Lecture Notes - Algorithms and Data Structures - Part 4: Searching and Sorting

Book · December 2013

DOI: 10.13140/2.1.5158.0486

Reiner Creutzburg
Brandenburg University of Applied Sciences

Algorithms and Data Structures

Part 4: Searching and Sorting (Wikipedia Book 2014)

By Wikipedians

Editors: Reiner Creutzburg, Jenny Knackmuß

PDF generated using the open source mwlib toolkit. See http://code.pediapress.com/ for more information.
PDF generated at: Sun, 22 Dec 2013 17:19:49 UTC

Contents

Searching

Search algorithm

In computer science, a **search algorithm** is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure, such as the roots of an equation with integer variables; or a combination of the two, such as the Hamiltonian circuits of a graph.

Classes of search algorithms

For virtual search spaces

Algorithms for searching virtual spaces are used in constraint satisfaction problem, where the goal is to find a set of value assignments to certain variables that will satisfy specific mathematical equations and inequations. They are also used when the goal is to find a variable assignment that will maximize or minimize a certain function of those variables. Algorithms for these problems include the basic brute-force search (also called "naïve" or "uninformed" search), and a variety of heuristics that try to exploit partial knowledge about structure of the space, such as linear relaxation, constraint generation, and constraint propagation.

An important subclass are the local search methods, that view the elements of the search space as the vertices of a graph, with edges defined by a set of heuristics applicable to the case; and scan the space by moving from item to item along the edges, for example according to the steepest descent or best-first criterion, or in a stochastic search. This category includes a great variety of general metaheuristic methods, such as simulated annealing, tabu search, A-teams, and genetic programming, that combine arbitrary heuristics in specific ways.

This class also includes various tree search algorithms, that view the elements as vertices of a tree, and traverse that tree in some special order. Examples of the latter include the exhaustive methods such as depth-first search and breadth-first search, as well as various heuristic-based search tree pruning methods such as backtracking and branch and bound. Unlike general metaheuristics, which at best work only in a probabilistic sense, many of these tree-search methods are guaranteed to find the exact or optimal solution, if given enough time.

Another important sub-class consists of algorithms for exploring the game tree of multiple-player games, such as chess or backgammon, whose nodes consist of all possible game situations that could result from the current situation. The goal in these problems is to find the move that provides the best chance of a win, taking into account all possible moves of the opponent(s). Similar problems occur when humans or machines have to make successive decisions whose outcomes are not entirely under one's control, such as in robot guidance or in marketing, financial or military strategy planning. This kind of problem - combinatorial search - has been extensively studied in the context of artificial intelligence. Examples of algorithms for this class are the minimax algorithm, alpha-beta pruning, and the A* algorithm.

For sub-structures of a given structure

The name combinatorial search is generally used for algorithms that look for a specific sub-structure of a given discrete structure, such as a graph, a string, a finite group, and so on. The term combinatorial optimization is typically used when the goal is to find a sub-structure with a maximum (or minimum) value of some parameter. (Since the sub-structure is usually represented in the computer by a set of integer variables with constraints, these problems can be viewed as special cases of constraint satisfaction or discrete optimization; but they are usually formulated and solved in a more abstract setting where the internal representation is not explicitly mentioned.)

An important and extensively studied subclass are the graph algorithms, in particular graph traversal algorithms, for finding specific sub-structures in a given graph — such as subgraphs, paths, circuits, and so on. Examples include Dijkstra's algorithm, Kruskal's algorithm, the nearest neighbour algorithm, and Prim's algorithm.

Another important subclass of this category are the string searching algorithms, that search for patterns within strings. Two famous examples are the Boyer–Moore and Knuth–Morris–Pratt algorithms, and several algorithms based on the suffix tree data structure.

Search for the maximum of a function

In 1953 J. Kiefer devised Fibonacci search which can be used to find the maximum of a unimodal function and has many other applications in computer science.

For quantum computers

There are also search methods designed for quantum computers, like Grover's algorithm, that are theoretically faster than linear or brute-force search even without the help of data structures or heuristics.

References

• Donald Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. ISBN 0-201-89685-0.

External links

• Uninformed Search Project [1] at the Wikiversity.
• Unsorted Data Searching Using Modulated Database [2].

References

[1] http://en.wikiversity.org/wiki/Uninformed_Search_Project
[2] http://sites.google.com/site/hantarto/quantum-computing/unsorted

Linear search

In computer science, **linear search** or **sequential search** is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list; and so is its expected cost, if all list elements are equally likely to be searched for. Therefore, if the list has more than a few elements, other methods (such as binary search or hashing) will be faster, but they also impose additional requirements.

Analysis

For a list with *n* items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case *n* comparisons are needed.

If the value being sought occurs *k* times in the list, and all orderings of the list are equally likely, the expected number of comparisons is

$$\begin{cases} n & \text{if } k = 0 \\ \dfrac{n+1}{k+1} & \text{if } 1 \le k \le n. \end{cases}$$

For example, if the value being sought occurs once in the list, and all orderings of the list are equally likely, the expected number of comparisons is $\frac{n+1}{2}$. However, if it is *known* that it occurs once, then at most *n* - 1 comparisons are needed, and the expected number of comparisons is

$$\frac{(n+2)(n-1)}{2n}$$

(for example, for *n* = 2 this is 1, corresponding to a single if-then-else construct).

Either way, asymptotically the worst-case cost and the expected cost of linear search are both O(*n*).

Non-uniform probabilities

The performance of linear search improves if the desired value is more likely to be near the beginning of the list than to its end. Therefore, if some values are much more likely to be searched than others, it is desirable to place them at the beginning of the list.

In particular, when the list items are arranged in order of decreasing probability, and these probabilities are geometrically distributed, the cost of linear search is only O(1). If the table size *n* is large enough, linear search will be faster than binary search, whose cost is O(log *n*).

Application

Linear search is usually very simple to implement, and is practical when the list has only a few elements, or when performing a single search in an unordered list.

When many values have to be searched in the same list, it often pays to pre-process the list in order to use a faster method. For example, one may sort the list and use binary search, or build any efficient search data structure from it. Should the content of the list change frequently, repeated re-organization may be more trouble than it is worth.

As a result, even though in theory other search algorithms may be faster than linear search (for instance binary search), in practice even on medium sized arrays (around 100 items or less) it might be infeasible to use anything else. On larger arrays, it only makes sense to use other, faster search methods if the data is large enough, because the initial time to prepare (sort) the data is comparable to many linear searches

Pseudocode

Forward iteration

This pseudocode describes a typical variant of linear search, where the result of the search is supposed to be either the location of the list item where the desired value was found; or an invalid location *Λ*, to indicate that the desired element does not occur in the list.

```
For each item in the list:
    if that item has the desired value,
        stop the search and return the item's location.
Return Λ.
```

In this pseudocode, the last line is executed only after all list items have been examined with none matching.

If the list is stored as an array data structure, the location may be the index of the item found (usually between 1 and *n*, or 0 and *n*−1). In that case the invalid location *Λ* can be any index before the first element (such as 0 or −1, respectively) or after the last one (*n*+1 or *n*, respectively).

If the list is a simply linked list, then the item's location is its reference, and *Λ* is usually the null pointer.

Recursive version

Linear search can also be described as a recursive algorithm:

```
LinearSearch(value, list)
  if the list is empty, return Λ;
  else
    if the first item of the list has the desired value, return its location;
    else return LinearSearch(value, remainder of the list)
```

Searching in reverse order

Linear search in an array is usually programmed by stepping up an index variable until it reaches the last index. This normally requires two comparison instructions for each list item: one to check whether the index has reached the end of the array, and another one to check whether the item has the desired value. In many computers, one can reduce the work of the first comparison by scanning the items in reverse order.

Suppose an array *A* with elements indexed 1 to *n* is to be searched for a value *x*. The following pseudocode performs a forward search, returning *n + 1* if the value is not found:

```
Set $i$ to 1.
Repeat this loop:
    If $i > n$, then exit the loop.
    If $A[i] = x$, then exit the loop.
    Set $i$ to $i + 1$.
Return $i$.
```

The following pseudocode searches the array in the reverse order, returning 0 when the element is not found:

```
Set $i$ to $n$.
Repeat this loop:
    If $i \leq 0$, then exit the loop.
    If $A[i] = x$, then exit the loop.
    Set $i$ to $i - 1$.
Return $i$.
```

Most computers have a conditional branch instruction that tests the sign of a value in a register, or the sign of the result of the most recent arithmetic operation. One can use that instruction, which is usually faster than a comparison against some arbitrary value (requiring a subtraction), to implement the command "If $i \leq 0$, then exit the loop".

This optimization is easily implemented when programming in machine or assembly language. That branch instruction is not directly accessible in typical high-level programming languages, although many compilers will be able to perform that optimization on their own.

Using a sentinel

Another way to reduce the overhead is to eliminate all checking of the loop index. This can be done by inserting the desired item itself as a sentinel value at the far end of the list, as in this pseudocode:

```
Set $A[n + 1]$ to $x$.
Set $i$ to 1.
Repeat this loop:
    If $A[i] = x$, then exit the loop.
    Set $i$ to $i + 1$.
Return $i$.
```

With this stratagem, it is not necessary to check the value of i against the list length n: even if x was not in A to begin with, the loop will terminate when $i = n + 1$. However this method is possible only if the array slot $A[n + 1]$ exists but is not being otherwise used. Similar arrangements could be made if the array were to be searched in reverse order, and element $A(0)$ were available.

Although the effort avoided by these ploys is tiny, it is still a significant component of the overhead of performing each step of the search, which is small. Only if many elements are likely to be compared will it be worthwhile considering methods that make fewer comparisons but impose other requirements.

Linear search on an ordered list

For ordered lists that must be accessed sequentially, such as linked lists or files with variable-length records lacking an index, the average performance can be improved by giving up at the first element which is greater than the unmatched target value, rather than examining the entire list.

If the list is stored as an ordered array, then binary search is almost always more efficient than linear search as with $n > 8$, say, unless there is some reason to suppose that most searches will be for the small elements near the start of the sorted list.

References

Binary search algorithm

Binary search algorithm

Class	Search algorithm
Data structure	Array
Worst case performance	$O(\log n)$
Best case performance	$O(1)$
Average case performance	$O(\log n)$
Worst case space complexity	$O(1)$

In computer science, a **binary search** or **half-interval search** algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic divide and conquer search algorithm.

Overview

Searching a sorted collection is a common task. A dictionary is a sorted list of word definitions. Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers. Knowing someone's name allows one to quickly find their telephone number and address.

