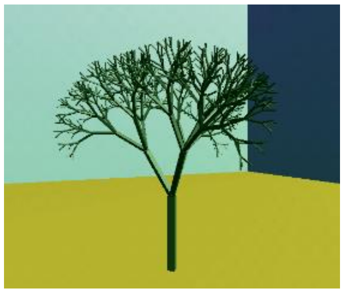


GATS Companion to: Searching and Sorting

Author: Garth Santor
Editors: Trinh Hān
Copyright Dates: 2020, 2019, 1991
Version: 1.1.0 (2020-02-29)

Overview

Searching and *sorting* are some of the most fundamental algorithms in computing, Donald Knuth devotes an entire volume of *The Art of Computer Programming* to it. The subjects naturally go together because they operated on the same data structures and the efficiency of searches are dependent on the order of the data. Here I will present many common searching sorting algorithms and provide a commentary for programmers.

Searching

Search algorithms are used to retrieve data stored within a data structure, or calculated in the search space of a problem domain. We'll only be looking that retrieval algorithms here.

Linear Searching

The simplest search algorithm is the linear search; which starts at one end of the data structure and through brute force, tests each element in turn until the target has been found.

Time Complexity

Best Case	$O(1)$	If we are lucky, the first element we look at will be the target element.
Average Case	$O\left(\frac{n}{2}\right)$	We may be lucky and find the target in the first spot, or unlucky and find the target in the last spot, or any spot in between. Any location is as likely as any other. The average case is sum of all possible time complexity outcomes divided by the number of different outcomes: $$\frac{O(1)+O(2)+\cdots+O(n-1)+O(n)}{n}=O\left(\frac{1+2+\cdots+(n-1)+n}{n}\right)$$ $$=O\left(\frac{\left(\frac{n(n+1)}{2}\right)}{n}\right)=O\left(\frac{n(n+1)}{2}\cdot\frac{1}{n}\right)=O\left(\frac{n+1}{2}\right)=O\left(\frac{n}{2}\right)$$
Worst Case	$O(n)$	The worst case is finding the target element at the very end of the search.

Implementation – Array

This implementation returns the index of the target element if found, or the size of the vector if not.

```
function LINEAR-SEARCH( A, key ) returns a natural number
    for $i \leftarrow 1$ to $length\{A\} - 1$ do
        if $A[i] = key$
            then return $i$
    return $length\{A\}$
```

Implementation – Pointer

This implementation returns the index of the target element if found, or the size of the vector if not.

```
function LINEAR-SEARCH( beg, end, key ) returns a pointer
    $current \leftarrow beg$
    while $current \neq end$ do
        if $value\{current\} = key$
            then return $current$
    return $end$
```

Binary Searching

Binary search runs in $O(\log_2 n)$ time on sorted data sets. An unsuccessful search requires $\lfloor \log N \rfloor + 1$ iterations of the loop or calls to the function.

Time Complexity

Best Case	$O(1)$	If we are lucky, the first element we look at will be the target element.
Average Case	$O(\log_2 n)$	Each iteration or recursion eliminates half of the remaining elements.
Worst Case	$O(\log_2 n)$	The worst case is to narrow down the list to a sub-list of one.

Implementation – Array, Recursive

This implementation returns the index of the target element if found, or the size of the vector if not.

```
function BINARY-SEARCH( A, key, low, high ) returns a natural number
    if $low \leq high$
        then    $middle \leftarrow (low + high)$ div 2
            if $key = A[middle]$          then   return $middle$
            else if $key < A[middle]$    then   return BINARY-SEARCH( $A, key, low, middle - 1$ )
            else        return BINARY-SEARCH( $A, key, middle + 1, high$ )
        else return $length\{A\}$
```

Implementation – Array, Iterative

This implementation returns the index of the target element if found, or the size of the vector if not.

```
function BINARY-SEARCH( A, key, low, high ) returns a natural number
    while $low \leq high$ do
        $middle \leftarrow (low + high)$ div 2
        if $key = A[middle]$          then    return $middle$
        else if $key < A[middle]$    then    $high \leftarrow middle - 1$
        else        $low \leftarrow middle + 1$
    return $length\{A\}$
```

Implementation – Array, Iterative

A potential speed improvement can be found by reducing the amount of branching inside the loop. In this version we always search down to a sub-list of size one, then test to see if the remaining element is the key.

```
function BINARY-SEARCH( A, key, low, high ) returns a natural number
    while low < high do
        middle ← (low + high) div 2
        if A[middle] < key
            then  low ← middle + 1
            else  high ← middle – 1
    if low = high and A[ low ] = key
        then  return low
        else  return length{A}
```

Sorting Algorithms

Introduction

Sorting is an operation that places things into some natural order.

