Jeff Nash

Insertion:

Algorithm

```
for i = 2:n,
    for (k = i; k > 1 and a[k] < a[k-1]; k--)
        swap a[k,k-1]
    → invariant: a[1..i] is sorted
end
```

Properties

- Stable
- $O(1)$ extra space
- $O(n^2)$ comparisons and swaps
- Adaptive: $O(n)$ time when nearly sorted
- Very low overhead

Discussion

Although it is one of the elementary sorting algorithms with $O(n^2)$ worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).

For these reasons, and because it is also stable, insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort.

Selection:

Algorithm

```
for i = 1:n,
    k = i
    for j = i+1:n, if a[j] < a[k], k = j
    → invariant: a[k] smallest of a[i..n]
    swap a[i,k]
    → invariant: a[1..i] in final position
end
```

Properties

- Not stable
- $O(1)$ extra space
- $\Theta(n^2)$ comparisons
- $\Theta(n)$ swaps
- Not adaptive

Discussion

From the comparions presented here, one might conclude that selection sort should never be used. It does not adapt to the data in any way (notice that the four animations above run in lock step), so its runtime is always quadratic.

However, selection sort has the property of minimizing the number of swaps. In applications where the cost of swapping items is high, selection sort very well may be the algorithm of choice.

Bubble:

Algorithm

```
for i = 1:n,
    swapped = false
    for j = n:i+1,
        if a[j] < a[j-1],
            swap a[j,j-1]
            swapped = true
    → invariant: a[1..i] in final position
    break if not swapped
end
```

Properties

- Stable
- $O(1)$ extra space
- $O(n^2)$ comparisons and swaps
- Adaptive: $O(n)$ when nearly sorted

Discussion

Bubble sort has many of the same properties as insertion sort, but has slightly higher overhead. In the case of nearly sorted data, bubble sort takes $O(n)$ time, but requires at least 2 passes through the data (whereas insertion sort requires something more like 1 pass).

Shell

Algorithm

```
h = 1
while h < n, h = 3*h + 1
while h > 0,
    h = h / 3
    for k = 1:h, insertion sort a[k:h:n]
    → invariant: each h-sub-array is sorted
end
```

Properties

- Not stable
- $O(1)$ extra space
- $O(n^{3/2})$ time as shown (see below)
- Adaptive: $O(n \cdot lg(n))$ time when nearly sorted

Discussion

The worse-case time complexity of shell sort depends on the increment sequence. For the increments *1 4 13 40 121...*, which is what is used here, the time complexity is $O(n^{3/2})$. For other increments, time complexity is known to be $O(n^{4/3})$ and even $O(n \cdot lg^2(n))$. Neither tight upper bounds on time complexity nor the best increment sequence are known.

Because shell sort is based on insertion sort, shell sort inherits insertion sort's adaptive properties. The adapation is not as dramatic because shell sort requires one pass through the data for each increment, but it is significant. For the increment sequence shown above, there are $log_3(n)$ increments, so the time complexity for nearly sorted data is $O(n \cdot log_3(n))$.

Merge:

Algorithm

```
# split in half
m = n / 2

# recursive sorts
sort a[1..m]
sort a[m+1..n]

# merge sorted sub-arrays using temp array
b = copy of a[1..m]
i = 1, j = m+1, k = 1
while i <= m and j <= n,
    a[k++] = (a[j] < b[i]) ? a[j++] : b[i++]
    → invariant: a[1..k] in final position
while i <= m,
    a[k++] = b[i++]
    → invariant: a[1..k] in final position
```

Properties

- Stable
- $\Theta(n)$ extra space for arrays (as shown)
- $\Theta(lg(n))$ extra space for linked lists
- $\Theta(n \cdot lg(n))$ time
- Not adaptive
- Does not require random access to data

Discussion

Merge sort is very predictable. It makes between $0.5*lg(n)$ and $lg(n)$ comparisons per element, and between $lg(n)$ and $1.5*lg(n)$ swaps per element. The minima are achieved for already sorted data; the maxima are achieved, on average, for random data. If using $\Theta(n)$ extra space is of no concern, then merge sort is an excellent choice: It is simple to implement, and it is the only stable $O(n \cdot lg(n))$ sorting algorithm. Note that when sorting linked lists, merge sort requires only $\Theta(lg(n))$ extra space (for recursion).

Merge sort is the algorithm of choice for a variety of situations: when stability is required, when sorting linked lists, and when random access is much more expensive than sequential access (for example, external sorting on tape).