If the list to be searched contains more than a few items (a dozen, say) a binary search will require far fewer comparisons than a linear search, but it imposes the requirement that the list be sorted. Similarly, a hash search can be faster than a binary search but imposes still greater requirements. If the contents of the array are modified between searches, maintaining these requirements may even take more time than the searches. And if it is known that some items will be searched for *much* more often than others, *and* it can be arranged that these items are at the start of the list, then a linear search may be the best.

More generally, algorithm allows searching over argument of any monotonic function for a point, at which function reaches the arbitrary value (enclosed between minimum and maximum at the given range).

Examples

Example: L = 1 3 4 6 8 9 11. X = 4.

```
   Compare X to 6. It's smaller. Repeat with L = 1 3 4.
   Compare X to 3. It's bigger. Repeat with L = 4.
   Compare X to 4. It's equal. We're done, we found X.
```

This is called Binary Search: each iteration of (1)-(4) the length of the list we are looking in gets cut in half. Therefore, the total number of iterations cannot be greater than logN.

Number guessing game

This rather simple game begins something like "I'm thinking of an integer between forty and sixty inclusive, and to your guesses I'll respond 'Higher', 'Lower', or 'Yes!' as might be the case." Supposing that N is the number of possible values (here, twenty-one as "inclusive" was stated), then at most $\lfloor \log_2 N \rfloor$ questions are required to determine the number, since each question halves the search space. Note that one less question (iteration) is required than for the general algorithm, since the number is already constrained to be within a particular range.

Even if the number to guess can be arbitrarily large, in which case there is no upper bound N, the number can be found in at most $2\lfloor \log_2 k \rfloor + 1$ steps (where k is the (unknown) selected number) by first finding an upper bound by repeated doubling.[*citation needed*] For example, if the number were 11, the following sequence of guesses could be used to find it: 1 (Higher), 2 (Higher), 4 (Higher), 8 (Higher), 16 (Lower), 12 (Lower), 10 (Higher). Now we know that the number must be 11 because it is higher than 10 and lower than 12.

One could also extend the method to include negative numbers; for example the following guesses could be used to find −13: 0, −1, −2, −4, −8, −16, −12, −14. Now we know that the number must be −13 because it is lower than −12 and higher than −14.

Word lists

People typically use a mixture of the binary search and interpolative search algorithms when searching a telephone book, after the initial guess we exploit the fact that the entries are sorted and can rapidly find the required entry. For example when searching for Smith, if Rogers and Thomas have been found, one can flip to a page about halfway between the previous guesses. If this shows Samson, it can be concluded that Smith is somewhere between the Samson and Thomas pages so these can be divided.

Applications to complexity theory

Even if we do not know a fixed range the number k falls in, we can still determine its value by asking $2\lceil \log_2 k \rceil$ simple yes/no questions of the form "Is k greater than x?" for some number x. As a simple consequence of this, if you can answer the question "Is this integer property k greater than a given value?" in some amount of time then you can find the value of that property in the same amount of time with an added factor of $\log_2 k$. This is called a *reduction*, and it is because of this kind of reduction that most complexity theorists concentrate on decision problems, algorithms that produce a simple yes/no answer.

For example, suppose we could answer "Does this n x n matrix have permanent larger than k?" in O(n^2) time. Then, by using binary search, we could find the (ceiling of the) permanent itself in O(n^2 log p) time, where p is the value of the permanent. Notice that p is not the size of the input, but the *value* of the output; given a matrix whose maximum item (in absolute value) is m, p is bounded by $m^n n!$. Hence log p = O(n log n + log m). A binary search could find the permanent in O(n^3 log n + n^2 log m).

Algorithm

Recursive

A straightforward implementation of binary search is recursive. The initial call uses the indices of the entire array to be searched. The procedure then calculates an index midway between the two indices, determines which of the two subarrays to search, and then does a recursive call to search that subarray. Each of the calls is tail recursive, so a compiler need not make a new stack frame for each call. The variables `imin` and `imax` are the lowest and highest inclusive indices that are searched.

```
int binary_search(int A[], int key, int imin, int imax)
{
  // test if array is empty
  if (imax < imin)
    // set is empty, so return value showing not found
    return KEY_NOT_FOUND;
  else
    {
      // calculate midpoint to cut set in half
      int imid = midpoint(imin, imax);

      // three-way comparison
      if (A[imid] > key)
        // key is in lower subset
        return binary_search(A, key, imin, imid-1);
      else if (A[imid] < key)
        // key is in upper subset
        return binary_search(A, key, imid+1, imax);
      else
        // key has been found
        return imid;
    }
}
```

It is invoked with initial `imin` and `imax` values of `0` and `N-1` for a zero based array of length N.

The number type "int" shown in the code has an influence on how the midpoint calculation can be implemented correctly. With unlimited numbers, the midpoint can be calculated as "`(imin + imax) / 2`". In practical programming, however, the calculation is often performed with numbers of a limited range, and then the intermediate result "`(imin + imax)`" might overflow. With limited numbers, the midpoint can be calculated correctly as "`imin + ((imax - imin) / 2)`".

Iterative

The binary search algorithm can also be expressed iteratively with two index limits that progressively narrow the search range.

```
int binary_search(int A[], int key, int imin, int imax)
{
  // continue searching while [imin,imax] is not empty
  while (imax >= imin)
    {
      // calculate the midpoint for roughly equal partition
      int imid = midpoint(imin, imax);
      if(A[imid] == key)
        // key found at index imid
        return imid;
      // determine which subarray to search
      else if (A[imid] < key)
        // change min index to search upper subarray
        imin = imid + 1;
      else
```

```
        // change max index to search lower subarray
        imax = imid - 1;
    }
  // key was not found
  return KEY_NOT_FOUND;
}
```

Deferred detection of equality

The above iterative and recursive versions take three paths based on the key comparison: one path for less than, one path for greater than, and one path for equality. (There are two conditional branches.) The path for equality is taken only when the record is finally matched, so it is rarely taken. That branch path can be moved outside the search loop in the deferred test for equality version of the algorithm. The following algorithm uses only one conditional branch per iteration.

```
// inclusive indices
//   0 <= imin when using truncate toward zero divide
//     imid = (imin+imax)/2;
//   imin unrestricted when using truncate toward minus infinity divide
//     imid = (imin+imax)>>1; or
//     imid = (int)floor((imin+imax)/2.0);
int binary_search(int A[], int key, int imin, int imax)
{
  // continually narrow search until just one element remains
  while (imin < imax)
    {
      int imid = midpoint(imin, imax);

      // code must guarantee the interval is reduced at each iteration
      assert(imid < imax);
      // note: 0 <= imin < imax implies imid will always be less than imax

      // reduce the search
      if (A[imid] < key)
        imin = imid + 1;
      else
        imax = imid;
    }
  // At exit of while:
  //   if A[] is empty, then imax < imin
  //   otherwise imax == imin

  // deferred test for equality
  if ((imax == imin) && (A[imin] == key))
    return imin;
  else
    return KEY_NOT_FOUND;
}
```

The deferred detection approach foregoes the possibility of early termination on discovery of a match, so the search will take about $\log_2(N)$ iterations. On average, a *successful* early termination search will not save many iterations. For large arrays that are a power of 2, the savings is about two iterations. Half the time, a match is found with one iteration left to go; one quarter the time with two iterations left, one eighth with three iterations, and so forth. The infinite series sum is 2.

The deferred detection algorithm has the advantage that if the keys are not unique, it returns the smallest index (the starting index) of the region where elements have the search key. The early termination version would return the first match it found, and that match might be anywhere in region of equal keys.

Performance

With each test that fails to find a match at the probed position, the search is continued with one or other of the two sub-intervals, each at most half the size. More precisely, if the number of items, N, is odd then both sub-intervals will contain $(N-1)/2$ elements, while if N is even then the two sub-intervals contain $N/2-1$ and $N/2$ elements.

If the original number of items is N then after the first iteration there will be at most $N/2$ items remaining, then at most $N/4$ items, at most $N/8$ items, and so on. In the worst case, when the value is not in the list, the algorithm must continue iterating until the span has been made empty; this will have taken at most $\lfloor\log_2(N)+1\rfloor$ iterations, where the $\lfloor\ \rfloor$ notation denotes the floor function that rounds its argument down to an integer. This worst case analysis is tight: for any N there exists a query that takes exactly $\lfloor\log_2(N)+1\rfloor$ iterations. When compared to linear search, whose worst-case behaviour is N iterations, we see that binary search is substantially faster as N grows large. For example, to search a list of one million items takes as many as one million iterations with linear search, but never more than twenty iterations with binary search. However, a binary search can only be performed if the list is in sorted order.

Average performance

$\log_2(N)-1$ is the expected number of probes in an average successful search, and the worst case is $\log_2(N)$, just one more probe.[*citation needed*] If the list is empty, no probes at all are made. Thus binary search is a logarithmic algorithm and executes in $O(\log N)$ time. In most cases it is considerably faster than a linear search. It can be implemented using iteration, or recursion. In some languages it is more elegantly expressed recursively; however, in some C-based languages tail recursion is not eliminated and the recursive version requires more stack space.

Binary search can interact poorly with the memory hierarchy (i.e. caching), because of its random-access nature. For in-memory searching, if the span to be searched is small, a linear search may have superior performance simply because it exhibits better locality of reference. For external searching, care must be taken or each of the first several probes will lead to a disk seek. A common method is to abandon binary searching for linear searching as soon as the size of the remaining span falls below a small value such as 8 or 16 or even more in recent computers. The exact value depends entirely on the machine running the algorithm.

Notice that for multiple searches *with a fixed value for N*, then (with the appropriate regard for integer division), the first iteration always selects the middle element at $N/2$, and the second always selects either $N/4$ or $3N/4$, and so on. Thus if the array's key values are in some sort of slow storage (on a disc file, in virtual memory, not in the cpu's on-chip memory), keeping those three keys in a local array for a special preliminary search will avoid accessing widely separated memory. Escalating to seven or fifteen such values will allow further levels at not much cost in storage. On the other hand, if the searches are frequent and not separated by much other activity, the computer's various storage control features will more or less automatically promote frequently accessed elements into faster storage.

When multiple binary searches are to be performed for the same key in related lists, fractional cascading can be used to speed up successive searches after the first one.

Even though in theory binary search is almost always faster than linear search, in practice even on medium sized arrays (around 100 items or less) it might be infeasible to ever use binary search. On larger arrays, it only makes sense to binary search if the number of searches is large enough, because the initial time to sort the array is comparable to many linear searches

Variations

There are many, and they are easily confused.