Sorts are said to be stable if the elements with the same key value appear in the output collection in the same order that they appear in the input collection. Sort stability is important when sorting a collection several times, each time by a different key. (E.g. firstly by day, secondly by month, lastly by year)

Name, best, average, worst, memory, stable, method, comparison, notes

Name	Best	Average	Worst	Memory	Stable	Method	Comparison	Notes
Bogosort	$O(n)$	$O(n \cdot n!)$	unbounded	$O(1)$	No	n/a	Yes	
Bubble	$O(n)$	—	$O(n^2)$	$O(1)$	Yes	Exchanging	Yes	Short-circuit variant
Cocktail	$O(n)$	—	$O(n^2)$	$O(1)$	Yes	Exchanging	Yes	
Comb	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	No	Exchanging	Yes	
Heapsort	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	No	Selection	Yes	
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Insertion	Yes	Can be performed on serialized input
Merge	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	Yes	Merging	Yes	Can be implemented for external sorting
Permutation	$O(n)$	$O(n \cdot n!)$	$O(n \cdot n!)$	$O(1)$	No	Selection	No	
Pigeonhole	$O(n + 2^k)$	$O(n + 2^k)$	$O(n + 2^k)$	$O(2^k)$	Yes	Tabulation	No	Assumes $n < 2^k$
Quicksort	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	No	Partitioning	Yes	
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection	Yes	
Shell's	—	—	$O(n^{1.5})$	$O(1)$	No	Insertion	Yes	

Is-Sorted – random-access container

‘IS-SORTED’ determines if a collection is sorted. The first algorithm is implemented for a *random-access collection*.

```
function IS-SORTED( A ) returns a Boolean
    for i ← 1 to length{A} – 1 do
        if A[i] < A[i – 1] then
            return false
    return true
```

Is-Sorted – forward iterators

```
**function** IS-SORTED( *beg*, *end* ) **returns** a Boolean
    if $beg = end$
        then return true
    $previous \leftarrow beg$
    $current \leftarrow successor\{beg\}$
    while $current \neq end$ do
        if $value\{current\} < value\{previous\}$
            then return false
        $previous \leftarrow current$
        $current \leftarrow successor\ \{current\}$
    return true
```

Bogo Sort

'BOGO-SORT' randomly shuffles the elements of the collection until satisfies 'IS-SORTED'.

```
procedure BOGO-SORT( $A$ )
    while not IS-SORTED( $A$ )
        RANDOMIZE( $A$ )
```

Is this the worst sorting algorithm of all time? Perhaps its not even an algorithm but is more likely an heuristic, and a bad one.

References

http://en.wikipedia.org/wiki/Bogosort

Bubble Sort

'Bubble-Sort' is the worst sorting algorithm in common use. It is both move intensive and comparison intensive. However, its lack-lustre performance has never severely deterred novices from using it.

... in fact, the Bubblesort has hardly anything to recommend it except its catchy name. (Wirth, 1976)

Algorithm

```
1.  repeat for as many times as there are elements less one;
2.      for each element pair;
3.          compare $K_i$ with $K_{i+1}$, interchanging $R_i$ with $R_{i+1}$ if the keys are out of order.
```

Where K represents keys and R represents records.

Implementation – random-access container (brute force)

This first implementation is a *brute-force* translation of the algorithm for a *random-access container*.

```
procedure BUBBLE-SORT( $A$ )
    for $pass \leftarrow 1$ to $length\{A\} - 1$ do
        for $idxPair \leftarrow 1$ to $length\{A\} - 1$ do
            if $A[idxPair] < A[idxPair - 1]$
                then exchange $A[idxPair] \leftrightarrow A[idxPair - 1]$
```

This can easily be improved upon by recognizing that each pass places a minimum of one element into its sorted place, (i.e. the sorted portion grows from the end to the beginning). Any further testing of those elements is redundant.

Implementation – random-access container (standard)

This second implementation removes those extra comparisons.

```
procedure BUBBLE-SORT( $A$ )
    for $pass \leftarrow 1$ to $length\{A\} - 1$ do
        for $idxPair \leftarrow 1$ to $length\{A\} - pass$ do
            if $A[idxPair] < A[idxPair - 1]$
                then exchange $A[idxPair] \leftrightarrow A[idxPair - 1]$
```

The next refinement removes the calculation within the inner loop's termination condition.

```
procedure BUBBLE-SORT( A )
    for $idxEndOfPass \leftarrow length\{A\} - 1$ downto 1 do
        for $idxPair \leftarrow 1$ to $idxEndOfPass$ do
            if $A[idxPair] < A[idxPair - 1]$
                then exchange $A[idxPair] \leftrightarrow A[idxPair - 1]$
```

Implementation – bi-directional iterators

Now we can write the equivalent implementation for *bi-directional iterators*.

```
procedure BUBBLE-SORT( $beg$, $end$ )
    $firstPair \leftarrow successor\{beg\}$
    while $firstPair \neq end$ do
        $previous \leftarrow beg$
        $current \leftarrow firstPair$
        while $current \neq end$ do
            if $value\{current\} < value\{previous\}$
                then exchange $value\{current\} \leftrightarrow value\{previous\}$
            $previous \leftarrow current$
            $current \leftarrow successor\{current\}$
        $end \leftarrow predecessor\{end\}$
```

Bubble sort's performance on sorted and partially sorted lists can be dramatically improved by tracking the occurrences of exchanges in each pass. If no exchanges occur, the then the list is already sorted and the algorithm could immediately be terminated.

This algorithm performs particularly poorly on reversed lists, since every element pair is swapped on every pass. The tests simply add overhead.

