Why do we study sorting?

Because we have to

Because sorting is beautiful

Example of algorithm analysis in a simple, useful setting

There are n sorting algorithms, how many should we study?

Why do we study more than one algorithm?

• Some are good, some are bad, some are very, very sad
• Paradigms of trade-offs and algorithmic design

Which sorting algorithm is best?

Which sort should you call from code you write?

Simple, O(n^2) sorts --- for sorting n elements

Selection sort --- n^2 comparisons, n swaps, easy to code

Insertion sort --- n^2 comparisons, n^2 moves, stable, fast

Bubble sort --- everything, slow, slower, and ugly

Divide and conquer faster sorts: O(n log n) for n elements

Quick sort: fast in practice, O(n^2) worst case

Merge sort: good worst case, great for linked lists, uses extra storage for vectors/arrays

Other sorts:

Heap sort, basically priority queue sorting

Radix sort: doesn't compare keys, uses digits/characters

Shell sort: quasi-insertion, fast in practice, non-recursive

Selection sort: summary

Simple to code n^2 sort: n^2 comparisons, n swaps

```java
void selectSort(String[] a) {
    int len = a.length;
    for (int k = 0; k < len; k++) {
        int mindex = getMinIndex(a, k, len);
        swap(a, k, mindex);
    }
}
```

# comparisons:

Swaps?

Invariant:
k=1

n_k = 1 + 2 + … + n = n(n + 1)/2 = O(n^2)

Sorted, won't move final position

Insertion sort: summary

Stable sort, O(n^2), good on nearly sorted vectors

Stable sorts maintain order of equal keys

Good for sorting on two criteria: name, then age

```java
void insertSort(String[] a) {
    int k, loc; String elt;
    for (k = 1; k < a.length; ++k) {
        elt = a[k];
        loc = k;
        // shift until spot for elt is found
        while (0 < loc && elt.compareTo(a[loc - 1]) < 0) {
            a[loc] = a[loc-1];   // shift right
            loc = loc-1;
        }
        a[loc] = elt;
    }
}
```

Sorted relative to each other

Root
Bubble sort: summary of a dog

For completeness you should know about this sort

Really, really slow (to run), really really fast (to code)

Can code to recognize already sorted vector (see insertion)

- Not worth it for bubble sort, much slower than insertion

```java
void bubbleSort(String[] a) {
    for (int j = a.length - 1; j >= 0; j--) {
        for (int k = 0; k < j; k++) {
            if (a[k].compareTo(a[k + 1]) > 0) { // bbble elements down the vector/array
                swap(a, k, k + 1);
            }
        }
    }
}
```

"buble" elements down the vector/array

Sorted, in final position

Summary of simple sorts

Selection sort has n swaps, good for "heavy" data

- Moving objects with lots of state, e.g., … • In C or C++ this is an issue • In Java everything is a pointer/reference, so swapping is fast since it's pointer assignment

Insertion sort is good on nearly sorted data, it's stable, it's fast

Also found as foundation for Shell sort, very fast non-recursively • More complicated to code, but relatively simple, and fast

Bubble sort is a travesty? But it's fast to code if you know it!

Can be parallelized, but on one machine doesn't go near it (see quotes at end of slides)

Quick sort: fast in practice

Invented in 1962 by C. A. R. Hoare, didn't understand recursion

Worst case is O(n^2), but avoidable in nearly all cases

In 1997 Introsort published (Mussner, introspective sort)

- Like quick sort in practice, but recognizes when it will be bad and changes to heap sort

```java
void quick(String[], int left, int right) {
    if (left < right) {
        int pivot = partition(a, left, right);
        quick(a, left, pivot - 1);
        quick(a, pivot + 1, right);
    }
}
```

Recurrence?

<= X > X X

pivot index

Partition code for quick sort

Easy to develop partition

```java
int partition(String[] a, int left, int right) {
    String pivot = a[left];
    int k, pIndex = left;
    for (k = left + 1; k <= right; k++) {
        if (a[k].compareTo(pivot) <= 0) {
            pIndex++;
            swap(a, k, pIndex);
        }
    }
    swap(a, left, pIndex);
}
```

Loop invariant:

- Statement true each time test is evaluated, used to verify correctness of loop

- Can swap into a[left] before loop

- Nearly sorted data still ok

```java
<= pivot > pivot

pIndex

k

left

right

what we want

what we have

invvariant

pIndex
```
Analysis of QuickSort!

Average case and worst case analysis

Recurrence for worst case: \( T(n) = \)

What about average?

Reason informally:

Two calls vector size \( n/2 \)

Four calls vector size \( n/4 \)

\ldots

How many calls? Work done on each call?

Partition: typically find middle of left, middle, right, swap, go

Avoid bad performance on nearly sorted data

In practice: remove some (all?) recursion, avoid lots of "clones"

\( T(n-1) + T(1) + O(n) \)

\( T(n) = 2T(n/2) + O(n) \)

Tail recursion elimination

If the last statement is a recursive call, recursion can be replaced with iteration

Call cannot be part of an expression

Some compilers do this automatically

\[
\begin{align*}
\text{void foo} & (\text{int} \ n) \{
\text{if (0 < n) \{ System.out.println(n); \}
\text{foo}(n-1); \\
\text{n = n - 1; \}}
\end{align*}
\]

What if print and recursive call switched?