Exclusive or inclusive bounds

The most significant differences are between the "exclusive" and "inclusive" forms of the bounds. In the "exclusive" bound form the span to be searched is *(L+1)* to *(R−1)*, and this may seem clumsy when the span to be searched could be described in the "inclusive" form, as *L* to *R*. Although the details differ the two forms are equivalent as can be seen by transforming one version into the other. The inclusive bound form can be attained by replacing all appearances of "L" by "(L−1)" and "R" by "(R+1)" then rearranging. Thus, the initialisation of *L := 0* becomes *(L−1) := 0* or *L := 1*, and *R := N+1* becomes *(R+1) := N+1* or *R := N*. So far so good, but note now that the changes to *L* and *R* are no longer simply transferring the value of *p* to *L* or *R* as appropriate but now must be *(R+1) := p* or *R := p−1*, and *(L−1) := p* or *L := p+1*.

Thus, the gain of a simpler initialisation, done once, is lost by a more complex calculation, and which is done for every iteration. If that is not enough, the test for an empty span is more complex also, as compared to the simplicity of checking that the value of *p* is zero. Nevertheless, the inclusive bound form is found in many publications, such as Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition.

Another common variation uses inclusive bounds for the left bound, but exclusive bounds for the right bound. This is derived from the fact that the bounds in a language with zero-based arrays can be simply initialized to 0 and the size of the array, respectively. This mirrors the way array slices are represented in some programming languages.

Midpoint and width

A different variation involves abandoning the *L* and *R* pointers and using a current position *p* and a width *w*. At each iteration, the position *p* is adjusted and the width *w* is halved. Knuth states, "It is possible to do this, but only if extreme care is paid to the details."

Search domain

There is no particular requirement that the array being searched has the bounds 1 to *N*. It is possible to search a specified range, elements *first* to *last* instead of 1 to *N*. All that is necessary is that the initialization of the bounds be *L := first−1* and *R := last+1*, then all proceeds as before.

The elements of the list are not necessarily all unique. If one searches for a value that occurs multiple times in the list, the index returned will be of the first-encountered equal element, and this will not necessarily be that of the first, last, or middle element of the run of equal-key elements but will depend on the positions of the values. Modifying the list even in seemingly unrelated ways such as adding elements elsewhere in the list may change the result.

If the location of the first and/or last equal element needs to be determined, this can be done efficiently with a variant of the binary search algorithms which perform only one inequality test per iteration. See deferred detection of equality.

Noisy search

Several algorithms closely related to or extending binary search exist. For instance, **noisy binary search** solves the same class of projects as regular binary search, with the added complexity that any given test can return a false value at random. (Usually, the number of such erroneous results are bounded in some way, either in the form of an average error rate, or in the total number of errors allowed per element in the search space.) Optimal algorithms for several classes of noisy binary search problems have been known since the late seventies, and more recently, optimal algorithms for noisy binary search in quantum computers (where several elements can be tested at the same time) have been discovered.

Implementation issues

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky… — Donald Knuth

When Jon Bentley assigned it as a problem in a course for professional programmers, he found that an astounding ninety percent failed to code a binary search correctly after several hours of working on it, and another study shows that accurate code for it is only found in five out of twenty textbooks.[1] Furthermore, Bentley's own implementation of binary search, published in his 1986 book *Programming Pearls*, contains an error that remained undetected for over twenty years.

Arithmetic

In a practical implementation, the variables used to represent the indices will often be of finite size, hence only capable of representing a finite range of values. For example, 32-bit unsigned integers can only hold values from 0 to 4294967295. 32-bit signed integers can only hold values from -2147483648 to 2147483647. If the binary search algorithm is to operate on large arrays, this has two implications:

- The values `first - 1` and `last + 1` must both be representable within the finite bounds of the chosen integer type . Therefore, continuing the 32-bit unsigned example, the largest value that `last` may take is +429496729**4**, not +429496729**5**. A problem exists even for the "inclusive" form of the method, as if `x > A(4294967295).Key`, then on the final iteration the algorithm will attempt to store 4294967296 into `L` and fail. Equivalent issues apply to the lower limit, where `first - 1` could become negative as when the first element of the array is at index zero.
- If the midpoint of the span is calculated as `p := (L + R)/2`, then the value `(L + R)` will exceed the number range if `last` is greater than (for unsigned) 4294967295/2 or (for signed) 2147483647/2 and the search wanders toward the upper end of the search space. This can be avoided by performing the calculation as `p := (R - L)/2 + L`. For example, this bug existed in Java SDK at `Arrays.binarySearch()` from 1.2 to 5.0 and fixed in 6.0.

Language support

Many standard libraries provide a way to do a binary search:

- C provides algorithm function `bsearch` in its standard library.
- C++'s STL provides algorithm functions `binary_search`, `lower_bound` and `upper_bound`.
- Java offers a set of overloaded `binarySearch()` static methods in the classes `Arrays` [2] and `Collections` [3] in the standard `java.util` package for performing binary searches on Java arrays and on `Lists`, respectively. They must be arrays of primitives, or the arrays or Lists must be of a type that implements the `Comparable` interface, or you must specify a custom `Comparator` object.
- Microsoft's .NET Framework 2.0 offers static generic versions of the binary search algorithm in its collection base classes. An example would be `System.Array`'s method `BinarySearch<T>(T[] array, T value)`.

- Python provides the `bisect` [4] module.
- COBOL can perform binary search on internal tables using the `SEARCH ALL` statement.
- Perl can perform a generic binary search using the CPAN module Search::Binary.
- Go's `sort` standard library package contains functions `Search`, `SearchInts`, `SearchFloat64s`, and `SearchStrings`, which implement general binary search, as well as specific implementations for searching slices of integers, floating-point numbers, and strings, respectively.
- For Objective-C, the Cocoa framework provides the NSArray -indexOfObject:inSortedRange:options:usingComparator: [5] method in Mac OS X 10.6+. Apple's Core Foundation C framework also contains a CFArrayBSearchValues() [6] function.

References

[1] cited at

[2] http://download.oracle.com/javase/7/docs/api/java/util/Arrays.html

[3] http://download.oracle.com/javase/7/docs/api/java/util/Collections.html

[4] http://docs.python.org/library/bisect.html

[5] http://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSArray_Class/NSArray.html#//apple_ref/occ/instm/NSArray/indexOfObject:inSortedRange:options:usingComparator:

[6] http://developer.apple.com/library/mac/documentation/CoreFoundation/Reference/CFArrayRef/Reference/reference.html#//apple_ref/c/func/CFArrayBSearchValues

Other sources

- Kruse, Robert L.: "Data Structures and Program Design in C++", Prentice-Hall, 1999, ISBN 0-13-768995-0, page 280.
- van Gasteren, Netty; Feijen, Wim (1995). "The Binary Search Revisited" (http://www.mathmeth.com/wf/files/wf2xx/wf214.pdf) (PDF). *AvG127/WF214*. (investigates the foundations of the binary search, debunking the myth that it applies only to sorted arrays)

External links

- NIST Dictionary of Algorithms and Data Structures: binary search (http://www.nist.gov/dads/HTML/binarySearch.html)
- Binary search implemented in 12 languages (http://www.codecodex.com/wiki/Binary_search)
- Binary search casual examples - dictionary, array and monotonic function (http://codeabbey.com/index/wiki/binary-search)

Sorting

Sorting algorithm

A **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation (reordering) of the input.

Further, the data is often taken to be in an array, which allows random access, rather than a list, which only allows sequential access, though often algorithms can be applied with suitable modification to either type of data.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956.[1] A fundamental limit of comparison sorting algorithms is that they require linearithmic time – O(n log n) – in the worst case, though better performance is possible on real-world data (such as almost-sorted data), and algorithms not based on comparison, such as counting sort, can have better performance. Although many consider sorting a solved problem – asymptotically optimal algorithms have been known since the mid-20th century – useful new algorithms are still being invented, with the now widely used Timsort dating to 2002, and the library sort being first published in 2006.

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and upper and lower bounds.

Classification

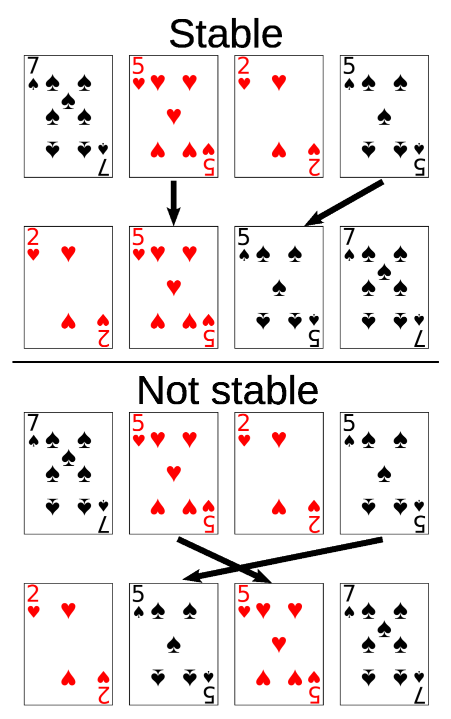
Sorting algorithms are often classified by:

- Computational complexity (worst, average and best behavior) of element comparisons in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O(n log n), with parallel sort in O($\log^2 n$), and bad behavior is O(n^2). (See Big O notation.) Ideal behavior for a serial sort is O(n), but this is not possible in the average case, optimal parallel sorting is O(log n). Comparison-based sorting algorithms, which evaluate the elements of the list via an abstract key comparison operation, need at least O(n log n) comparisons for most inputs.
- Computational complexity of swaps (for "in place" algorithms).
- Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place". Strictly, an in place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(n)) additional memory is considered "in place".
- Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
- Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.

- General method: insertion, exchange, selection, merging, *etc*. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort. Also whether the algorithm is serial or parallel. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.
- Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

Stability

When sorting some kinds of data, only part of the data is examined when determining the sort order. For example, in the card sorting example to the right, the cards are being sorted by their rank, and their suit is being ignored. The result is that it's possible to have multiple different correctly sorted versions of the original list. Stable sorting algorithms choose one of these, according to the following rule: if two items compare as equal, like the two 5 cards, then their relative order will be preserved, so that if one came before the other in the input, it will also come before the other in the output.



