## Insertion Sort

## Algorithm

→ invariant: a[1..i] is sorted end -->

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(n2) 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(n2) 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.

**Merge sort**

**definition:** A [k-way merge sort](http://www.itl.nist.gov/div897/sqg/dads/HTML/kwayMergeSort.html) that sorts a data stream using repeated merges. It distributes the input into two streams by repeatedly reading a [block](http://www.itl.nist.gov/div897/sqg/dads/HTML/block.html) of input that fits in memory, a run, sorting it, then writing it to the next stream. It merges runs from the two streams into an output stream. It then repeatedly distributes the runs in the output stream to the two streams and merges them until there is a single sorted output.

## 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
* 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

## 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 do 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.

**Selection Sort**

## Algorithm

void selection\_sort(int num[SIZE]) {

int i, j, min;

for (i = 0; i < (SIZE-1); i++) {

min = i;

for (j = (i+1); j < SIZE; j++) {

if(num[j] < num[min]) {

min = j;

}

}

if (i != min) {

int swap = num[i];

num[i] = num[min];

num[min] = swap;

} }

}

## Properties

* Not stable
* O(1) extra space
* Θ(n2) comparisons
* Θ(n) swaps

## 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 Sort**

## Algorithm

**procedure** bubbleSort( A **:** list of sortable items ) **defined as:**

**do**

swapped := false

**for each** i **in** 0 **to** length(A) - 2 **inclusive do:**

**if** A[i] > A[i+1] **then**

swap( A[i], A[i+1] )

swapped := true

**end if**

**end for**

**while** swapped

**end procedure**

## Properties

* Stable
* O(1) extra space
* O(n2) 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).

**Heap Sort**

## Algorithm

***Max-heapify(A,i) O(lg n)***

* *L=left(i)*
* *R=right(i)*
* *If L<=heap-size[A] and A[L]>A[i]*
  + *then largest=L*
  + *else largest=i*
* *If R<=heap-size[A] and A[R]>A[largest]*
  + *then largest=R*
* *If largest≠i*
  + *then exchange A[i]↔A[largest]*
* *Max-heapify(A,largest)*

***Build-Max-Heap(A) O(n)***

* *Heap-size[A]=length[A]*
* *For i=└length[A]/2┘ downto 1*
  + *Do max-heapify(A,i)*

***Heapsort(A) O(nlg n)***

* *Build-Max-Heap(A)*
* *For i=length[A] downto 2*
  + *Do exchange A[1]↔A[i]*
  + *Heap-size[A]= Heap-size[A]-1*
  + *Max-heapify(A,i)*

## Properties

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

## 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.

**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]

## Properties

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

## 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](http://www.sorting-algorithms.com/static/QuicksortIsOptimal.pdf) 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.

Radix sort(A,d)

For i=1 to d

Do use stable sort to sort array A on digit i.

Θ(d(n+k)) when d is constant and k=O(n) radix sort runs in linear time. Each digit can take on up to k possible values.