Announcements

Project 3 autograder has been updated to do randomized testing.

- In general, it is not reasonable to hardcode answers to projects in any class, ever (e.g. a program which consists of if statements).

More people are failing the 3x test than we expected, will investigate.

- Possible culprit: You might be tested against a solution that uses Euclidean rather than Great Circle distance, which is slower.

Extra deadline for part 1 is today.

CS61B

Lecture 34: Sorting III

- Quicksort Flavors vs. Mergesort
- Selection (Quick Select)
- Stability, Adaptiveness, and Optimization
- Shuffling

Sorting Summary (so far)

Listed by mechanism:

- Selection sort: Find the smallest item and put it at the front.
- Insertion sort: Figure out where to insert the current item.
- Merge sort: Merge two sorted halves into one sorted whole.
- Partition (quick) sort: Partition items around a pivot.

Listed by memory and runtime:

	Memory	Time	Notes
Heapsort	$\Theta(1)$	$\Theta(N \log N)$	Bad caching (61C)
Insertion	$\Theta(1)$	$\Theta(N^2)$	$\Theta(N)$ if almost sorted
Mergesort	$\Theta(N)$	$\Theta(N \log N)$	
Random Quicksort	$\Theta(\log N)$ expected	$\Theta(N \log N)$ expected	Fastest sort

Quicksort Flavors

We said Quicksort is the fastest, but this is only true if we make the right decisions about:

- Pivot selection.
- Partition algorithm.
- How we deal with avoiding the worst case (can be covered by the above choices).

We'll call this QuicksortL3S

Let's speed test Mergesort vs. Quicksort from last time, which had:

- Pivot selection: Always use **leftmost**.
- Partition algorithm: Make an array copy then do **three** scans for red, white, and blue items (white scan trivially finishes in one compare).
- **Shuffle** before starting (to avoid worst case).

Quicksort vs. Mergesort

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints
Mergesort	N/A	N/A	N/A	2.1 seconds
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds

Quicksort didn’t do so well!

Note: These are unoptimized versions of mergesort and quicksort, i.e. no switching to insertion sort for small arrays.

Tony Hoare’s In-place Partitioning Scheme

Tony originally proposed a scheme where two pointers walk towards each other.

- Left pointer loves small items.
- Right pointer loves large items.
- Big idea: Walk towards each other, swapping anything they don’t like.
  - End result is that things on left are “small” and things on the right are “large”.

Full details here: Demo

Using this partitioning scheme yields a very fast Quicksort.

- Though faster schemes have been found since.
- Overall runtime still depends crucially on pivot selection strategy!

Quicksort vs. Mergesort

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints
Mergesort	N/A	N/A	N/A	2.1 seconds
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds
Quicksort LTHS	Leftmost	Tony Hoare	Shuffle	1.7 seconds

Using Tony Hoare's two pointer scheme, Quicksort is better than mergesort!

- More recent pivot/partitioning schemes do somewhat better.
  - Best known Quicksort uses a two-pivot scheme.
  - Interesting note, this version of Quicksort was introduced to the world by a previously unknown guy, in a Java developers forum (Link).

Note: These are unoptimized versions of mergesort and quicksort, i.e. no switching to insertion sort for small arrays.

What If We Don’t Want Randomness?

Our approach so far: Use randomness to avoid worst case behavior, but some people don’t like having randomness in their sorting routine.

Another approach: Use the median (or an approximation) as our pivot.

Four philosophies:

This is what we’ve been using.

1. **Randomness**: Pick a random pivot or shuffle before sorting.
2. **Smarter pivot selection**: Calculate or approximate the median.
3. **Introspection**: Switch to a safer sort if recursion goes to deep.
4. **Try to cheat**: If the array is already sorted, don’t sort (this doesn’t work).

Philosophy 2a: Smarter Pivot Selection (linear time pivot pick)

The best possible pivot is the median.

- Splits problem into two problems of size N/2.

Obvious approach: Just calculate the actual median and use that as pivot.

- But how?

Goal: Come up with an algorithm for finding the median of an array. Bonus points if your algorithm takes linear time.

Philosophy 2a: Smarter Pivot Selection (linear time pivot pick)

The best possible pivot is the median.