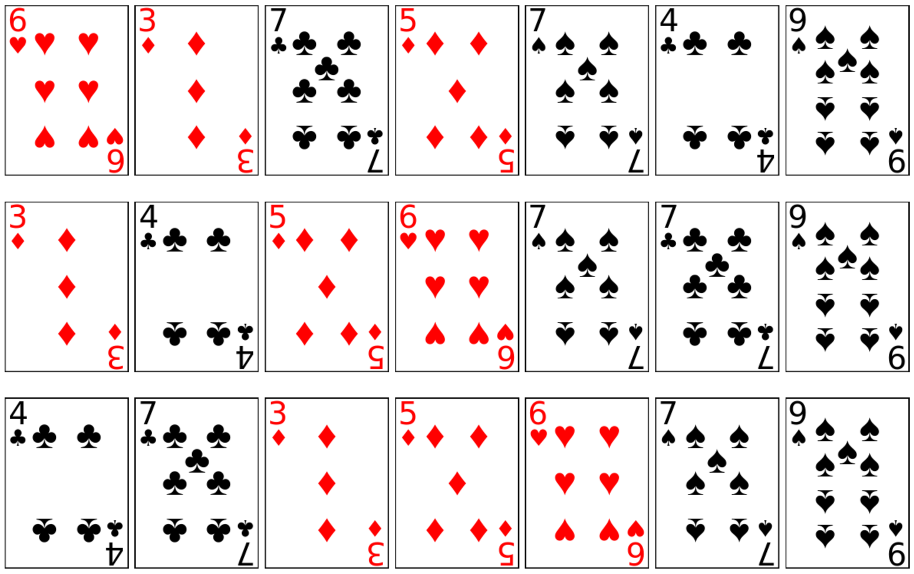
An example of stable sorting on playing cards. When the cards are sorted by rank with a stable sort, the two 5s must remain in the same order in the sorted output that they were originally in. When they are sorted with a non-stable sort, the 5s may end up in the opposite order in the sorted output.

More formally, the data being sorted can be represented as a record or tuple of values, and the part of the data that is used for sorting is called the *key*. In the card example, cards are represented as a record (rank, suit), and the key is the rank. A sorting algorithm is stable if whenever there are two records R and S with the same key, and R appears before S in the original list, then R will always appear before S in the sorted list.

When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.

Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original input list as a tie-breaker. Remembering this order, however, may require additional time and space.

One application for stable sorting algorithms is sorting a list using a primary and secondary key. For example, suppose we wish to sort a hand of cards such that the suits are in the order clubs (♣), diamonds (♦), hearts (♥), spades (♠), and within each suit, the cards are sorted by rank. This can be done by first sorting the cards by rank (using any sort), and then doing a stable sort by suit:



Within each suit, the stable sort preserves the ordering by rank that was already done. This idea can be extended to any number of keys, and is leveraged by radix sort. The same effect can be achieved with an unstable sort by using a lexicographic key comparison, which e.g. compares first by suits, and then compares by rank if the suits are the same.

Comparison of algorithms

In this table, n is the number of records to be sorted. The columns "Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. These are all comparison sorts, and so cannot perform better than O(n log n) in the average or worst case. The run time and the memory of algorithms could be measured using various notations like theta, omega, Big-O, small-o, etc. The memory and the run times below are applicable for all the 5 notations.

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	typical in-place sort is not stable; stable versions exist	Partitioning	Quicksort is usually done in place with O(log(n)) stack space.[citation needed] Most implementations are unstable, as stable in-place partitioning is more complex. Naïve variants use an O(n) space array to store the partition.[citation needed]
Merge sort	$n \log n$	$n \log n$	$n \log n$	DependsWikipedia:Please clarify; worst case is n	Yes	Merging	Highly parallelizable (up to O(log(n)) using the Three Hungarian's Algorithm or more practically, Cole's parallel merge sort) for processing large amounts of data.
In-place merge sort	—	—	$n\left(\log n\right)^2$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[2]

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.[3]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	n comparisons when the data is already sorted or reverse sorted.
Shell sort	n	$n(\log n)^2$ or $n^{3/2}$	Depends on gap sequence; best known is $n(\log n)^2$	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	$n \log n$	$n \log n$	n	Yes	Insertion	When using a self-balancing binary search tree
Cycle sort	—	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes
Library sort	—	$n \log n$	n^2	n	Yes	Insertion	
Patience sorting	—	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences within O(*n* log *n*)
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive sort - n comparisons when the data is already sorted, and 0 swaps.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	—	$n \log n$	$n \log n$	n[4]		Selection	
Cocktail sort	n	n^2	n^2	1	Yes	Exchanging	
Comb sort	n	$n \log n$	n^2	1	No	Exchanging	Small code size
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size
Franceschini's method	–	$n \log n$	$n \log n$	1	No		

The following table describes integer sorting algorithms and other sorting algorithms that are not comparison sorts. As such, they are not limited by a $\Omega\left(n \log n\right)$ lower bound. Complexities below are in terms of *n*, the number of items to be sorted, *k*, the size of each key, and *d*, the digit size used by the implementation. Many of them are based on the assumption that the key size is large enough that all entries have unique key values, and hence that *n* << 2^k, where << means "much less than."

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.
Bucket sort (integer keys)	—	$n+r$	$n+r$	$n+r$	Yes	Yes	r is the range of numbers to be sorted. If r = $\mathcal{O}\left(n\right)$ then Avg RT = $\mathcal{O}\left(n\right)$
Counting sort	—	$n+r$	$n+r$	$n+r$	Yes	Yes	r is the range of numbers to be sorted. If r = $\mathcal{O}\left(n\right)$ then Avg RT = $\mathcal{O}\left(n\right)$
LSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	n	Yes	No	
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$n+\frac{k}{d} \cdot 2^d$	Yes	No	Stable version uses an external array of size n to hold all of the bins
MSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	$\frac{k}{d} \cdot 2^d$	No	No	In-Place. 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.

The following table describes some sorting algorithms that are impractical for real-life use due to extremely poor performance or a requirement for specialized hardware.

Name	Best	Average	Worst	Memory	Stable	Comparison	Other notes
Bead sort	—	N/A	N/A	—	N/A	No	Requires specialized hardware
Simple pancake sort	—	n	n	$\log n$	No	Yes	Count is number of flips.
Spaghetti (Poll) sort	n	n	n	n^2	Yes	Polling	This A linear-time, analog algorithm for sorting a sequence of items, requiring O(*n*) stack space, and the sort is stable. This requires n parallel processors. Spaghetti sort#Analysis
Sorting networks	—	$\log n$	$\log n$	$n \cdot \log(n)$	Yes	No	Requires a custom circuit of size $\mathcal{O}\left(n \cdot \log(n)\right)$

Additionally, theoretical computer scientists have detailed other sorting algorithms that provide better than $\mathcal{O}\left(n \log n\right)$ time complexity with additional constraints, including:

- Han's algorithm, a deterministic algorithm for sorting keys from a domain of finite size, taking $\mathcal{O}\left(n \log \log n\right)$ time and $\mathcal{O}\left(n\right)$ space.[5]
- Thorup's algorithm, a randomized algorithm for sorting keys from a domain of finite size, taking $\mathcal{O}\left(n \log \log n\right)$ time and $\mathcal{O}\left(n\right)$ space.[6]
- A randomized integer sorting algorithm taking $\mathcal{O}\left(n\sqrt{\log \log n}\right)$ expected time and $\mathcal{O}\left(n\right)$ space.[7]

Algorithms not yet compared above include:

- Odd-even sort
- Flashsort
- Burstsort
- Postman sort
- Stooge sort
- Samplesort

- Bitonic sorter

Popular sorting algorithms

While there are a large number of sorting algorithms, in practical implementations a few algorithms predominate. Insertion sort is widely used for small data sets, while for large data sets an asymptotically efficient sort is used, primarily heap sort, merge sort, or quicksort. Efficient implementations generally use a hybrid algorithm, combining an asymptotically efficient algorithm for the overall sort with insertion sort for small lists at the bottom of a recursion. Highly tuned implementations use more sophisticated variants, such as Timsort (merge sort, insertion sort, and additional logic), used in Android, Java, and Python, and introsort (quicksort and heap sort), used (in variant forms) in some C++ sort implementations and in .Net.

For more restricted data, such as numbers in a fixed interval, distribution sorts such as counting sort or radix sort are widely used. Bubble sort and variants are rarely used in practice, but are commonly found in teaching and theoretical discussions.

When physically sorting objects, such as alphabetizing papers (such as tests or books), people intuitively generally use insertion sorts for small sets. For larger sets, people often first bucket, such as by initial letter, and multiple bucketing allows practical sorting of very large sets. Often space is relatively cheap, such as by spreading objects out on the floor or over a large area, but operations are expensive, particularly moving an object a large distance – locality of reference is important. Merge sorts are also practical for physical objects, particularly as two hands can be used, one for each list to merge, while other algorithms, such as heap sort or quick sort, are poorly suited for human use. Other algorithms, such as library sort, a variant of insertion sort that leaves spaces, are also practical for physical use.

Simple sorts

Two of the simplest sorts are insertion sort and selection sort, both of which are efficient on small data, due to low overhead, but not efficient on large data. Insertion sort is generally faster than selection sort in practice, due to fewer comparisons and good performance on almost-sorted data, and thus is preferred in practice, but selection sort uses fewer writes, and thus is used when write performance is a limiting factor.

Insertion sort

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort (see below) is a variant of insertion sort that is more efficient for larger lists.

Selection sort

Selection sort is an in-place comparison sort. It has $O(n^2)$ complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

Efficient sorts

Practical general sorting algorithms are almost always based on an algorithm with average complexity (and generally worst-case complexity) O(n log n), of which the most common are heap sort, merge sort, and quicksort. Each has advantages and drawbacks, with the most significant being that simple implementation of merge sort uses O(n) additional space, and simple implementation of quicksort has O(n^2) worst-case complexity. These problems can be solved or ameliorated at the cost of a more complex algorithm.

While these algorithms are asymptotically efficient on random data, for practical efficiency on real-world data various modifications are used. First, the overhead of these algorithms becomes significant on smaller data, so often a hybrid algorithm is used, commonly switching to insertion sort once the data is small enough. Second, the algorithms often perform poorly on already sorted data or almost sorted data – these are common in real-world data, and can be sorted in O(n) time by appropriate algorithms. Finally, they may also be unstable, and stability is often a desirable property in a sort. Thus more sophisticated algorithms are often employed, such as Timsort (based on merge sort) or introsort (based on quicksort, falling back to heap sort).

Merge sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(n log n). It is also easily applied to lists, not only arrays, as it only requires sequential access, not random access. However, it has additional O(n) space complexity, and involves a large number of copies in simple implementations.

Merge sort has seen a relatively recent surge in popularity for practical implementations, due to its use in the sophisticated algorithm Timsort, which is used for the standard sort routine in the programming languages Python[8] and Java (as of JDK7[9]). Merge sort itself is the standard routine in Perl,[10] among others, and has been used in Java at least since 2000 in JDK1.3.[11][12]

Heapsort

Heapsort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O(log n) time, instead of O(n) for a linear scan as in simple selection sort. This allows Heapsort to run in O(n log n) time, and this is also the worst case complexity.

