Sorting Algorithm

The ideal sorting algorithm would have the following properties:

Stable: Equal keys aren't reordered.
Operates in place, requiring O(1) extra space.
Worst-case O(n·lg(n)) key comparisons.
Worst-case O(n) swaps.
Adaptive: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.

There is no algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

Insertion Sort

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
```

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.

PROPERTIES

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

Selection Sort

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
```

DISCUSSION
From the comparison 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.

PROPERTIES

- Not stable
- O(1) extra space
- Θ(n^2) comparisons
- Θ(n) swaps
- Not adaptive

Bubble Sort

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
```

DISCUSSION
Bubble sort has many of the same properties as insertion sort, but has a 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).

PROPERTIES

- Stable
- O(1) extra space
- O(n2) comparisons and swaps
- Adaptive: O(n) when nearly sorted

Shell Sort

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
```

DISCUSSION
The worst-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(n3/2). For other increments, time complexity is known to be O(n4/3) and even O(n·lg2(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 adaptation 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 log3(n) increments, so the time complexity for nearly sorted data is O(n·log3(n)).

Because of its low overhead, relatively simple implementation, adaptive properties, and sub-quadratic time complexity, shell sort may be a viable alternative to the O(n·lg(n)) sorting algorithms for some applications when the data to be sorted is not very large.

PROPERTIES

- Not stable
- O(1) extra space
- O(n3/2) time as shown (see below)
- Adaptive: O(n·lg(n)) time when nearly sorted

Merge Sort

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
```

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 Θ(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·lg(n)) sorting algorithm. Note that when sorting linked lists, merge sort requires only Θ(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).

There exist linear time *in-place* merge algorithms for the last step of the algorithm, but they are both expensive and complex. The complexity is justified for applications such as external sorting when Θ(n) extra space is not available.

PROPERTIES

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

Heap Sort

ALGORITHM

```
# 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)
```

DISCUSSION

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

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

The sink function is written recursively for clarity. Thus, as shown, the code requires Θ(lg(n)) space for the recursive call stack. However, the tail recursion in sink() is easily converted to iteration, which yields the O(1) space bound.

Both phases are slightly adaptive, though not in any particularly useful manner. In the nearly sorted case, the heapify phase destroys the original order. In the reversed case, the heapify phase is as fast as possible since the array starts in heap order, but then the sortdown phase is typical. In the few unique keys case, there is some speedup but not as much as in shell sort or 3-way quicksort.

PROPERTIES

Not stable
O(1) extra space (see discussion)
O(n·lg(n)) time
Not really adaptive

Quick Sort

ALGORITHM

```
_# choose pivot_
swap a[1,rand(1,n)]

_# 2-way partition_
k = 1
for i = 2:n, if a[i] < a[1], swap a[++k,i]
swap a[1,k]
_→ invariant: a[1..k-1] < a[k] <= a[k+1..n]_

_# recursive sorts_
sort a[1..k-1]
sort a[k+1,n]
```

DISCUSSION

When carefully implemented, quick sort is robust and has low overhead. When a stable sort is not needed, quick sort is an excellent general-purpose sort – although the 3-way partitioning version should always be used instead.

The 2-way partitioning code shown above is written for clarity rather than optimal performance; it exhibits poor locality, and, critically, exhibits O(n2) time when there are few unique keys. A more efficient and robust 2-way partitioning method is given in Quicksort is Optimal by Robert Sedgewick and Jon Bentley. The robust partitioning produces balanced recursion when there are many values equal to the pivot, yielding probabilistic guarantees of O(n·lg(n)) time and O(lg(n)) space for all inputs.

With both sub-sorts performed recursively, quick sort requires O(n) extra space for the recursion stack in the worst case when recursion is not balanced. This is exceedingly unlikely to occur, but it can be avoided by sorting the *smaller* sub-array recursively first; the second sub-array sort is a tail recursive call, which may be done with iteration instead. With this optimization, the algorithm uses O(lg(n)) extra space in the worst case.

PROPERTIES

Not stable
O(lg(n)) extra space (see discussion)
O(n2) time, but typically O(n·lg(n)) time
Not adaptive

Quick Sort 3 Way

ALGORITHM

```
_# choose pivot_
swap a[n,rand(1,n)]

_# 3-way partition_
i = 1, k = 1, p = n
while i < p,
  if a[i] < a[n], swap a[i++,k++]
  else if a[i] == a[n], swap a[i,--p]
  else i++
end
_→ invariant: a[p..n] all equal_
_→ invariant: a[1..k-1] < a[p..n] < a[k..p-1]_

_# move pivots to center_
m = min(p-k,n-p+1)
swap a[k..k+m-1,n-m+1..n]

_# recursive sorts_
sort a[1..k-1]
sort a[n-p+k+1,n]
```

DISCUSSION

The 3-way partition variation of quick sort has slightly higher overhead compared to the standard 2-way partition version. Both have the same best, typical, and worst case time bounds, but this version is highly adaptive in the very common case of sorting with few unique keys.

The 3-way partitioning code shown above is written for clarity rather than optimal performance; it exhibits poor locality, and performs more swaps than necessary. A more efficient but more elaborate 3-way partitioning method is given in Quicksort is Optimal by Robert Sedgewick and Jon Bentley.

When stability is not required, quick sort is the general purpose sorting algorithm of choice. Recently, a novel dual-pivot variant of 3-way partitioning has been discovered that beats the single-pivot 3-way partitioning method both in theory and in practice.

PROPERTIES

Not stable
O(lg(n)) extra space
O(n2) time, but typically O(n·lg(n)) time
Adaptive: O(n) time when O(1) unique keys