- Splits problem into two problems of size N/2.

Obvious approach: Just calculate the actual median and use that as pivot.

- But how?

Goal: Come up with an algorithm for finding the median of an array. Bonus points if your algorithm takes linear time.

Your answer:

Median Identification

Is it possible to find the median in Θ(N) time?

- Yes! Use ‘BFPRT’ (called PICK in original paper).
- Algorithm developed in 1972 by a team including my former TA, Bob Tarjan (well before I was born).
- In practice, rarely used.

Historical note: The authors of this paper include FOUR Turing Award winners (and Pratt is no slouch!)

Time Bounds for Selection*

MANUEL BLUM, ROBERT W. FLOYD, VAUGHAN PRATT, RONALD L. RIVEST, AND ROBERT E. TARJAN

Department of Computer Science, Stanford University, Stanford, California 94305

Received November 14, 1972

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm—PICK. Specifically, no more than $5.430\dot{5}\,n$ comparisons are ever required. This bound is improved for extreme values of i, and a new lower bound on the requisite number of comparisons is also proved.

Let’s see how Exact Median Quicksort performs.

Quicksort vs. Mergesort

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints	Worst Case
Mergesort	N/A	N/A	N/A	2.1 seconds	$\Theta(N \log N)$
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds	$\Theta(N^2)$
Quicksort LTHS	Leftmost	Tony Hoare	Shuffle	1.7 seconds	$\Theta(N^2)$
Quicksort PickTH	Exact Median	Tony Hoare	Exact Median	10.0 seconds	$\Theta(N \log N)$

Quicksort using PICK to find the exact median (Quicksort PickTH) is terrible!

- Cost to compute medians is too high.
- Have to live with worst case $\Theta(N^2)$ if we want good practical performance.

Note: These are unoptimized versions of mergesort and quicksort, i.e. no switching to insertion sort for small arrays.

Quick Select

The Selection Problem

Computing the exact median would be great for picking an item to partition around. Gives us a “safe quick sort”.

- Unfortunately, it turns out that exact median computation is too slow.

However, it turns out that partitioning can be used to find the exact median.

- The resulting algorithm is the best known median identification algorithm.

Quick Select

Goal, find the median:

9	550	14	6	10	5	330	817	913

Partition, pivot lands at 2.

- Not the median. Why?
- So what next? Partition right subproblem, median can't be to the left!

6	5	9	550	14	10	330	817	913

			550	14	10	330	817	913

Now pivot lands at 6.

- Not the median.

			14	10	330	550	817	913

Pivot lands at 4. Are we done?

- Yep, 9/2 = 4.

			14	10	330			

			10	14	330			

Worst case performance?

What is the worst case performance for Quick Select? Give an array that causes this worst case (assuming we always pick leftmost item as pivot).

Worst case performance?

What is the worst case performance for Quick Select? Give an array that causes this worst case (assuming we always pick leftmost item as pivot).

Worst asymptotic performance $\Theta(N^2)$ occurs if array is in sorted order.

[1 2 3 4 5 6 7 8 9 10 … N]

[1 **2 3 4 5 6 7 8 9 10 … N**]

[1 2 **3 4 5 6 7 8 9 10 … N**]