Quicksort

Quicksort is a divide and conquer algorithm which relies on a *partition* operation: to partition an array an element called a *pivot* is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sublists are then recursively sorted. This yields average time complexity of O(n log n), with low overhead, and thus this is a popular algorithm. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries.

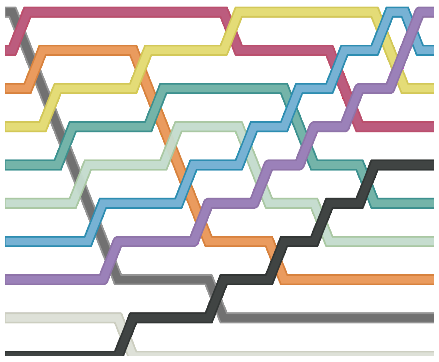
The important caveat about quicksort is that its worst-case performance is O(n^2); while this is rare, in naive implementations (choosing the first or last element as pivot) this occurs for sorted data, which is a common case. The most complex issue in quicksort is thus choosing a good pivot element, as consistently poor choices of pivots can result in drastically slower O(n^2) performance, but good choice of pivots yields O(n log n) performance, which is asymptotically optimal. For example, if at each step the median is chosen as the pivot then the algorithm works in O(n log n). Finding the median, such as by the median of medians selection algorithm is however an O(n) operation on unsorted lists and therefore exacts significant overhead with sorting. In practice choosing a random pivot almost certainly yields O(n log n) performance.

Bubble sort and variants

Bubble sort, and variants such as the cocktail sort, are simple but highly inefficient sorts. They are thus frequently seen in introductory texts, and are of some theoretical interest due to ease of analysis, but they are rarely used in practice, and primarily of recreational interest. Some variants, such as the Shell sort, have open questions about their behavior.

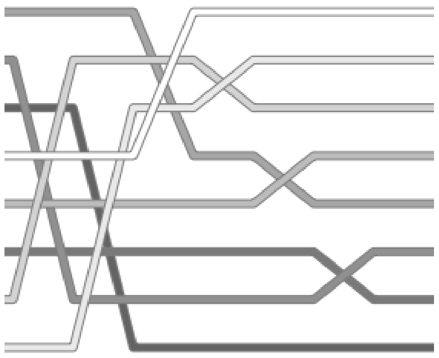
Bubble sort

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average and worst case performance is O(n^2), so it is rarely used to sort large, unordered data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty). Bubble sort can also be used efficiently on a list of any length that is nearly sorted (that is, the elements are not significantly out of place). For example, if any number of elements are out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort's exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only 2n time.


A bubble sort, a sorting algorithm that continuously steps through a list, swapping items until they appear in the correct order.

Shell sort

Shell sort was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.


A Shell sort, different from bubble sort in that it moves elements to numerous swapping positions

Comb sort

Comb sort is a relatively simple sorting algorithm originally designed by Wlodzimierz Dobosiewicz in 1980.[13] Later it was rediscovered and popularized by Stephen Lacey and Richard Box with a Byte Magazine article published in April 1991. Comb sort improves on bubble sort. The basic idea is to eliminate *turtles*, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. (*Rabbits*, large values around the beginning of the list, do not pose a problem in bubble sort)

Distribution sort

Distribution sort refers to any sorting algorithm where data are distributed from their input to multiple intermediate structures which are then gathered and placed on the output. For example, both bucket sort and flashsort are distribution based sorting algorithms. Distribution sorting algorithms can be used on a single processor, or they can be a distributed algorithm, where individual subsets are separately sorted on different processors, then combined. This allows external sorting of data too large to fit into a single computer's memory.

Counting sort

Counting sort is applicable when each input is known to belong to a particular set, S, of possibilities. The algorithm runs in $O(|S| + n)$ time and $O(|S|)$ memory where n is the length of the input. It works by creating an integer array of size $|S|$ and using the ith bin to count the occurrences of the ith member of S in the input. Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order. This sorting algorithm cannot often be used because S needs to be reasonably small for it to be efficient, but the algorithm is extremely fast and demonstrates great asymptotic behavior as n increases. It also can be modified to provide stable behavior.

Bucket sort

Bucket sort is a divide and conquer sorting algorithm that generalizes Counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm. A variation of this method called the single buffered count sort is faster than quicksort.[*citation needed*]

Due to the fact that bucket sort must use a limited number of buckets it is best suited to be used on data sets of a limited scope. Bucket sort would be unsuitable for data that have a lot of variation, such as social security numbers.

Radix sort

Radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in $O(n \cdot k)$ time. Radix sort can process digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD). The LSD algorithm first sorts the list by the least significant digit while preserving their relative order using a stable sort. Then it sorts them by the next digit, and so on from the least significant to the most significant, ending up with a sorted list. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable. It is common for the counting sort algorithm to be used internally by the radix sort. A hybrid sorting approach, such as using insertion sort for small bins improves performance of radix sort significantly.

Memory usage patterns and index sorting

When the size of the array to be sorted approaches or exceeds the available primary memory, so that (much slower) disk or swap space must be employed, the memory usage pattern of a sorting algorithm becomes important, and an algorithm that might have been fairly efficient when the array fit easily in RAM may become impractical. In this scenario, the total number of comparisons becomes (relatively) less important, and the number of times sections of memory must be copied or swapped to and from the disk can dominate the performance characteristics of an algorithm. Thus, the number of passes and the localization of comparisons can be more important than the raw number of comparisons, since comparisons of nearby elements to one another happen at system bus speed (or, with caching, even at CPU speed), which, compared to disk speed, is virtually instantaneous.

For example, the popular recursive quicksort algorithm provides quite reasonable performance with adequate RAM, but due to the recursive way that it copies portions of the array it becomes much less practical when the array does not fit in RAM, because it may cause a number of slow copy or move operations to and from disk. In that scenario, another algorithm may be preferable even if it requires more total comparisons.

One way to work around this problem, which works well when complex records (such as in a relational database) are being sorted by a relatively small key field, is to create an index into the array and then sort the index, rather than the entire array. (A sorted version of the entire array can then be produced with one pass, reading from the index, but often even that is unnecessary, as having the sorted index is adequate.) Because the index is much smaller than the entire array, it may fit easily in memory where the entire array would not, effectively eliminating the disk-swapping problem. This procedure is sometimes called "tag sort".[14]

Another technique for overcoming the memory-size problem is to combine two algorithms in a way that takes advantages of the strength of each to improve overall performance. For instance, the array might be subdivided into chunks of a size that will fit in RAM, the contents of each chunk sorted using an efficient algorithm (such as quicksort), and the results merged using a *k*-way merge similar to that used in mergesort. This is faster than performing either mergesort or quicksort over the entire list.

Techniques can also be combined. For sorting very large sets of data that vastly exceed system memory, even the index may need to be sorted using an algorithm or combination of algorithms designed to perform reasonably with virtual memory, i.e., to reduce the amount of swapping required.

Inefficient/humorous sorts

Some algorithms are slow compared to those discussed above, such as the bogosort $O(n \cdot n!)$ and the stooge sort $O(n^{2.7})$.

Related algorithms

Related problems include partial sorting (sorting only the *k* smallest elements of a list, or alternatively computing the *k* smallest elements, but unordered) and selection (computing the *k*th smallest element). These can be solved inefficiently by a total sort, but more efficient algorithms exist, often derived by generalizing a sorting algorithm. The most notable example is quickselect, which is related to quicksort. Conversely, some sorting algorithms can be derived by repeated application of a selection algorithm; quicksort and quickselect can be seen as the same pivoting move, differing only in whether one recurses on both sides (quicksort, divide and conquer) or one side (quickselect, decrease and conquer).

A kind of opposite of a sorting algorithm is a shuffling algorithm. These are fundamentally different because they require a source of random numbers. Interestingly, shuffling can also be implemented by a sorting algorithm, namely by a random sort: assigning a random number to each element of the list and then sorting based on the random numbers. This is generally not done in practice, however, and there is a well-known simple and efficient algorithm for shuffling: the Fisher–Yates shuffle.

References

[1] Demuth, H. Electronic Data Sorting. PhD thesis, Stanford University, 1956.

[2] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.54.8381

[3] http://www.algolist.net/Algorithms/Sorting/Selection_sort

[4] http://dbs.uni-leipzig.de/skripte/ADS1/PDF4/kap4.pdf

[5] Y. Han. *Deterministic sorting in UNIQ-math-0-d4396b1bef59a1cc-QINU time and linear space*. Proceedings of the thirty-fourth annual ACM symposium on Theory of computing, Montreal, Quebec, Canada, 2002,p.602-608.

[6] M. Thorup. *Randomized Sorting in $n \log n$ Time and Linear Space Using Addition, Shift, and Bit-wise Boolean Operations*. Journal of Algorithms, Volume 42, Number 2, February 2002, pp. 205-230(26)

[7] Han, Y. and Thorup, M. 2002. Integer Sorting in $n \log n$ Expected Time and Linear Space. In *Proceedings of the 43rd Symposium on Foundations of Computer Science* (November 16–19, 2002). FOCS. IEEE Computer Society, Washington, DC, 135-144.

[8] Tim Peters's original description of timsort (http://svn.python.org/projects/python/trunk/Objects/listsort.txt)

[9] http://hg.openjdk.java.net/jdk7/tl/jdk/rev/bfd7abda8f79

[10] Perl sort documentation (http://perldoc.perl.org/functions/sort.html)
[11] Merge sort in Java 1.3 (http://java.sun.com/j2se/1.3/docs/api/java/util/Arrays.html#sort(java.lang.Object[])), Sun.
[12] Java 1.3 live since 2000
[13] Brejová, Bronislava. "Analyzing variants of Shellsort" (http://www.sciencedirect.com/science/article/pii/S0020019000002234)
[14] Definition of "tag sort" according to PC Magazine (http://www.pcmag.com/encyclopedia_term/0,2542,t=tag+sort&i=52532,00.asp)

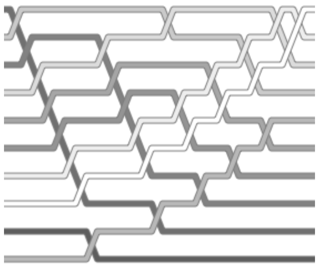
- D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*.