Implementation – random-access container (short-circuit)

```
procedure BUBBLE-SORT( $A$ )
    for $idxEndOfPass \leftarrow length\{A\} - 1$ downto 1 do
        $hasSwapped \leftarrow$ false
        for $idxPair \leftarrow 1$ to $idxEndOfPass$ do
            if $A[idxPair] < A[idxPair - 1]$
                then    exchange $A[idxPair] \leftrightarrow A[idxPair - 1]$
                        $hasSwapped \leftarrow$ true
        if $hasSwapped$ is false
            then return
```

Implementation – bi-directional iterators (short-circuit)

The *bi-directional iterator* implementation:

```
procedure BUBBLE-SORT( beg, end )
    firstPair ← successor{beg}
    hasSwapped ← true
    while firstPair ≠ end and hasSwapped do
        hasSwapped ← false
        previous ← beg
        current ← firstPair
        while current ≠ end do
            if value{current} < value{previous}
                then  exchange value{current} ↔ value{previous}
                      hasSwapped ← true
            previous ← current
            current ← successor{current}
        end ← predecessor{end}
```

References

http://en.wikipedia.org/wiki/Bubble_sort

Cocktail-shaker

The cocktail-shaker sort is a variation of the bubble sort. The difference is that the bubble passes are performed in both directions as opposed to bubble sort's forward only passes.

Algorithm

```
1.  while the unsorted range is greater than 1
2.      for each element pair in the unsorted range;
3.          compare $K_i$ with $K_{i+1}$, interchanging $R_i$ with $R_{i+1}$ if the keys are out of order.
4.      for each element pair in the unsorted range;
5.          compare $K_i$ with $K_{i-1}$, interchanging $R_i$ with $R_{i-1}$ if the keys are out of order.
6.      reduce the unsorted range by 1 at each end
```

Where K represents keys and R represents records.

Implementation – random-access container (naïve)

The first implementation is the direct translation of the algorithm.

```
procedure COCKTAIL-SHAKER-SORT( A )
    idxFirstUnsorted ← 0
    idxLastUnsorted ← length{A} – 1
    idxCurrent ← idxFirstUnsorted
    while idxFirstUnsorted < idxLastUnsorted do
        while idxCurrent < idxLastUnsorted do
            if A[idxCurrent + 1] < A[idxCurrent]
                then  exchange A[idxCurrent + 1] ↔ A[idxCurrent]
            idxCurrent ← idxCurrent + 1
        idxLastUnsorted ← idxLastUnsorted – 1
        while idxCurrent > idxFirstUnsorted do
            if A[idxCurrent – 1] > A[idxCurrent]
                then exchange A[idxCurrent – 1] ↔ A[idxCurrent]
            idxCurrent ← idxCurrent – 1
        idxFirstUnsorted ← idxFirstUnsorted + 1
```

In the case of an empty container, *idxLastUnsorted* is initialized to –1. However, if the index type is unsigned, numeric underflow occurs causing to be initialized to std::numeric_limits<unsigned>::max() – a clearly invalid location. The algorithm must either be implemented using a signed range type or be rewritten.

Implementation – random-access container (safe)

```
procedure COCKTAIL-SHAKER-SORT( A )
    idxFirstUnsorted ← 0
    idxEnd ← length{A}
    idxCurrent ← idxFirstUnsorted
    while idxFirstUnsorted < idxEnd do
        idxCurrent ← idxCurrent + 1
        while idxCurrent < idxEnd do
            if A[idxCurrent] < A[idxCurrent – 1]
                then exchange A[idxCurrent – 1] ↔ A[idxCurrent]
            idxCurrent ← idxCurrent + 1
        idxEnd ← idxEnd – 1
        idxCurrent ← idxCurrent – 1
        while idxCurrent > idxFirstUnsorted do
            if A[idxCurrent] < A[idxCurrent – 1]
                then exchange A[idxCurrent – 1] ↔ A[idxCurrent]
            idxCurrent ← idxCurrent – 1
        idxFirstUnsorted ← idxFirstUnsorted + 1
```

Implementation – bi-directional iterators

```
procedure COCKTAIL-SHAKER-SORT( beg, end )
    if beg = end then
        return
    previous ← beg
    next ← successor{beg}
    while next ≠ end do
        while next ≠ end do
            if value{previous} > value{next}
                then exchange value{previous} ↔ value{next}
            previous ← successor{previous}
            next ← successor{next}
        previous ← predecessor{previous}
        next ← predecessor{next}
        end ← predecessor{end}
        if previous = beg
            then return
        repeat
            previous ← predecessor{previous}
            next ← predecessor{next}
            if value{previous} > value{next}
                then exchange value{previous} ↔ value{next}
        until previous = beg
        previous ← successor{previous}
        next ← successor{next}
        beg ← successor{beg}
```

References

http://en.wikipedia.org/wiki/Cocktail_sort

Comb Sort

The comb sort is a variation of the bubble sort first published in *Byte* by Steven Lacy and Richard Box (Stephen Lacy and Richard Box, "A Fast, Easy Sort", Byte, April 1991, p.315). Comb sort can also be thought of as a variation of Shell's sort. It uses Shell's idea of enhancing the performance of the insert by inserting in steps or shells of a magnitude greater than 1. The comb sort uses bubble passes that compare non-contiguous elements; thereby moving an element closer to its correct resting spot in fewer moves. Lacy and Box investigated the gap reduction factor and found that shrink factor of 1.3 was ideal and that further performance gains occur in the gap values of 9 and 10 are exchanged for 11.