Heap:

```
# heapify
for i = n/2:1, sink(a,i,n)
→ invariant: a[1,n] in heap order

# sortdown
for i = 1:n,
    swap a[1,n-i+1]
    sink(a,1,n-i)
    → invariant: a[n-i+1,n] in final position
end

# sink from i in a[1..n]
function sink(a,i,n):
    # {lc,rc,mc} = {left,right,max} child index
    lc = 2*i
    if lc > n, return # no children
    rc = lc + 1
    mc = (rc > n) ? lc : (a[lc] > a[rc]) ? lc : rc
    if a[i] >= a[mc], return # heap ordered
    swap a[i,mc]
    sink(a,mc,n)
```

Properties

- Not stable
- $O(1)$ extra space (see discussion)
- $O(n \cdot lg(n))$ time
- Not really adaptive

Discussion

Heap sort is simple to implement, performs an $O(n \cdot lg(n))$ in-place sort, but is not stable.

The first loop, the $\Theta(n)$ "heapify" phase, puts the array into heap order. The second loop, the $O(n \cdot lg(n))$ "sortdown" phase, repeatedly extracts the maximum and restores heap order.

Nearly Sorted:

Sorting nearly sorted data is quite common in practice. Some observations:

- Insertion sort is the clear winner on this initial condition.
- Bubble sort is fast, but insertion sort has lower overhead.
- Shell sort is fast because it is based on insertion sort.
- Merge sort, heap sort, and quick sort do not adapt to nearly sorted data.

Insertion sort provides a O(n^2) worst case algorithm that adapts to O(n) time when the data is nearly sorted. One would like an O(n·lg(n)) algorithm that adapts to this situation; smoothsort is such an algorithm, but is complex. Shell sort is the only sub-quadratic algorithm shown here that is also adaptive in this case.

Reversed:

Discussion

Sorting an array that is initially in reverse sorted order is an interesting case because it is common in practice and it brings out worse-case behavior for insertion sort, bubble sort, and shell sort.

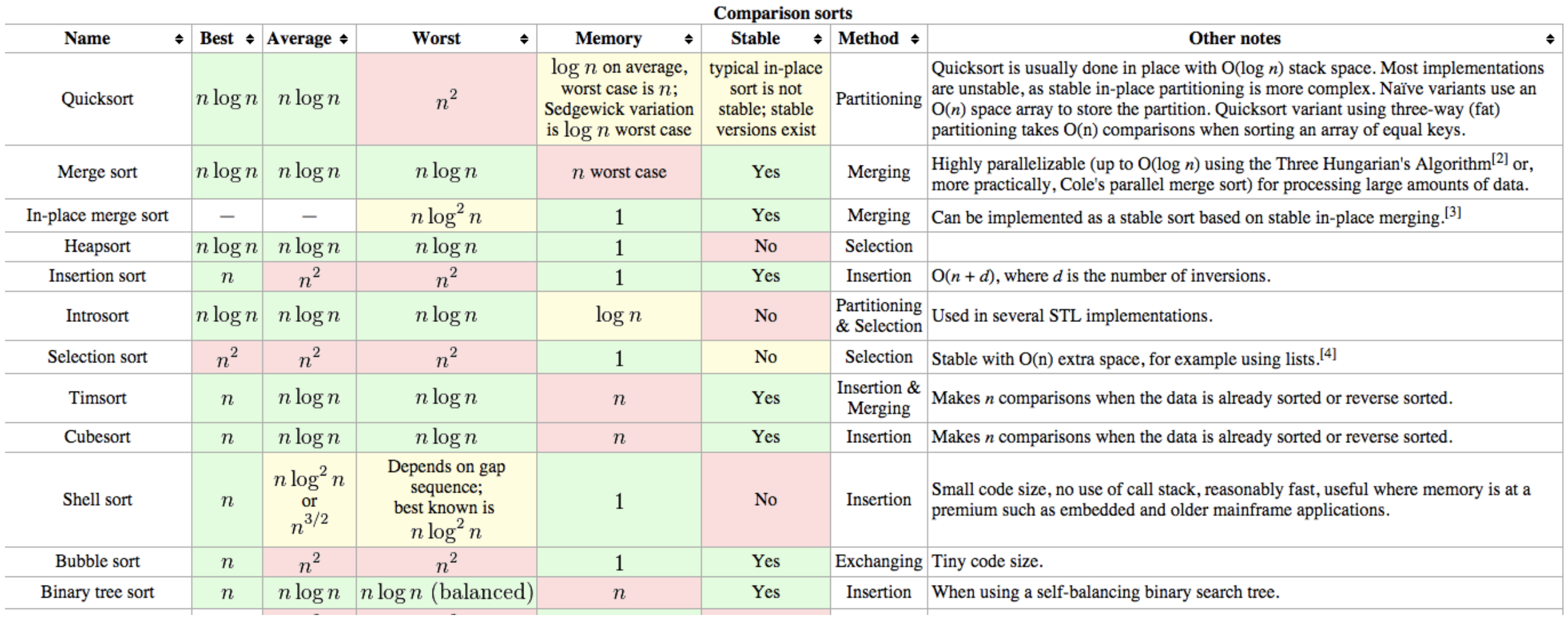
Few Unique:

Discussion

Sorting an array that consists of a small number of unique keys is common in practice. One would like an algorithm that adapts to O(n) time when the number of unique keys is O(1). In this example, there are 4 unique keys.