External links

- Sorting Algorithm Animations (http://www.sorting-algorithms.com/) - Graphical illustration of how different algorithms handle different kinds of data sets.
- Sequential and parallel sorting algorithms (http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/algoen.htm) - Explanations and analyses of many sorting algorithms.
- Dictionary of Algorithms, Data Structures, and Problems (http://www.nist.gov/dads/) - Dictionary of algorithms, techniques, common functions, and problems.
- Slightly Skeptical View on Sorting Algorithms (http://www.softpanorama.org/Algorithms/sorting.shtml) Discusses several classic algorithms and promotes alternatives to the quicksort algorithm.
- 15 Sorting Algorithms in 6 Minutes (Youtube) (http://www.youtube.com/watch?v=kPRA0W1kECg) Visualization and "audibilization" of 15 Sorting Algorithms in 6 Minutes.

Bubble sort

Bubble sort



Class	Sorting algorithm
Data structure	Array
Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n^2)$
Worst case space complexity	$O(1)$ auxiliary

Bubble sort, sometimes incorrectly referred to as **sinking sort**, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists.

Analysis

Performance

Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$. Even other $O(n^2)$ sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.

An example of bubble sort. Starting from the beginning of the list, compare every adjacent pair, swap their position if they are not in the right order (the latter one is smaller than the former one). After each iteration, one less element (the last one) is needed to be compared until there are no more elements left to be compared.

The only significant advantage that bubble sort has over most other implementations, even quicksort, but not insertion sort, is that the ability to detect that the list is sorted is efficiently built into the algorithm. Performance of bubble sort over an already-sorted list (best-case) is $O(n)$. By contrast, most other algorithms, even those with better average-case complexity, perform their entire sorting process on the set and thus are more complex. However, not only does insertion sort have this mechanism too, but it also performs better on a list that is substantially sorted (having a small number of inversions).

Bubble sort should be avoided in case of large collections. It will not be efficient in case of reverse ordered collection.

Rabbits and turtles

The positions of the elements in bubble sort will play a large part in determining its performance. Large elements at the beginning of the list do not pose a problem, as they are quickly swapped. Small elements towards the end, however, move to the beginning extremely slowly. This has led to these types of elements being named rabbits and turtles, respectively.

Various efforts have been made to eliminate turtles to improve upon the speed of bubble sort. Cocktail sort is a bi-directional bubble sort that goes from beginning to end, and then reverses itself, going end to beginning. It can move turtles fairly well, but it retains $O(n^2)$ worst-case complexity. Comb sort compares elements separated by large gaps, and can move turtles extremely quickly before proceeding to smaller and smaller gaps to smooth out the list. Its average speed is comparable to faster algorithms like quicksort.

Step-by-step example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass:
(**5** **1** 4 2 8) → (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5** **4** 2 8) → (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) → (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) → (1 4 2 **5** **8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1** **4** 2 5 8) → (**1** **4** 2 5 8)
(1 **4** **2** 545) → (1 **2** **4** 5 8), Swap since 4 > 2
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)
(1 2 4 **5** **8**) → (1 2 4 **5** **8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1** **2** 4 5 8) → (**1** **2** 4 5 8)
(1 **2** **4** 5 8) → (1 **2** **4** 5 8)
(1 2 **4** **5** 8) → (1 2 **4** **5** 8)
(1 2 4 **5** **8**) → (1 2 4 **5** **8**)

Implementation

Pseudocode implementation

The algorithm can be expressed as (0-based array):

```
procedure bubbleSort( A : list of sortable items )
   repeat
     swapped = false
     for i = 1 to length(A) - 1 inclusive do:
       /* if this pair is out of order */
       if A[i-1] > A[i] then
         /* swap them and remember something changed */
         swap( A[i-1], A[i] )
         swapped = true
       end if
     end for
   until not swapped
end procedure
```

Optimizing bubble sort

The bubble sort algorithm can be easily optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. So, the inner loop can avoid looking at the last n-1 items when running for the n-th time:

```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
       swapped = false
       for i = 1 to n-1 inclusive do
          if A[i-1] > A[i] then
             swap(A[i-1], A[i])
             swapped = true
          end if
       end for
       n = n - 1
    until not swapped
end procedure
```

More generally, it can happen that more than one element is placed in their final position on a single pass. In particular, after every pass, all elements after the last swap are sorted, and do not need to be checked again. This allows us to skip over a lot of the elements, resulting in about a worst case 50% improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the "swapped" variable:

To accomplish this in pseudocode we write the following:

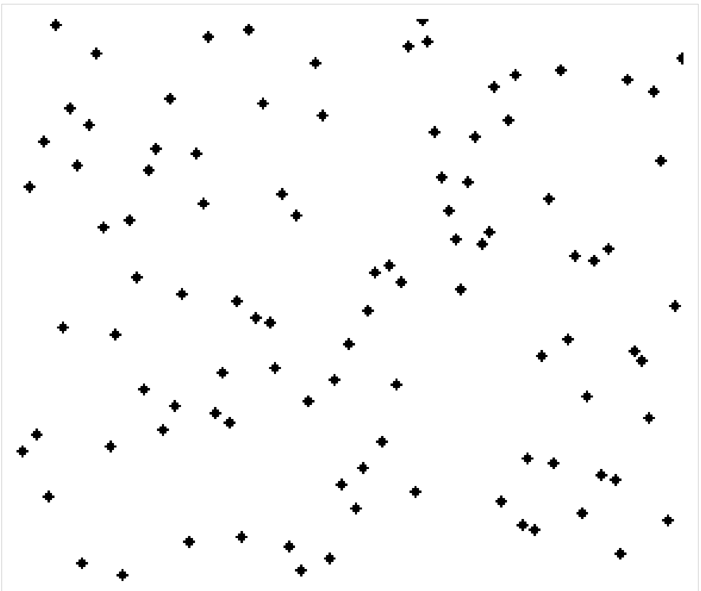
```
procedure bubbleSort( A : list of sortable items )
    n = length(A)
    repeat
       newn = 0
       for i = 1 to n-1 inclusive do
```

```
          if A[i-1] > A[i] then
             swap(A[i-1], A[i])
             newn = i
          end if
       end for
       n = newn
    until n = 0
end procedure
```

Alternate modifications, such as the cocktail shaker sort attempt to improve on the bubble sort performance while keeping the same idea of repeatedly comparing and swapping adjacent items.

In practice

Although bubble sort is one of the simplest sorting algorithms to understand and implement, its *O(n^2)* complexity means that its efficiency decreases dramatically on lists of more than a small number of elements. Even among simple *O(n^2)* sorting algorithms, algorithms like insertion sort are usually considerably more efficient.


A bubble sort, a sorting algorithm that continuously steps through a list, swapping items until they appear in the correct order. The list was plotted in a Cartesian coordinate system, with each point (x,y) indicating that the value y is stored at index x. Then the list would be sorted by Bubble sort according to every pixel's value. Note that the largest end gets sorted first, with smaller elements taking longer to move to their correct positions.

Due to its simplicity, bubble sort is often used to introduce the concept of an algorithm, or a sorting algorithm, to introductory computer science students. However, some researchers such as Owen Astrachan have gone to great lengths to disparage bubble sort and its continued popularity in computer science education, recommending that it no longer even be taught.

The Jargon file, which famously calls bogosort "the archetypical [sic] perversely awful algorithm", also calls bubble sort "the generic **bad** algorithm".[1] Donald Knuth, in his famous book *The Art of Computer Programming*, concluded that "the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems", some of which he then discusses.[2]

Bubble sort is asymptotically equivalent in running time to insertion sort in the worst case, but the two algorithms differ greatly in the number of swaps necessary. Experimental results such as those of Astrachan have also shown that insertion sort performs considerably better even on random lists. For these reasons many modern algorithm textbooks avoid using the bubble sort algorithm in favor of insertion sort.

Bubble sort also interacts poorly with modern CPU hardware. It requires at least twice as many writes as insertion sort, twice as many cache misses, and asymptotically more branch mispredictions. Experiments by Astrachan sorting strings in Java show bubble sort to be roughly 5 times slower than insertion sort and 40% slower than selection sort.[]

In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines.

Variations

- Odd-even sort is a parallel version of bubble sort, for message passing systems.
- Cocktail sort is another parallel version of the bubble sort
- In some cases, the sort works from right to left (the opposite direction), which is more appropriate for partially sorted lists, or lists with unsorted items added to the end.

Alone bubble sort

Alone bubble sort is a 1992 modification[*citation needed*] of the simple bubble sorting algorithm. Unlike the normal bubble sort where the loop resets after every performed swap of elements, in the alone bubble sort, the loop index only returns back by one step thus allowing the swapping to continue until a smaller value element in the array is reached. The following is the alone bubble realization (the algorithm only) in Pascal:

```
f:=false;
for x:=1 to max-1 do
    if a[x]>a[x+1] then
        begin
            if f=false then d:=x;
            f:=true;
            t:=a[x];
            a[x]:=a[x+1];
            a[x+1]:=t;
            if x>1 then dec(x,2) else x:=0;
        end
    else
        if f=true then
            begin
                x:=d;
                f:=false;
            end;
```

Debate Over Name

Bubble sort has occasionally been referred to as a "sinking sort," including by the National Institute of Standards and Technology. [3]

However, in Donald Knuth's *The Art of Computer Programming*, Volume 3: *Sorting and Searching* he states in section 5.2.1 'Sorting by Insertion', that [the value] "settles to its proper level" this method of sorting has often been called the *sifting* or *sinking* technique. Furthermore the *larger* values might be regarded as *heavier* and therefore be seen to progressively *sink* to the *bottom* of the list.

Notes

[1] http://www.jargon.net/jargonfile/b/bogo-sort.html

[2] Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Pages 106–110 of section 5.2.2: Sorting by Exchanging.

[3] http://xlinux.nist.gov/dads/HTML/bubblesort.html

References

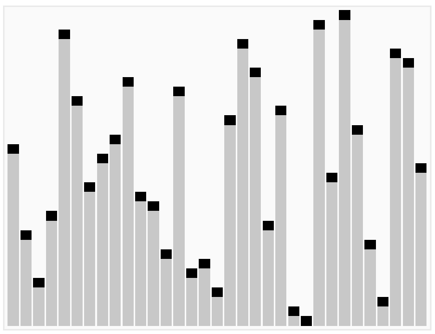
- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 106–110 of section 5.2.2: Sorting by Exchanging.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Problem 2-2, pg.38.
- Sorting in the Presence of Branch Prediction and Caches (https://www.cs.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-57.pdf)
- Fundamentals of Data Structures by Ellis Horowitz, Sartaj Sahni and Susan Anderson-Freed ISBN 81-7371-605-6

External links

- "Bubble Sort implemented in 34 languages" (http://codecodex.com/wiki/Bubble_sort).
- David R. Martin. "Animated Sorting Algorithms: Bubble Sort" (http://www.sorting-algorithms.com/bubble-sort). – graphical demonstration and discussion of bubble sort
- "Lafore's Bubble Sort" (http://lecture.ecc.u-tokyo.ac.jp/~ueda/JavaApplet/BubbleSort.html). (Java applet animation)
- (sequence A008302 in OEIS) Table (statistics) of the number of permutations of [n] that need k pair-swaps during the sorting.

