Sorting: From Theory to Practice

- 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?
  - `O(n), O(log n), …`
  - 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?

Sorting out sorts

- 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 --- n^2 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

```
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: $\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = n(n+1)/2 = O(n^2)$
  - Swaps?
  - Invariant:

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

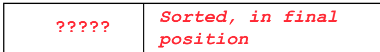
```
void insertSort(String[] a){
    int k, loc; String elt;
    for(k=1; k < a.length; ++k) {
        elt = a[k];
        loc = k;
        // shift until spot for elt is found
        while (0 < loc && elt.compareTo(a[loc-1]) < 0) {
            a[loc] = a[loc-1];   // shift right
            loc=loc-1;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

Sorted relative to each other	?????

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

```
void bubbleSort(String[] a)
{
    for(int j=a.length-1; j >= 0; j--) {
        for(int k=0; k < j; k++) {
            if (a[k] > a[k+1])
                swap(a,k,k+1);
        }
    }
}
```



- "bubble" elements down the vector/array

Summary of simple sorts

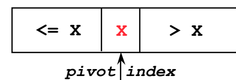
- Selection sort has n swaps, good for "heavy" data
  - moving objects with lots of state, e.g., ...
    - In C or C++ this is an issue
    - In Java everything is a pointer/reference, so swapping is fast since it's pointer assignment
- Insertion sort is good on nearly sorted data, it's stable, it's fast
  - Also foundation for Shell sort, very fast non-recursive
  - 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 don't go near it (see quotes at end of slides)

Quicksort: 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 (Musser, introspective sort)
    - Like quicksort in practice, but recognizes when it will be bad and changes to heapsort

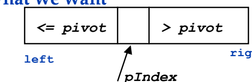
```
void quick(String[], int left, int right)
{
    if (left < right) {
        int pivot = partition(a,left,right);
        quick(a,left,pivot-1);
        quick(a,pivot+1, right);
    }
}
```

- Recurrence?

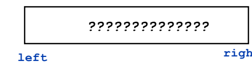


Partition code for quicksort

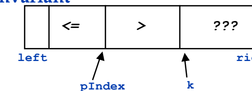
what we want



what we have



invariant



- Easy to develop partition

```
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 loop test is evaluated, used to verify correctness of loop
- Can swap into a[left] before loop
  - Nearly sorted data still ok

Analysis of Quicksort

- Average case and worst case analysis
  - Recurrence for worst case: T(n) = `T(n-1) + T(1) + O(n)`
  - What about average? `T(n) = 2T(n/2) + O(n)`
- Reason informally:
  - Two calls vector size n/2
  - Four calls vector size n/4
  - … 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"

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

```
void foo(int n)
{
  if (0 < n) {
    System.out.println(n);
    foo(n-1);
  }
}
```

```
void foo2(int n)
{
  while (0 < n) {
    System.out.println(n);
    n = n-1;
  }
}
```

- What if print and recursive call switched?
- What about recursive factorial? `return n*factorial(n-1);`

Merge sort: worst case O(n log n)

- Divide and conquer --- recursive sort
  - Divide list/vector into two halves
    - Sort each half
    - Merge sorted halves together
  - What is complexity of merging two sorted lists?
  - What is recurrence relation for merge sort as described?

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

Merge sort: lists or arrays or …

- Mergesort for arrays

```
void mergesort(String[] a, int left, int right)
{
    if (left < right) {
         int mid = (right+left)/2;
         mergesort(a, left, mid);
         mergesort(a, mid+1, right);
         merge(a,left,mid,right);
    }
}
```

- What's different when linked lists used?
  - Do differences affect complexity? Why?
- How does merge work?

Merge for LinkedList

```
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??
```


Merge for linked list (lower case)

```
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?
```


Merge for arrays

- Array code for merge isn't pretty, but it's not hard
  - Mergesort itself is elegant

```
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
  - Can alternate between arrays: one merged into, then swap

Summary of O(n log n) sorts

- Quicksort is relatively straight-forward 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 quicksort
  - Extra storage for array/vector
- Heapsort, more complex to code, good worst case, not stable
  - Basically heap-based priority queue in a vector

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++ we have STL
    - 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 won't quicksort be used?
- Comparators permit sorting criteria to change simply

Non-comparison-based sorts

- lower bound: $\Omega(n \log n)$ 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 ints 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

23 34 56 25 44 73 42 26 10 16

0	1	2	3	4	5	6	7	8	9
10		42	73 23	44 34	25	16 26 56			

10 42 23 73 34 44 25 56 26 16

0	1	2	3	4	5	6	7	8	9
	16 10	26 25 23	34	44 42	56		73		

10 16 23 25 26 34 42 44 56 73

11/08/77

17 Nov 75

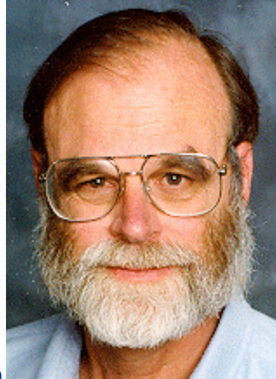
Not needed

Can be tightened considerably

Jim Gray (Turing 1998)

- Bubble sort is a good argument for analyzing algorithm performance. It is a perfectly correct algorithm. But it's performance is among the worst imaginable. *So, it crisply shows the difference between correct algorithms and good algorithms.*

(italics ola's)



Brian Reid (Hopper Award 1982)

Feah. 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.

(*continued*)

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 quicksort library.

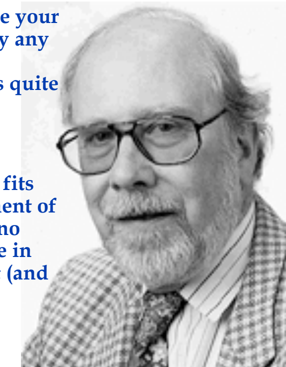
I have a dual-processor 1.2 GHz Powermac and it sneers at your N squared for most interesting values of N. And my source code is smaller than yours.

Brian Reid
who keeps all of his bubbles sorted anyhow.

Niklaus Wirth (Turing award 1984)

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

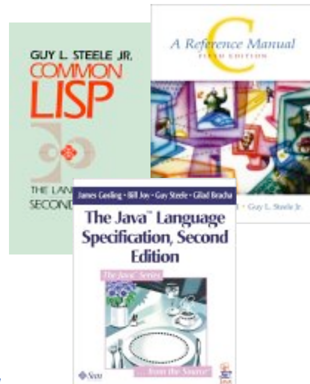
I think BS is popular, because it fits well into a systematic development of sorting algorithms. But it plays no role in actual applications. Quite in contrast to C, also without merit (and its derivative Java), among programming codes.



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

(Thank you for your fascinating paper and inquiry. Here are some off-the-cuff thoughts on the subject.)

I think that one reason for the popularity of Bubble Sort is that it is easy to see why it works, and the idea is simple enough that one can carry it around in one's head …



continued

Guy L. Steele, Jr.

As for its status today, it may be an example of that phenomenon whereby the first widely popular version of something becomes frozen as a common term or cultural icon. Even in the 1990s, a comic-strip bathtub very likely sits off the floor on claw feet.

… it is the first thing that leaps to mind, the thing that is easy to recognize, the thing that is easy to doodle on a napkin, when one thinks generically or popularly about sort routines.