Implementation – random-access container

```
procedure COMB-SORT( $A$ )
    $gap \leftarrow length\{A\}$
    loop
        $gap \leftarrow \lfloor gap \div 1.3 \rfloor$
        if $gap = 9$ or $gap = 10$
            then $gap \leftarrow 11$
        else if $gap < 1$
            then $gap \leftarrow 1$
        $swapped \leftarrow$ false
        for $i \leftarrow 0$ to $length\{A\} - gap - 1$ do
            $j \leftarrow i + gap$
            if $A[\,i\,] > A[\,j\,]$
                then exchange $A[\,i\,] \leftrightarrow A[\,j\,]$
                     $swapped \leftarrow$ true
        if $gap = 1$ and not $swapped$
            then return
```

Note that the container can not be empty if the range type is unsigned.

References

http://en.wikipedia.org/wiki/Comb_sort

Count Sort

See ‘Pigeonhole Sort’

Heap Sort

‘Heap sort’ is what you would call a ‘good performer’. Its sorts in $O(n \log n)$ time – always!

The heap sort works by first arranging all of the elements of the container into a heap. A heap is binary tree in which the key values of the child nodes are both less than the key value of the parent node.

A binary tree structure can be implemented in a 1-base array by employing the relationship of *left-child-index* is double the *parent-index*, *right-child-index* is one greater than the *left-child-index*.

Once formed, the heap can be converted to a sorted list by swapping the element at the top of the heap with the last element in the heap; shrinking the heap by one; then fixing the heap by recursively adjusting the top element until the heap has been restored.

Implementation – random-access container (iterative)

```
procedure HEAP-SORT( $A$ )
    $heapSize \leftarrow length\{A\}$
    for $parentNode \leftarrow heapSize$ div 2 downto 1 do
        HEAPIFY( $A$, $parentNode$, $heapSize$ )
    for $idx \leftarrow length\{A\} - 1$ downto 1 do
        exchange $A[0] \leftrightarrow A[idx]$
        $heapSize \leftarrow heapSize - 1$
        HEAPIFY( $A$, 1, $heapSize$ )
```

1.

```
procedure HEAPIFY( A, parentNode, heapSize )
    loop
        leftNode ← parentNode × 2
        if leftNode > heapSize
            then    return
        if A[leftNode – 1] > A[parentNode – 1]
            then    largestNode ← leftNode
            else largestNode ← parentNode
        rightNode ← leftNode + 1
        if rightNode ≤ heapSize and A[rightNode – 1] > A[largestNode – 1]
            then    largestNode ← rightNode
        if largestNode ≠ parentNode
            then    exchange A[largestNode – 1] ↔ A[parentNode – 1]
                parentNode ← largestNode
            else return
```

Limited Heap Sort

In web search engines we often want only the *k* best results presented in a sorted list, where there may have been as many as $n > k$ results. There is no point in storing all *n* results, sorting them all then throwing away the $n – k$ excess elements.

A limited heap sort, sorts the *k* highest elements of a set of *n* elements. We first feed *k* elements into the heap and build an ascending order heap (*the lowest value at the top of the heap*). For the remaining elements we compare the new element to the top element of the heap replacing the top element if the new element is larger – then heapify. Once all elements have been tested and inserted into the heap, we complete the heap sort producing an ascending order list.

We can then reverse the list to produce a descending order list if necessary.

References

http://en.wikipedia.org/wiki/Heapsort

Insertion Sort

Insertion sort is an attempt to minimize the number of comparisons performed during a sort by shifting through sorted space when positioning an element.

Algorithm

```
1.  for each element $E_i$ in the array;
2.      we assume that the preceding elements $E_1, \ldots, E_{i-1}$ have already been sorted;
3.      we insert $E_i$, into its proper place among the sorted records
```

where *E* represents keys

Implementation – random-access container

Here is an implementation for a *random-access container*.

```
procedure INSERTION-SORT( A )
    for idxFirstUnsorted ← 1 to length{A} – 1 do
        idx ← idxFirstUnsorted
        while idx > 0 and A[idx] < A[idx – 1] do
            exchange A[idx] ↔ A[idx – 1]
            idx ← idx – 1
```

Notes

For empty arrays the first line of the algorithm results in an index being created that has a negative value (i.e. –1). If your implementing language's index type is an unsigned type this statement would produce numeric underflow.

Implementation – bi-directional iterators

```
procedure INSERTION-SORT( $beg, end$ )
    if $beg = end$ then
        return
    $firstUnsorted \leftarrow beg$
    loop
        $firstUnsorted \leftarrow successor\{firstUnsorted\}$
        if $firstUnsorted = end$
            then return
        $element \leftarrow firstUnsorted$
        $previous \leftarrow predecessor\{element\}$
        while $element \neq beg$ and $value\{element\} < value\{previous\}$ do
            exchange $value\{element\} \leftrightarrow value\{previous\}$
            $element \leftarrow predecessor\{element\}$
            $previous \leftarrow predecessor\{previous\}$
```

This implementation does present a challenge when converted to code since many modern implementations of iterators have ‘safe’ debug versions that test to see that the iterators stay within bounds. This algorithm violates that rule on the final line.