Quicksort

Quicksort



Visualization of the quicksort algorithm. The horizontal lines are pivot values.

Class	Sorting algorithm
Worst case performance	O(n^2)
Best case performance	O(*n* log *n)*
Average case performance	O(*n* log *n*)

Worst case space complexity	O(*n*) auxiliary (naive)
	O(log *n*) auxiliary (Sedgewick 1978)

Quicksort, or **partition-exchange sort**, is a sorting algorithm developed by Tony Hoare that, on average, makes O(*n* log *n*) comparisons to sort *n* items. In the worst case, it makes O(n^2) comparisons, though this behavior is rare. Quicksort is often faster in practice than other O(*n* log *n*) algorithms. Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log *n*) additional space used by the stack during the recursion.

History

The quicksort algorithm was developed in 1960 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University. At that time, Hoare worked in a project on machine translation for the National Physical Laboratory. He developed the algorithm in order to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.

Algorithm

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a **pivot**, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.

The base case of the recursion are lists of size zero or one, which never need to be sorted.

Simple version

In simple pseudocode, the algorithm might be expressed as this:

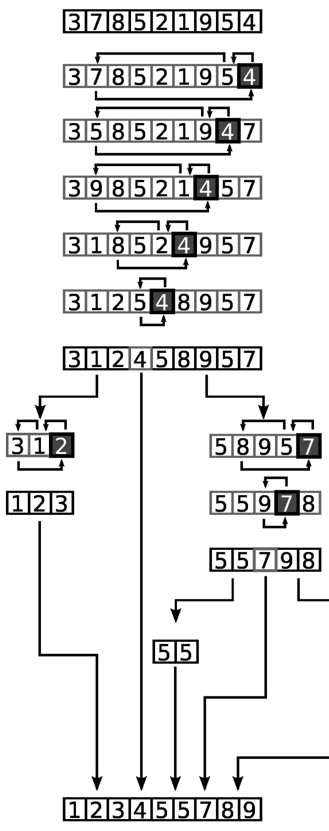
```
function quicksort(array)
    if length(array) ≤ 1
        return array  // an array of zero or one elements is already sorted
    select and remove a pivot element pivot from 'array'  // see '#Choice of pivot' below
    create empty lists less and greater
    for each x in array
        if x ≤ pivot then append x to less
        else append x to greater
    return concatenate(quicksort(less), list(pivot), quicksort(greater)) // two recursive calls
```

Notice that we only examine elements by comparing them to other elements. This makes quicksort a comparison sort. This version is also a stable sort (assuming that the "for each" method retrieves elements in original order, and the pivot selected is the last among those of equal value).

The correctness of the partition algorithm is based on the following two arguments:

- At each iteration, all the elements processed so far are in the desired position: before the pivot if less than the pivot's value, after the pivot if greater than the pivot's value (loop invariant).
- Each iteration leaves one fewer element to be processed (loop variant).

The correctness of the overall algorithm can be proven via induction: for zero or one element, the algorithm leaves the data unchanged; for a larger data set it produces the concatenation of two parts, elements less than the pivot and elements greater than it, themselves sorted by the recursive hypothesis.

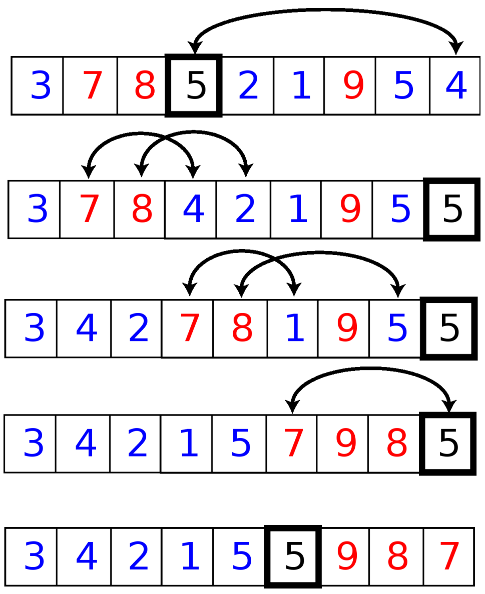


Full example of quicksort on a random set of numbers. The shaded element is the pivot. It is always chosen as the last element of the partition. However, always choosing the last element in the partition as the pivot in this way results in poor performance ($O(n^2)$) on *already sorted* lists, or lists of identical elements. Since sub-lists of sorted / identical elements crop up a lot towards the end of a sorting procedure on a large set, versions of the quicksort algorithm which choose the pivot as the middle element run much more quickly than the algorithm described in this diagram on large sets of numbers.

An example of quicksort.

In-place version

The disadvantage of the simple version above is that it requires O(*n*) extra storage space, which is as bad as naïve merge sort. The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complex version which uses an in-place partition algorithm and can achieve the complete sort using O(log *n*) space (not counting the input) on average (for the call stack). We start with a partition function:



In-place partition in action on a small list. The boxed element is the pivot element, blue elements are less or equal, and red elements are larger.

```
// left is the index of the leftmost element of the subarray
// right is the index of the rightmost element of the subarray (inclusive)
// number of elements in subarray = right-left+1
function partition(array, left, right, pivotIndex)
   pivotValue := array[pivotIndex]
   swap array[pivotIndex] and array[right]  // Move pivot to end
   storeIndex := left
   for i from left to right - 1  // left ≤ i < right
       if array[i] <= pivotValue
           swap array[i] and array[storeIndex]
           storeIndex := storeIndex + 1  // only increment storeIndex if swapped
   swap array[storeIndex] and array[right]  // Move pivot to its final place
```

```
    return storeIndex
```

This is the in-place partition algorithm. It partitions the portion of the array between indexes *left* and *right*, inclusively, by moving all elements less than `array[pivotIndex]` before the pivot, and the equal or greater elements after it. In the process it also finds the final position for the pivot element, which it returns. It temporarily moves the pivot element to the end of the subarray, so that it doesn't get in the way. Because it only uses exchanges, the final list has the same elements as the original list. Notice that an element may be exchanged multiple times before reaching its final place. Also, in case of pivot duplicates in the input array, they can be spread across the right subarray, in any order. This doesn't represent a partitioning failure, as further sorting will reposition and finally "glue" them together.

This form of the partition algorithm is not the original form; multiple variations can be found in various textbooks, such as versions not having the storeIndex. However, this form is probably the easiest to understand.

Once we have this, writing quicksort itself is easy:

```
function quicksort(array, left, right)
    // If the list has 2 or more items
    if left < right
        // See "#Choice of pivot" section below for possible choices
        choose any pivotIndex such that left ≤ pivotIndex ≤ right
        // Get lists of bigger and smaller items and final position of pivot
        pivotNewIndex := partition(array, left, right, pivotIndex)
        // Recursively sort elements smaller than the pivot
        quicksort(array, left, pivotNewIndex - 1)
        // Recursively sort elements at least as big as the pivot
        quicksort(array, pivotNewIndex + 1, right)
```

Each recursive call to this *quicksort* function reduces the size of the array being sorted by at least one element, since in each invocation the element at *pivotNewIndex* is placed in its final position. Therefore, this algorithm is guaranteed to terminate after at most *n* recursive calls. However, since *partition* reorders elements within a partition, this version of quicksort is not a stable sort.

Implementation issues

Choice of pivot

In very early versions of quicksort, the leftmost element of the partition would often be chosen as the pivot element. Unfortunately, this causes worst-case behavior on already sorted arrays, which is a rather common use-case. The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot (as recommended by Sedgewick).

Selecting a pivot element is also complicated by the existence of integer overflow. If the boundary indices of the subarray being sorted are sufficiently large, the naïve expression for the middle index, *(left + right)/2*, will cause overflow and provide an invalid pivot index. This can be overcome by using, for example, *left + (right-left)/2* to index the middle element, at the cost of more complex arithmetic. Similar issues arise in some other methods of selecting the pivot element.

Optimizations

Two other important optimizations, also suggested by Sedgewick and widely used in practice are:[1][2]

- To make sure at most O(log N) space is used, recurse first into the smaller half of the array, and use a tail call to recurse into the other.
- Use insertion sort, which has a smaller constant factor and is thus faster on small arrays, for invocations on small arrays (i.e. where the length is less than a threshold *k* determined experimentally). This can be implemented by simply stopping the recursion when less than *k* elements are left, leaving the entire array *k*-sorted: each element will be at most *k* positions away from its final position. Then, a single insertion sort pass finishes the sort in O(*k*×*n*) time. A separate insertion sort of each small segment as they are identified adds the overhead of starting and stopping many small sorts, but avoids wasting effort comparing keys across the many segment boundaries, which keys will be in order due to the workings of the quicksort process.

Parallelization

Like merge sort, quicksort can also be parallelized due to its divide-and-conquer nature. Individual in-place partition operations are difficult to parallelize, but once divided, different sections of the list can be sorted in parallel. The following is a straightforward approach: If we have p processors, we can divide a list of n elements into p sublists in *O*(*n*) average time, then sort each of these in $O\left(\frac{n}{p}\log\frac{n}{p}\right)$ average time. Ignoring the *O*(*n*) preprocessing and merge times, this is linear speedup. If the split is blind, ignoring the values, the merge naïvely costs *O*(*n*). If the split partitions based on a succession of pivots, it is tricky to parallelize and naïvely costs *O*(*n*). Given *O*(log *n*) or more processors, only *O*(*n*) time is required overall, whereas an approach with linear speedup would achieve *O*(log *n*) time for overall.

One advantage of this simple parallel quicksort over other parallel sort algorithms is that no synchronization is required, but the disadvantage is that sorting is still *O*(*n*) and only a sublinear speedup of *O*(log *n*) is achieved. A new thread is started as soon as a sublist is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.

Other more sophisticated parallel sorting algorithms can achieve even better time bounds. For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in *O*(log *n*) time on a CRCW PRAM with *n* processors by performing partitioning implicitly.[3]

Formal analysis

Average-case analysis using discrete probability

Quicksort takes *O*(*n* log *n*) time on average, when the input is a random permutation. Why? For a start, it is not hard to see that the partition operation takes *O*(*n*) time.