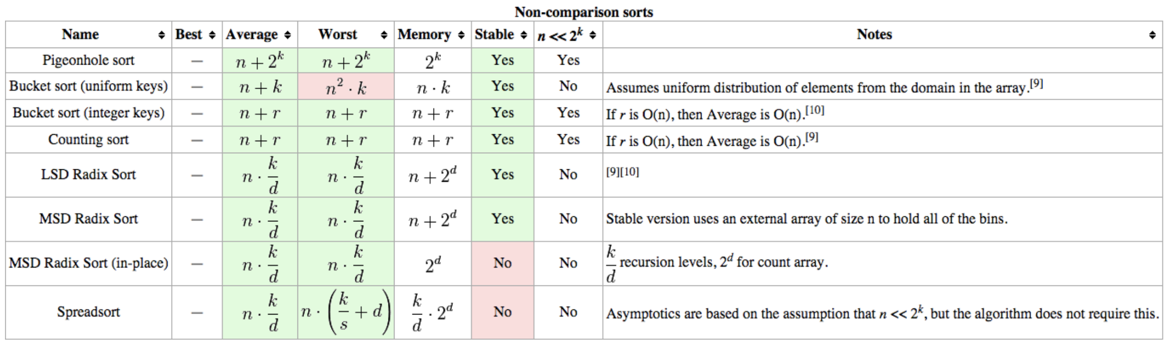
The traditional 2-way partitioning quicksort exhibits its worse-case O(n^2) behavior here. For this reason, any quicksort implementation should use 3-way partitioning, where the array is partitioned into values less than, equal, and greater than the pivot. Because the pivot values need not be sorted recursively, 3-way quick sort adapts to O(n) time in this case.

Shell sort also adapts to few unique keys, though I do not know its time complexity in this case.



Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$ on average, worst case is n; Sedgewick variation is $\log n$ worst case	typical in-place sort is not stable; stable versions exist	Partitioning	Quicksort is usually done in place with O(log *n*) stack space. Most implementations are unstable, as stable in-place partitioning is more complex. Naïve variants use an O(*n*) space array to store the partition. Quicksort variant using three-way (fat) partitioning takes O(n) comparisons when sorting an array of equal keys.
Merge sort	$n \log n$	$n \log n$	$n \log n$	n worst case	Yes	Merging	Highly parallelizable (up to O(log *n*) using the Three Hungarian's Algorithm[2] or, more practically, Cole's parallel merge sort) for processing large amounts of data.
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[3]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	O(*n* + *d*), where *d* is the number of inversions.
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with O(n) extra space, for example using lists.[4]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes *n* comparisons when the data is already sorted or reverse sorted.
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes *n* comparisons when the data is already sorted or reverse sorted.
Shell sort	n	$n \log^2 n$ or $n^{3/2}$	Depends on gap sequence; best known is $n \log^2 n$	1	No	Insertion	Small code size, no use of call stack, reasonably fast, useful where memory is at a premium such as embedded and older mainframe applications.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Binary tree sort	n	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.



Non-comparison sorts

Name	Best	Average	Worst	Memory	Stable	$n << 2^k$	Notes
Pigeonhole sort	—	$n + 2^k$	$n + 2^k$	2^k	Yes	Yes	
Bucket sort (uniform keys)	—	$n + k$	$n^2 \cdot k$	$n \cdot k$	Yes	No	Assumes uniform distribution of elements from the domain in the array.[9]
Bucket sort (integer keys)	—	$n + r$	$n + r$	$n + r$	Yes	Yes	If *r* is O(n), then Average is O(n).[10]
Counting sort	—	$n + r$	$n + r$	$n + r$	Yes	Yes	If *r* is O(n), then Average is O(n).[9]
LSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + 2^d$	Yes	No	[9][10]
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n + 2^d$	Yes	No	Stable version uses an external array of size n to hold all of the bins.
MSD Radix Sort (in-place)	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	2^d	No	No	$\frac{k}{d}$ recursion levels, 2^d for count array.
Spreadsort	—	$n \cdot \frac{k}{d}$	$n \cdot \left(\frac{k}{s} + d\right)$	$\frac{k}{d} \cdot 2^d$	No	No	Asymptotics are based on the assumption that $n << 2^k$, but the algorithm does not require this.