Implementation – bi-directional iterators (safe)

```
procedure INSERTION-SORT( $beg, end$ )
    if $beg = end$ then
        return
    $firstUnsorted \leftarrow beg$
    loop
        $firstUnsorted \leftarrow successor\{firstUnsorted\}$
        if $firstUnsorted = end$
            then return
        $element \leftarrow firstUnsorted$
        $previous \leftarrow element$
        while $element \neq beg$ do
            $previous \leftarrow predecessor\{previous\}$
            if $value\{element\} \geq value\{previous\}$
                then break
            exchange $value\{element\} \leftrightarrow value\{previous\}$
            $element \leftarrow predecessor\{element\}$
```

This is adjustment produces some ugly code, but languages that permit assignment inside loop conditions can significantly clean up the implementation:

Implementation – bi-directional iterators (elegant)

```
procedure INSERTION-SORT( $beg, end$ )
    if $beg = end$ then
        return
    $firstUnsorted \leftarrow beg$
    loop
        $firstUnsorted \leftarrow successor\{firstUnsorted\}$
        if $firstUnsorted = end$
            then return
        $previous \leftarrow element \leftarrow firstUnsorted$
        while $element \neq beg$ and $value\{element\} < value\{\ previous \leftarrow predecessor\{previous\}\ \}$ do
            exchange $value\{element\} \leftrightarrow value\{previous\}$
            $element \leftarrow predecessor\{element\}$
```

References

http://en.wikipedia.org/wiki/Insertion_sort

Merge Sort

Merge sort is the original *divide and conquer* sort. It executes in $O(n \log_2 n)$ time whether sorting vectors, linked lists or files. The vector and file implementations require additional storage of equal size to process the merge; the linked list version does not.

Merge sort can be processed using recursion or iteration. The recursive version is not easily adapted to files while the iterative version adapts easily to all data types and stores.

Algorithm

1. If the list size is 0 or 1, return.
2. Split the list into 2 equal parts.
3. Merge-Sort each half-list separately.
4. Merge the two sorted lists into a single list.

Implementation – Random-access Container Using Recursion

```
procedure MERGE-SORT( A, beg, end )
    if end – beg > 1 then
        split ← (beg + end) div 2
        MERGE-SORT( A, beg, split )
        MERGE-SORT( A, split, end )
        MERGE( A, beg, split, end )
```

```
procedure MERGE( A, low, high, end)
    create a output array D the same size of A
    lowWalker ← low
    highWalker ← high
    destWalker ← low
    while lowWalker < high and highWalker < end
        if A[lowWalker] < A[highWalker]
            then  D[destWalker] ← A[lowWalker]
                  lowWalker ← lowWalker + 1
            else  D[destWalker] ← A[highWalker]
                  highWalker ← highWalker + 1
        destWalker ← destWalker + 1
    if lowWalker < high
        then    D[destWalker .. end – 1] ← A[lowWalker .. high – 1]
        else    D[destWalker .. end – 1] ← A[highWalker .. end – 1]
    A ← D
```

Optimization – Pre-allocate the copy vector.

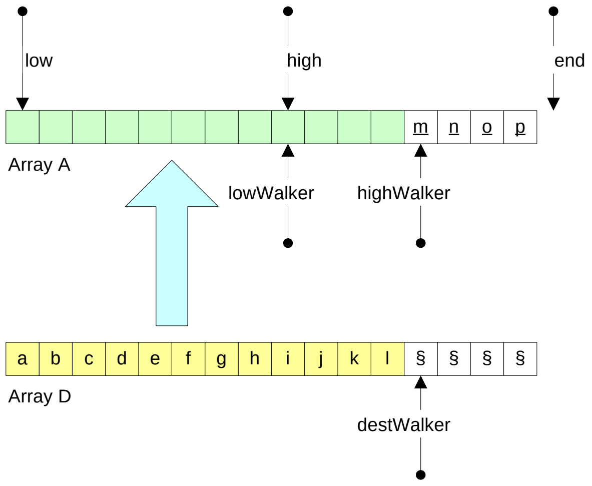
The biggest performance hit in the preceding implementation is the constant reallocations of the hold array – D. The simple solution is to have the driver function create D and then pass it to the merge function.

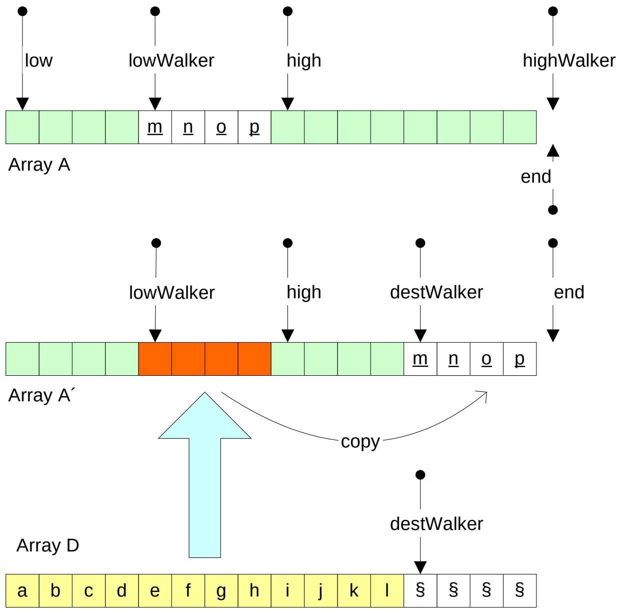
Implementation – Random-access Container Using Iteration

```
procedure MERGE-SORT( $A$ )
    $sublistSize \leftarrow 1$
    while $sublistSize < length\{A\}$ do
        $low \leftarrow 0$
        while $low < length\{A\}$ do
            $high \leftarrow low + sublistSize$
            $end \leftarrow high + sublistSize$
            if $end > length\{A\}$
                then    $end \leftarrow length\{A\}$
                    if $high > length\{A\}$
                        then $high \leftarrow length\{A\}$
            if $high \neq length\{A\}$
                then MERGE( $A, low, high, end$ )
            $low \leftarrow low + 2 \times sublistSize$
        $sublistSize \leftarrow 2 \times sublistSize$
```