…

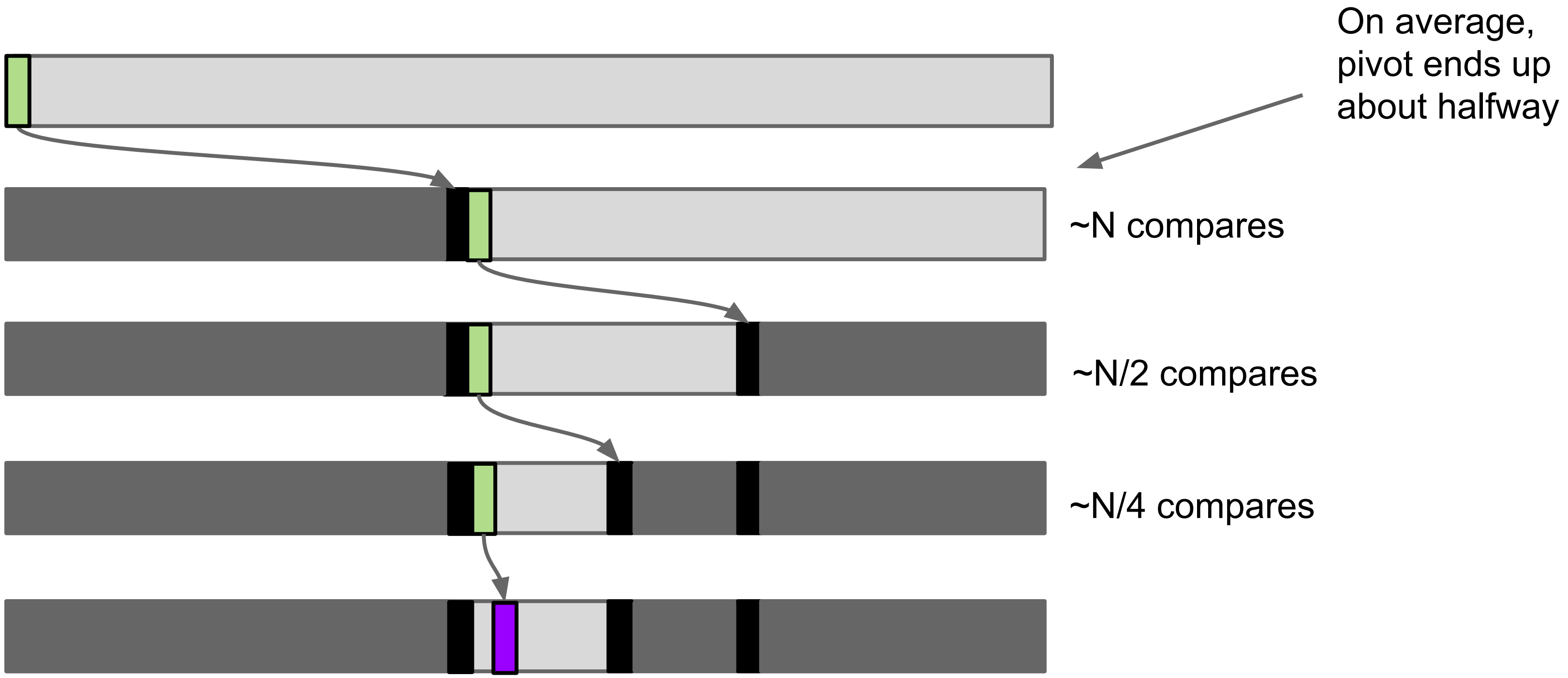
[1 2 3 4 5 … **N/2** … N]

Expected Performance

On average, Quick Select will take Θ(N) time.

- Intuitive picture (not a proof!):

$$N + N/2 + N/4 + \ldots + 1 = \Theta(N)$$



Quicksort With Quickselect?

	Pivot Selection Strategy	Partition Algorithm	Worst Case Avoidance Strategy	Time to sort 1000 arrays of 10000 ints	Worst Case
Mergesort	N/A	N/A	N/A	2.1 seconds	$\Theta(N \log N)$
Quicksort L3S	Leftmost	3-scan	Shuffle	4.4 seconds	$\Theta(N^2)$
Quicksort LTHS	Leftmost	Tony Hoare	Shuffle	1.7 seconds	$\Theta(N^2)$
Quicksort PickTH	Exact Median	Tony Hoare	Exact Median	10.0 seconds	$\Theta(N \log N)$

Quicksort with PICK to find exact median was terrible.

What if we used Quickselect to find the exact median?

- Resulting algorithm is still quite slow. Also: a little strange to do a bunch of partitions to identify the optimal item to partition around.

Stability, Adaptiveness, Optimization

Sorting Summary (so far)

Listed by mechanism:

- Selection sort: Find the smallest item and put it at the front.
- Insertion sort: Figure out where to insert the current item.
- Merge sort: Merge two sorted halves into one sorted whole.
- Partition (quick) sort: Partition items around a pivot.

