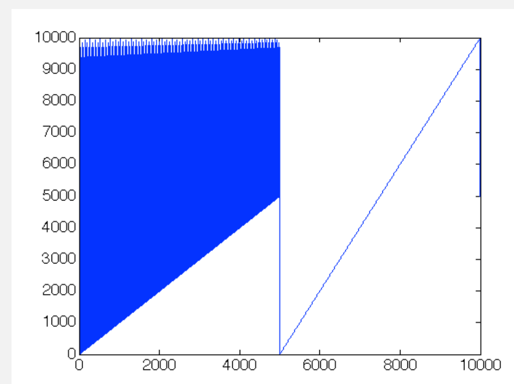
Every non-deterministic Quicksort is vulnerable

- Pivot is reused early and often by all algorithms one might call 'Quicksort'.
  - Pivot is guaranteed to be within k positions of the front.
  - Arithmetic decrease in problem size means $\Theta(N^2)$ performance.

Neat fact

Each deterministic Quicksort has its own Achilles shape

- Median-of-3s with 2-way partitioning:

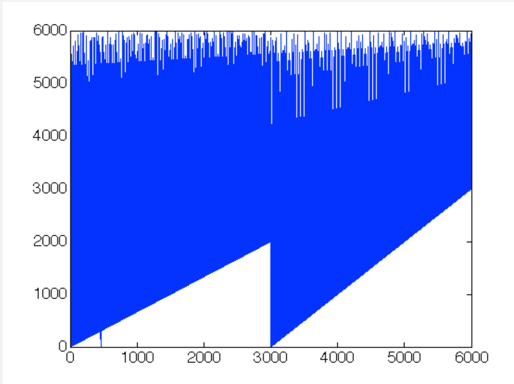


[9800, 9850, 9200, 9801, 9851, 9201...]

Neat fact

Each deterministic Quicksort has its own Achilles shape

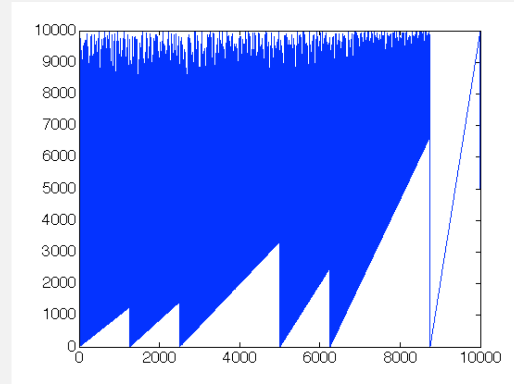
- Median-of-3s with 3-way partitioning:



Neat fact

Each deterministic Quicksort has its own Achilles shape

- Arrays.sort() in Java



Arrays.sort

troublesomeIntegers.txt

```
0
87499
67183
11
66672
75394
33339
90626
67185
17
91800
62508
75002
71877
34380
22 ...
```

```
$ java-226 SortIntegers 100000 < troublesomeIntegers.txt
Sorting 100000 integers
Exception in thread "main" java.lang.StackOverflowError
        at java.util.Arrays.sort1(Arrays.java:561)
        at java.util.Arrays.sort1(Arrays.java:605)
        at java.util.Arrays.sort1(Arrays.java:607)
        at java.util.Arrays.sort1(Arrays.java:607)
```