Optimization – Random-access Container Using Iteration with Short-circuiting

Another optimization is the recognition that there is unnecessary copying during the merge when the lower half of the array empties first. The data remaining in the top half of the array is already in the correct position, so copying those elements to the temporary store would be pointless.





```
procedure MERGE( $A$, $low$, $high$, $end$ )
    $lowWalker \leftarrow low$
    $highWalker \leftarrow high$
    $destWalker \leftarrow low$
    while $lowWalker < high$ and $highWalker < end$
        if $A[lowWalker] < A[highWalker]$
            then    $D[destWalker] \leftarrow A[lowWalker]$
                $lowWalker \leftarrow lowWalker + 1$
            else $D[destWalker] \leftarrow A[highWalker]$
                $highWalker \leftarrow highWalker + 1$
        $destWalker \leftarrow destWalker + 1$
    if $lowWalker < high$
        then $A[destWalker .. end - 1] \leftarrow A[lowWalker .. high - 1]$
    $A[low .. destWalker - 1] \leftarrow D[low .. destWalker - 1]$
```

Note that the block copy in line 12 must be a high to low copy since the ranges could overlap.

Implementation – Linked List with Iteration

```
procedure MERGE-SORT( $L$ )
    $sublistSize \leftarrow 1$
    while $sublistSize < length\{L\}$ do
        empty the $holdList$
        while $L$ is not empty do
            split a list of no greater size than $sublistSize$ from the front of $L$
            split another list of no greater size than $sublistSize$ from the front of $L$
            Merge the two lists
            splice the merged list onto the end of the $holdList$
        exchange $holdList \leftrightarrow L$
        $sublistSize \leftarrow sublistSize \times 2$
```

References

http://en.wikipedia.org/wiki/Merge_sort

Ordersort

The *ordersort* algorithm sorts by determining the order in which elements should be arranged, then arranging them into that order. The arrangement order is determined by locating the immediate predecessors and successors of each element.

References

C/C++ Users Journal (September 2005)

Permutation Sort

Permutation sort is a trial-and-error type sort. Try every possible arrangement of the elements – at least one of them is sorted.

Pigeonhole Sort

Pigeonhole sort is a special case sort capable of sorting in linear time. Counting sort requires that the input array being sorted only contains ordinal values in a known and finite range. For our discussion we will assume that values are in the range [0..k]. The idea is to count the number of instances of each value in the input array. Then for each value detected in the input array, write back to the array the correct number of instances detected. However, we write the input values back in their sorted order.

One pass of the input array is required to accumulate the frequency table of values, and one more pass is required to write the values back to the array. This simple approach destroys the original elements and is therefore of little practical use. The sort can be modified to copy the original values from one array to another.

Implementation – random-access container

```
procedure PEGEONHOLE-SORT( $A$, $k$ )
    for $i \leftarrow 0$ to $k$ do
        $C[\ i\ ] \leftarrow 0$
    for $i \leftarrow 0$ to $length\{A\} - 1$ do
        $C[\ A[\ i\ ]\ ] \leftarrow C[\ A[\ i\ ]\ ] + 1$
    $j \leftarrow 0$
    for $i \leftarrow 0$ to $length\{C\} - 1$ do
        while $C[\ i\ ] > 0$ do
            $C[\ i\ ] \leftarrow C[\ i\ ] - 1$
            $A[\ j\ ] \leftarrow i$
            $j \leftarrow j + 1$
```

References

http://en.wikipedia.org/wiki/Pigeonhole_sort

Quick sort

'Quick-sort', as named by its originator – Sir Anthony (C.A.R.) Hoare (Oxford University) is one of the smallest, fastest and most elegant algorithms known.

The basic idea is to perform 'swapping passes' instead of search and swap. During each 'swapping-pass', low-value elements in the top half of the list are swapped for high-value elements in the bottom half of the list. Eventually the swapping pass would come to a middle point such that all of the values in the bottom half of the list will have a value lower than *every* value in the top half of the list. The reverse – that all of the values in the top half of the list will have a value higher than *every* value in the bottom half of the list – is necessarily true. The two sub-lists could then be individually 'quick-sorted'.

Algorithm

1. If the size of the list is 0 or 1, return.
2. Pick *any* element from the list. Call this value '*pivot*.' ($v \in A$)
3. Partition the list into two distinct groups, such that the lower list only contains elements that are less than or equal to the *pivot* value and the upper list only contains elements that are greater than or equal to the *pivot* value.
   $L = \{x \in A - \{v\} \mid x \le v\}$ and $U = \{x \in A - \{v\} \mid x \ge v\}$
4. Return the sequential join of *Quicksort*(*L*) and *Quicksort*(*U*)

Implementation – random-access container (low is pivot)

The pivot value will be partitioned into the top half of the range.

```
procedure QUICK-SORT( $A$, $low$, $high$ )
    if $low < high$
        then  $pivot \leftarrow$ PARTITION( $A$, $low$, $high$ )
              QUICK-SORT( $A$, $low$, $pivot$ )
              QUICK-SORT( $A$, $pivot + 1$, $high$ )
```