Listed by memory and runtime:

	Memory	# Compares	Notes
Heapsort	$\Theta(1)$	$\Theta(N \log N)$ worst	Bad caching (61C)
Insertion	$\Theta(1)$	$\Theta(N^2)$ worst	$\Theta(N)$ if almost sorted
Mergesort	$\Theta(N)$	$\Theta(N \log N)$ worst	
Random Quicksort	$\Theta(\log N)$ (call stack)	$\Theta(N \log N)$ expected	Fastest sort

Other Desirable Sorting Properties: Stability

A sort is said to be stable if order of equivalent items is preserved.

sort(studentRecords, BY_NAME);

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

sort(studentRecords, BY_SECTION);

Lara	1
Sigurd	2
Bas	3
Jana	3
Jouni	3
Rosella	3
Fikriyya	4
Nikolaj	4

Equivalent items don't 'cross over' when being stably sorted.

Other Desirable Sorting Properties: Stability

A sort is said to be stable if order of equivalent items is preserved.

sort(studentRecords, BY_NAME);

Bas	3
Fikriyya	4
Jana	3
Jouni	3
Lara	1
Nikolaj	4
Rosella	3
Sigurd	2

sort(studentRecords, BY_SECTION);

Lara	1
Sigurd	2
Jouni	3
Rosella	3
Bas	3
Jana	3
Fikriyya	4
Nikolaj	4

Sorting instability can be really annoying! Wanted students listed alphabetically by section.

Sorting Stability

Is insertion sort stable?

```
S O R T E X A M P L E
S O R T E X A M P L E   (0 swaps)
O S R T E X A M P L E   (1 swap )
O R S T E X A M P L E   (1 swap )
O R S T E X A M P L E   (0 swaps)
E O R S T X A M P L E   (4 swaps)
E O R S T X A M P L E   (0 swaps)
A E O R S T X M P L E   (6 swaps)
A E M O R S T X P L E   (5 swaps)
A E M O P R S T X L E   (4 swaps)
A E L M O P R S T X E   (7 swaps)
A E E L M O P R S T X   (8 swaps)
```

Is Quicksort stable?

- Consider -------->

6	8	3	1	2	7	4

Sorting Stability

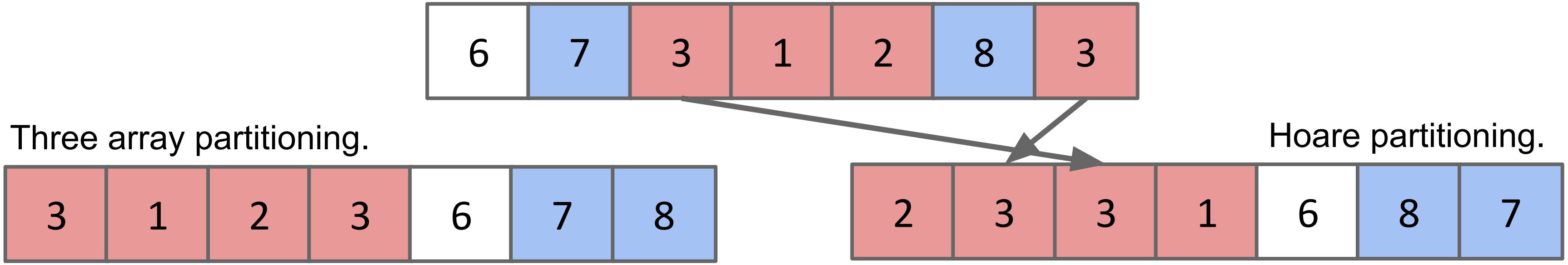
Is insertion sort stable?

- Yes.
- Equivalent items never move past their equivalent brethren.

```
S O R T E X A M P L E
S O R T E X A M P L E   (0 swaps)
O S R T E X A M P L E   (1 swap )
O R S T E X A M P L E   (1 swap )
O R S T E X A M P L E   (0 swaps)
E O R S T X A M P L E   (4 swaps)
E O R S T X A M P L E   (0 swaps)
A E O R S T X M P L E   (6 swaps)
A E M O R S T X P L E   (5 swaps)
A E M O P R S T X L E   (4 swaps)
A E L M O P R S T X E   (7 swaps)
A E E L M O P R S T X   (8 swaps)
```

Is Quicksort stable?

- Depends on your partitioning strategy.