What about recursive factorial?

return \( n \star \text{factorial}(n-1) \);

Merge sort: worst case \( O(n \log n) \)

Divide and conquer - - - sort recursively

Divide list/vector into two halves

\bullet Sort each half

\bullet Merge sorted halves together

What is complexity of merging two sorted lists?

What is recurrence relation for merge sort as described?

\( T(n) = \)

What is advantage of array over linked list for merge sort?

What about merging, advantage of linked list?

Array requires auxiliary storage (or very fancy coding)

\( T(n) = 2T(n/2) + O(n) \)

Merge sort: lists or arrays or …

Merge sort for arrays

\[
\begin{align*}
\text{void merge} & (\text{String \[\] a, \text{int} \ \text{left, int} \ \text{right}) \{
\text{if (left < right) \{ int mid = (right + left) / 2; \}
\text{merge}(a, left, mid); \\
\text{merge}(a, mid + 1, right); \\
\text{merge(a,left,mid,right); \}}
\end{align*}
\]

What's different when linked lists used?

Do differences affect complexity? Why?

How does merge work?
public static LinkedList<String> merge(LinkedList<String> a, LinkedList<String> b) {
    LinkedList<String> result = new LinkedList<String>();
    while (a.size() != 0 && b.size() != 0) {
        String as = a.getFirst();
        String bs = b.getFirst();
        if (as.compareTo(bs) <= 0) {
            result.add(a.remove());
        } else {
            result.add(b.remove());
        }
    }
    // what's missing here??
}

public static Node merge(Node a, Node b) {
    Node result = new Node("dummy");
    Node last = result;
    while (a != null && b != null) {
        String as = a.info;
        String bs = b.info;
        if (as.compareTo(bs) <= 0) {
            last.next = a;
            a = a.next;
            last = last.next; last = null;
        } else {
            // similar code for b
        }
    }
    // what's missing here??
    // what's returned?
}

void merge(String[] a, int left, int middle, int right) {
    // pre:  left <= middle <= right,
    //       a[left] <= …< =  a[middle] ,
    //       a[middle+1] <= …< =  a[right]
    // post: a[left] <= …< =  a[right]
    // need extra storage, can't easily merge in place!
}

Summary of O(n log n) sorts

Quick sort is relatively straightforward to code, very fast!
Worst case is very unlikely, but possible, therefore …
But, if lots of elements are equal, performance will be bad
• One million integers from range 0 to 10,000
• How can we change partition to handle this?

Merge sort is stable, it's fast, good for linked lists, harder to code?
Worst case performance is O(n log n), compare quick sort!
Extra storage for array/vector!

Heapsort, more complex to code, good worst case, not stable!
Basic based priority queue in a vector
11.17

Sorting in practice

Rarely will you need to roll your own sort, but when you do...

What are key issues?

If you use a library sort, you need to understand the interface.

In C++, STL has sort, and stable_sort.

In C generic sort is complex to use because arrays are ugly.

In Java guarantees and worst-case are important.

Why wouldn’t quicksort be used?

Comparators permit sorting criteria to change simply.

Non-comparison-based sorts

Lower bound: \((n \log n)\) for comparison-based sorts (like searching lower bound).

Bucket sort/radix sort are not comparison-based, faster asymptotically and in practice.

Sort a vector of ints, all in the range 1..100, how?

(use extra storage)

Radix: examine each digit of numbers being sorted.

One-pass per digit.

Sort based on digit 0

10 23 34 42 44 73 25 26 16

10 25 26 23 34 42 44 73 16

10 16 23 25 26 34 42 44 56 73

Jimm Gray (Turing 1998)

![Feather. I love bubble sort, and I grow weary of people who have nothing better to do than to preach about it. Universities are good places to keep such people, so that they don’t scare the general public.](http://example.com)

![Brian Reid (Hopper Award 1982)](http://example.com)
Brian Reid (Hopper 1982)

I am quite capable of squaring N with or without a calculator, and I know how long my sorts will bubble. I can type every form of bubble sort into a text editor from memory. If I am writing some quick code and I need a sort quick, as opposed to a quick sort, I just type in the bubble sort as if it were a statement. I'm done with it before I could look up the data type of the third argument to the quick sort library.

I have a dual-processor 1.2 GHz Power Mac and it sneers at your N squared for most interesting values of N. And my source code is smaller than yours.

Brian Reid who keeps all of his bubbles sorted any how.

Niklaus (Turin award 1984)

I have read your article and share your view that Bubble Sort hardly has any merits. I think that it is so often mentioned, because it illustrates quite well the principle of sorting by exchanging.

I think BS is popular, because it fits well into a systematic development of sorting algorithms. But it plays no role in actual applications. Quite in contrast to C, also without merit (and its derivative Java) among programming codes.

Guy L. Steele, Jr. (Hopper '88)