The following partition algorithm returns the high index of the lower partition.

```
function PARTITION( A, low, high ) returns integer
    pivotValue ← A[low]
    low ← low – 1
    high ← high + 1
    loop
        repeat
            low ← low + 1
        until A[low] ≥ pivotValue
        repeat
            high ← high – 1
        while A[high] ≤ pivotValue
        if low < high
            then   exchange A[low] ↔ A[high]
            else   return high
```

Implementation – random-access container (random pivot)

Choosing the first element as a pivot is as good as any other for a randomly scrambled list. However, it is terrible for a sorted or reverse-sorted list, causing each swapping pass to correctly place only one element – making the sort perform in O(n^2) time. A randomly chosen pivot often achieves the needed variation to avoid degenerate partitioning.

```
function PARTITION( A, low, high ) returns integer
    pivot ← random value in the range [ low, high ]
    pivotValue ← A[pivot]
    low ← low – 1
    …
```

Implementation – random-access container (middle pivot)

An alternative to the random pivot is the mid-point pivot.

```
function PARTITION( A, low, high ) returns integer
    pivot ← (low + high) div 2
    pivotValue ← A[pivot]
    low ← low – 1
    …
```

Implementation – random-access container (Hoare)

Hoare’s implementation is an optimization of the mid-point pivot that merges the partition function and the recursive function. Most significantly it does not move the pivot value to the beginning of the sub-list.

```
procedure QUICK-SORT( A, low, high )
    lo ← low
    hi ← high
    pivotValue ← A[ (low + high) div 2 ]
    while lo ≤ hi do
        while A[ lo ] < pivotValue do
            lo ← lo + 1
        while A[ hi ] > pivotValue do
            hi ← hi – 1
        if lo ≤ hi
            then   exchange A[ lo ] ↔ A[ hi ]
                   lo ← lo + 1
                   hi ← hi – 1
    if low < hi
        then QUICK-SORT( A, low, hi )
    if lo < high
        then QUICK-SORT( A, lo, high )
```

Implementation – medium of 3

On randomly ordered data, the midpoint pivot is no better than the low pivot, or any other single element as pivot. Medium of three improves the odds by selecting median of the first, last, and middle elements. The hope is that the probability of all three being skewed to one end or the other is significantly lower than the midpoint method.

```
procedure QUICK-SORT( A, low, high )
    lo ← low
    hi ← high
    first ← A[lo]
    last ← A[hi]
    pivotValue  ← A[(low + high) div 2]
    if pivotValue < first then    exchange A[pivotValue] ↔ A[first]
    if last < first then          exchange A[last] ↔ A[first]
    if last < pivotValue then     exchange A[last] ↔ A[pivotValue]
    while lo ≤ hi do
        while A[lo] < pivotValue do
            lo ← lo + 1
        while A[hi] > pivotValue do
            hi ← hi – 1
        if lo ≤ hi
            then   exchange A[lo] ↔ A[hi]
                   lo ← lo + 1
                   hi ← hi – 1
    if low < hi
        then QUICK-SORT( A, low, hi )
    if lo < high
        then QUICK-SORT( A, lo, high )
```

Implementation – hybrid sort

While quicksort is great on large lists, the overhead of recursions right down to one element and the setup complexity is often more costly than a simple sort such as insertion sort. This hybrid approach switches to insertion sort when the sub-list descends below a set threshold (typically 8-10).

```
procedure QUICK-SORT( A, low, high )
    if low + INSERT_SORT_THRESHOLD > high then
        INSERTION-SORT( A, low, high )
    else
        lo ← low
        hi ← high
        first ← A[lo]
        last ← A[hi]
        pivotValue  ← A[(low + high) div 2]
        if pivotValue < first then   exchange A[pivotValue] ↔ A[first]
        if last < first then         exchange A[last] ↔ A[first]
        if last < pivotValue then    exchange A[last] ↔ A[pivotValue]
        while lo ≤ hi do
            while A[lo] < pivotValue do
                lo ← lo + 1
            while A[hi] > pivotValue do
                hi ← hi – 1
            if lo ≤ hi
               then  exchange A[lo] ↔ A[hi]
                     lo ← lo + 1
                     hi ← hi – 1
        if low < hi
            then QUICK-SORT( A, low, hi )
        if lo < high
            then QUICK-SORT( A, lo, high )
```

Implementation – unchecked hybrid sort

Once the first block has been insert-sorted, all the other blocks *know* that they will always short circuit and therefore not need to check the index to see if it decremented past the beginning of the array. In this variation we don't sort the blocks until the very end (after all the quick recursions). Then we standard insert sort the first block, then we unchecked insert sort the remainder.

```
procedure QUICK-SORT( A, low, high )
    if low + INSERT_SORT_THRESHOLD ≤ high then
        lo ← low
        hi ← high
        first ← A[lo]
        last ← A[hi]
        pivotValue  ← A[(low + high) div 2]
        if pivotValue < first then   exchange A[pivotValue] ↔ A[first]
        if last < first then         exchange A[last] ↔ A[first]
        if last < pivotValue then    exchange A[last] ↔ A[pivotValue]
        while lo ≤ hi do
            while A[lo] < pivotValue do
                lo ← lo + 1
            while A[hi] > pivotValue do
                hi ← hi – 1
            if lo ≤ hi
               then  exchange A[lo] ↔ A[hi]
                     lo ← lo + 1
                     hi ← hi – 1
        if low < hi
            then QUICK-SORT( A, low, hi )
        if lo < high
            then QUICK-SORT( A, lo, high )
```

```
procedure UNCHECKED-INSERTION-SORT( $A$, $low$, $high$ )
    for $idxFirstUnsorted \leftarrow low$ to $high$ do
        $idxSink \leftarrow idxFirstUnsorted$
        while $A[idxSink] < A[idxSink - 1]$ do
            exchange $A[idxSink] \leftrightarrow A[idxSink - 1]$
            $idxSink \leftarrow idxSink - 1$

procedure QUICK-SORT( $A$ )
    if $empty\{A\}$ then return
    QUICK-SORT($A$, 0, $length\{A\} - 1$)
    $checkedSortLimit \leftarrow$ MIN($length\{A\}$, INSERT_SORT_THRESHOLD)
    INSERT-SORT($A$, 0, $checkedSortLimit - 1$)
    UNCHECKED-INSERT-SORT($A$, $checkedSortLimit$, $length\{A\} - 1$)
```