Stability

	Memory	# Compares	Notes	Stable?
Heapsort	Θ(1)	Θ(N log N)	Bad caching (61C)	No
Insertion	Θ(1)	$\Theta(N^2)$	Θ(N) if almost sorted	Yes
Mergesort	Θ(N)	Θ(N log N)		Yes
Quicksort LTHS	Θ(log N)	Θ(N log N) expected	Fastest sort	No

This is due to the cost of tracking recursive calls by the computer, and is also an "expected" amount. The difference between log N and constant memory is trivial.

You can create a stable Quicksort. However, unstable partitioning schemes (like Hoare partitioning) tend to be faster. All reasonable partitioning schemes yield Θ(N log N) expected runtime, but with different constants.

Optimizing Sorts

Additional tricks we can play:

- Switch to insertion sort:
  - When a subproblem reaches size 15 or lower, use insertion sort.
- Make sort ***adaptive***: Exploit existing order in array (Insertion Sort, SmoothSort, TimSort (*the* sort in Python and Java)).
- Exploit restrictions on set of keys. If number of keys is some constant, e.g. [3, 4, 1, 2, 4, 3, …, 2, 2, 2, 1, 4, 3, 2, 3], can sort faster (see 3-way quicksort -- if you're curious, see: http://goo.gl/3sYnv3).
- For Quicksort: Make the algorithm introspective, switching to a different sorting method if recursion goes too deep. Only a problem for deterministic flavors of Quicksort.

Arrays.sort

In Java, Arrays.sort(someArray) uses:

- Mergesort (specifically the TimSort variant) if someArray consists of Objects.
- Quicksort if someArray consists of primitives.

Why? See A level problems.

`static void`	`sort(Object[] a)` Sorts the specified array of objects into ascending order, according to the **natural ordering** of its elements.

`static void`	`sort(int[] a)` Sorts the specified array into ascending numerical order.

Shuffling

Shuffling

How do we shuffle an array?




Shuffling

How do we shuffle an array?

- Easiest way:
  - Generate N random numbers, and attach one to each array item.
  - Sort the items by the attached random number.

0.8741

0.5512

0.211

0.366

Shuffling

How do we shuffle an array?

- Easiest way:
  - Generate N random numbers, and attach one to each array item.
  - Sort the items by the attached random number.