Reference

http://en.wikipedia.org/wiki/Quicksort

Selection Sort

Selection sort is an attempt to minimize the number of swaps performed during a sort by employing a search through the unsorted space to locate the exact element to be placed in the sorted portion of the list.

Algorithm

1. for each element E_i in the array;
2. we assume that the preceding elements $E_1, \ldots, E_{i-1}$ have already been sorted;
3. we search $E_i, \ldots, E_n$ for the element with the lowest value $\{ E_k \}$;
4. exchange elements E_i and E_k.

Notes:

- E represents keys.
- assume the array to be 1-based.

Implementation – random-access container (lowest)

First, I'll present a direct translation of the algorithm.

```
procedure SELECTION-SORT( A )
    for $idxFirstUnsorted \leftarrow 0$ to $length\{A\} - 2$ do
        $idxOfLowest \leftarrow idxFirstUnsorted$
        for $idx \leftarrow idxFirstUnsorted + 1$ to $length\{A\} - 1$ do
            if $A[idxOfLowest] > A[idx]$ then
                $idxOfLowest \leftarrow idx$
        if $idxOfLowest \neq idxFirstUnsorted$ then
            exchange $A[idxOfLowest] \leftrightarrow A[idxFirstUnsorted]$
```

Notes:

- For empty arrays the first line of the algorithm results in an index being created that has a negative value (i.e. –1). If your implementing language's index type is an unsigned type this statement would produce number underflow.

Implementation – random-access container (highest)

Often a more efficient implementation can be achieved by searching for the highest value and sorting from the high indices and working down. The following implements such an approach

```
procedure SELECTION-SORT($A$)
    for $idxLastUnsorted \leftarrow length\{A\} - 1$ downto 1 do
        $idxOfHighest \leftarrow idxLastUnsorted$
        for $idx \leftarrow idxLastUnsorted - 1$ downto 0 do
            if $A[idxOfHighest] < A[idx]$ then
                $idxOfHighest \leftarrow idx$
        if $idxOfHighest \neq idxLastUnsorted$ then
            exchange $A[idxOfHighest] \leftrightarrow A[idxLastUnsorted]$
```

Notes:

- ❖ For empty arrays the first line of the algorithm results in an index being created that has a negative value (i.e. –1). If your implementing language's index type is an unsigned type this statement would produce number underflow.

Implementation – forward iterators

The final implementation mimics the previous but replaces array references with pointers or iterators.

```
procedure SELECTION-SORT( $beg$, $end$ )
    while $beg \neq end$ do
        $lowest \leftarrow beg$
        $current \leftarrow beg$
        loop
            $current \leftarrow successor\{current\}$
            if $current = end$ then
                break
            if $value\{lowest\} > value\{current\}$ then
                $lowest \leftarrow current$
        if $lowest \neq beg$ then
            exchange $value\{lowest\} \leftrightarrow value\{beg\}$
        $beg \leftarrow successor\{beg\}$
```

Reference

http://en.wikipedia.org/wiki/Selection_sort

Shell's Sort

Insertion sort has good performance on sorted list, but poor performance on reversed lists. This fact inspired D. L. Shell to create this variation on insertion sort. Insertion sort's major problem is that when an element is found to be a great distance from its proper spot, it is only moved there one position at a time. This fault produces a costly O(n^2) behaviour.

Shell's sort eliminates this problem by segmenting the data set and performing the insertion on intermittent portions of the array.

Implementation – random-access container

```
procedure SHELL'S-SORT( $A$ )
    $stepSize \leftarrow length\{A\}$ div 2
    while $stepSize > 0$ do
        for $idxLastInSegment \leftarrow stepSize$ to $length\{A\} - 1$ do
            $idxCurrent \leftarrow idxLastInSegment$
            while $idxCurrent \geq stepSize$ and $A[idxCurrent] < A[idxCurrent - stepSize]$ do
                exchange $A[idxCurrent] \leftrightarrow A[[idxCurrent - stepSize]$
                $idxCurrent \leftarrow idxCurrent - stepSize$
        $stepSize \leftarrow stepSize$ div 2
```

Reference

http://en.wikipedia.org/wiki/Shell_sort

Appendix

Document History

Version	Date	Notes
n/a	1991-2018	Material developed and published in the *Gats Encyclopedia*
1.0.0	2019-02-01	Searching and sorting extracted from *Gats Encyclopedia* and published in this document.
1.1.0	2020-02-29	Quick-Sort section expanded to include: • Medium of three partitioning • Hybrid quick-insertion sort • Uncheck insertion sort optimization