0.211

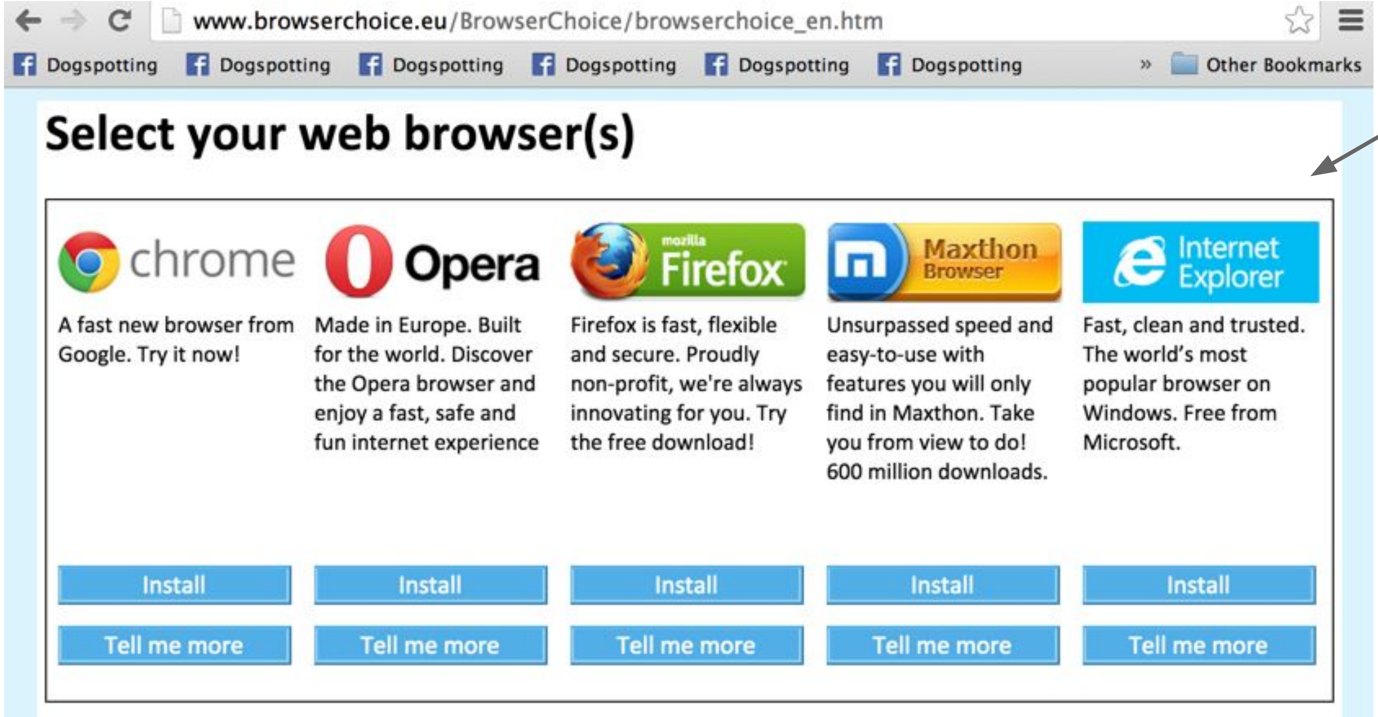
0.366

0.5512

0.8741

How do we get random numbers? See Fa14 lecture 33.

How do we NOT shuffle?



Appeared last 50% of the time.

In response to anti-trust investigation, Microsoft agreed to provide a random browser selection website.

The (Bad) Randomization Approach

Implementation: Make compareTo() return a random answer.

```
public int compareTo(Browser that) {
    double r = Math.random();
    if (r < 0.5) return -1;
    if (r > 0.5) return +1;
    return 0;
}
```

For more, see: http://www.robweir.com/blog/2010/03/new-microsoft-shuffle.html

Appeared last 50% of the time.

The (Bad) Randomization Approach

Implementation: Make compareTo() return a random answer.

- Doesn't play well with insertion sort.
- We use insertion sort for N < ~15.

Simple programming error… or was it?

- http://psych.fullerton.edu/mbirnbaum/papers/Nihm_BPS_1984.pdf

Appeared last 50% of the time.

Sounds of Sorting (Fun)

Sounds of Sorting Algorithms (of 125 items)

Starts with selection sort: https://www.youtube.com/watch?v=kPRA0W1kECg

Insertion sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m9s

Quicksort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=0m38s

Mergesort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m05s

Heapsort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m28s

LSD sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=1m54s [coming next Wednesday]

MSD sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=2m10s [coming next Wednesday]

Shell’s sort: https://www.youtube.com/watch?v=kPRA0W1kECg&t=3m37s [bonus from last time]

Questions to ponder (later… after class):

- How many items for selection sort?
- Why does insertion sort take longer / more compares than selection sort?
- At what time stamp does the first partition complete for Quicksort?
- Could the size of the input to mergesort be a power of 2?
- What do the colors mean for heapsort?
- How many characters are in the alphabet used for the LSD sort problem?
- How many digits are in the keys used for the LSD sort problem?

Citations

Title image:
http://www.constructionphotography.com/ImageThumbs/A168-02831/3/A168-02831_plastic_bottles_sorted_by_colour_compressed_into_bales_and_ready_for_recycling.jpg

Sorting, Puppies, Cats, and Dogs

A solution to the sorting problem also provides a solution to puppy, cat, dog.

- Thus: Sorting must be at least as hard as puppy, cat, dog.
- Because [difficulty of sorting] ≥ [difficulty of puppy, cat, dog], any lower bound on difficulty of puppy, cat, dog must ALSO apply to sorting.

Physics analogy: Climbing a hill with your legs is one way to solve the problem of getting up a hill.

- Thus: Using “climbing a hill with your legs” must be at least as hard as “getting up a hill”.
- Because CAHWYL ≥ GUAH, any lower bound on energy to GUAH must also apply to CAHWYL.
- Example bound: Takes m*g*h energy to climb hill, so using legs to climb the hill takes at least m*g*h energy.