In the most unbalanced case, each time we perform a partition we divide the list into two sublists of size 0 and $n-1$ (for example, if all elements of the array are equal). This means each recursive call processes a list of size one less than the previous list. Consequently, we can make $n-1$ nested calls before we reach a list of size 1. This means that the call tree is a linear chain of $n-1$ nested calls. The i th call does $O(n-i)$ work to do the partition, and $\sum_{i=0}^{n}(n-i)=O(n^2)$, so in that case Quicksort takes $O(n^2)$ time. That is the worst case: given knowledge of which comparisons are performed by the sort, there are adaptive algorithms that are effective at generating worst-case input for quicksort on-the-fly, regardless of the pivot selection strategy.

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\log_2 n$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $\log_2 n$. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only *O*(*n*) time all together (each call has some constant overhead, but since there are only *O*(*n*) calls at each level, this is subsumed in the *O*(*n*) factor). The result is that the algorithm uses only *O*(*n* log *n*) time.

In fact, it's not necessary to be perfectly balanced; even if each pivot splits the elements with 75% on one side and 25% on the other side (or any other fixed fraction), the call depth is still limited to $\log_{4/3} n$, so the total running time is still *O*(*n* log *n*).

So what happens on average? If the pivot has rank somewhere in the middle 50 percent, that is, between the 25th percentile and the 75th percentile, then it splits the elements with at least 25% and at most 75% on each side. If we could consistently choose a pivot from the two middle 50 percent, we would only have to split the list at most $\log_{4/3} n$ times before reaching lists of size 1, yielding an *O*(*n* log *n*) algorithm.

When the input is a random permutation, the pivot has a random rank, and so it is not guaranteed to be in the middle 50 percent. However, when we start from a random permutation, in each recursive call the pivot has a random rank in its list, and so it is in the middle 50 percent about half the time. That is good enough. Imagine that you flip a coin: heads means that the rank of the pivot is in the middle 50 percent, tail means that it isn't. Imagine that you are flipping a coin over and over until you get *k* heads. Although this could take a long time, on average only 2*k* flips are required, and the chance that you won't get k heads after $100k$ flips is highly improbable (this can be made rigorous using Chernoff bounds). By the same argument, Quicksort's recursion will terminate on average at a call depth of only $2\log_{4/3} n$. But if its average call depth is *O*(log *n*), and each level of the call tree processes at most n elements, the total amount of work done on average is the product, *O*(*n* log *n*). Note that the algorithm does not have to verify that the pivot is in the middle half—if we hit it any constant fraction of the times, that is enough for the desired complexity.

Average-case analysis using recurrences

An alternative approach is to set up a recurrence relation for the *T*(*n*) factor, the time needed to sort a list of size n. In the most unbalanced case, a single Quicksort call involves *O*(*n*) work plus two recursive calls on lists of size 0 and $n-1$, so the recurrence relation is

$$T(n) = O(n) + T(0) + T(n-1) = O(n) + T(n-1).$$

This is the same relation as for insertion sort and selection sort, and it solves to worst case $T(n) = O(n^2)$.

In the most balanced case, a single quicksort call involves *O*(*n*) work plus two recursive calls on lists of size $n/2$, so the recurrence relation is

$$T(n) = O(n) + 2T\left(\frac{n}{2}\right).$$

The master theorem tells us that *T*(*n*) = *O*(*n* log *n*).

The outline of a formal proof of the *O*(*n* log *n*) expected time complexity follows. Assume that there are no duplicates as duplicates could be handled with linear time pre- and post-processing, or considered cases easier than the analyzed. When the input is a random permutation, the rank of the pivot is uniform random from 0 to *n*-1. Then the resulting parts of the partition have sizes i and n-i-1, and i is uniform random from 0 to *n*-1. So, averaging over all possible splits and noting that the number of comparisons for the partition is $n-1$, the average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-i-1))$$

Solving the recurrence gives $C(n) = 2n\ln n = 1.39n\log_2 n.$

This means that, on average, quicksort performs only about 39% worse than in its best case. In this sense it is closer to the best case than the worst case. Also note that a comparison sort cannot use less than $\log_2(n!)$ comparisons on average to sort n items (as explained in the article Comparison sort) and in case of large n, Stirling's

approximation yields $\log_2(n!) \approx n(\log_2 n - \log_2 e)$, so quicksort is not much worse than an ideal comparison sort. This fas
another reason for quicksort's practical dominance over other sorting algorithms.

Analysis of Randomized quicksort

Using the same analysis, one can show that Randomized quicksort has the desirable property that, for any input, it requires only *O*(*n* log *n*) expected time (averaged over all choices of pivots). However, there exists a combinatorial proof, more elegant than both the analysis using discrete probability and the analysis using recurrences.

To each execution of Quicksort corresponds the following binary search tree (BST): the initial pivot is the root node; the pivot of the left half is the root of the left subtree, the pivot of the right half is the root of the right subtree, and so on. The number of comparisons of the execution of Quicksort equals the number of comparisons during the construction of the BST by a sequence of insertions. So, the average number of comparisons for randomized Quicksort equals the average cost of constructing a BST when the values inserted $(x_1, x_2, ..., x_n)$ form a random permutation.

Consider a BST created by insertion of a sequence $(x_1, x_2, ..., x_n)$ of values forming a random permutation. Let C denote the cost of creation of the BST. We have: $C = \sum_i \sum_{j<i}$ (whether during the insertion of x_i there was a comparison to x_j).

By linearity of expectation, the expected value E(C) of C is $E(C) = \sum_i \sum_{j<i}$ Pr(during the insertion of x_i there was a comparison to x_j).

Fix i and j<i. The values $x_1, x_2, ..., x_j$, once sorted, define j+1 intervals. The core structural observation is that x_i is compared to x_j in the algorithm if and only if x_i falls inside one of the two intervals adjacent to x_j .

Observe that since $(x_1, x_2, ..., x_n)$ is a random permutation, $(x_1, x_2, ..., x_j, x_i)$ is also a random permutation, so the probability that x_i is adjacent to x_j is exactly $2/(j+1)$.

We end with a short calculation: $E(C) = \sum_i \sum_{j<i} 2/(j+1) = O(\sum_i \log i) = O(n \log n).$

Space complexity

The space used by quicksort depends on the version used.

The in-place version of quicksort has a space complexity of *O*(log *n*), even in the worst case, when it is carefully implemented using the following strategies:

- in-place partitioning is used. This unstable partition requires *O*(1) space.
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most *O*(log *n*) space. Then the other partition is sorted using tail recursion or iteration, which doesn't add to the call stack. This idea, as discussed above, was described by R. Sedgewick, and keeps the stack depth bounded by *O*(log *n*).

Quicksort with in-place and unstable partitioning uses only constant additional space before making any recursive call. Quicksort must store a constant amount of information for each nested recursive call. Since the best case makes at most *O*(log *n*) nested recursive calls, it uses *O*(log *n*) space. However, without Sedgewick's trick to limit the recursive calls, in the worst case quicksort could make *O*(*n*) nested recursive calls and need *O*(*n*) auxiliary space.

From a bit complexity viewpoint, variables such as *left* and *right* do not use constant space; it takes *O*(log *n*) bits to index into a list of *n* items. Because there are such variables in every stack frame, quicksort using Sedgewick's trick requires $O((\log n)^2)$ bits of space. This space requirement isn't too terrible, though, since if the list contained distinct elements, it would need at least *O*(*n* log *n*) bits of space.

Another, less common, not-in-place, version of quicksort uses *O*(*n*) space for working storage and can implement a stable sort. The working storage allows the input array to be easily partitioned in a stable manner and then copied back to the input array for successive recursive calls. Sedgewick's optimization is still appropriate.

Selection-based pivoting

A selection algorithm chooses the *k*th smallest of a list of numbers; this is an easier problem in general than sorting. One simple but effective selection algorithm works nearly in the same manner as quicksort, and is accordingly known as quickselect. The difference is that instead of making recursive calls on both sublists, it only makes a single tail-recursive call on the sublist which contains the desired element. This change lowers the average complexity to linear or $O(n)$ time, which is optimal for selection, but worst-case time is still $O(n^2)$.

A variant of quickselect, the median of medians algorithm, chooses pivots more carefully, ensuring that the pivots are near the middle of the data (between the 30th and 70th percentiles), and thus has guaranteed linear time – worst-case $O(n)$. This same pivot strategy can be used to construct a variant of quickselect (median of medians quicksort) with worst-case $O(n)$ time. However, the overhead of choosing the pivot is significant, so this is generally not used in practice.

More abstractly, given a worst-case $O(n)$ selection algorithm, one can use it to find the ideal pivot (the median) at every step of quicksort, producing a variant with worst-case $O(n \log n)$ running time. In practical implementations this variant is considerably slower on average, but it is of theoretical interest, showing how an optimal selection algorithm can yield an optimal sorting algorithm.

Variants

There are four well known variants of quicksort:

- **Balanced quicksort:** choose a pivot likely to represent the middle of the values to be sorted, and then follow the regular quicksort algorithm.
- **External quicksort:** The same as regular quicksort except the pivot is replaced by a buffer. First, read the M/2 first and last elements into the buffer and sort them. Read the next element from the beginning or end to balance writing. If the next element is less than the least of the buffer, write it to available space at the beginning. If greater than the greatest, write it to the end. Otherwise write the greatest or least of the buffer, and put the next element in the buffer. Keep the maximum lower and minimum upper keys written to avoid resorting middle elements that are in order. When done, write the buffer. Recursively sort the smaller partition, and loop to sort the remaining partition. This is a kind of three-way quicksort in which the middle partition (buffer) represents a sorted subarray of elements that are *approximately* equal to the pivot.
- **Three-way radix quicksort** (developed by Sedgewick and also known as **multikey quicksort**): is a combination of radix sort and quicksort. Pick an element from the array (the pivot) and consider the first character (key) of the string (multikey). Partition the remaining elements into three sets: those whose corresponding character is less than, equal to, and greater than the pivot's character. Recursively sort the "less than" and "greater than" partitions on the same character. Recursively sort the "equal to" partition by the next character (key). Given we sort using bytes or words of length W bits, the best case is O(KN) and the worst case $O(2^K N)$ or at least $O(N^2)$ as for standard quicksort, given for unique keys $N<2^K$, and K is a hidden constant in all standard comparison sort algorithms including quicksort. This is a kind of three-way quicksort in which the middle partition represents a (trivially) sorted subarray of elements that are *exactly* equal to the pivot.
- **Quick radix sort** (also developed by Powers as a o(K) parallel PRAM algorithm). This is again a combination of radix sort and quicksort but the quicksort left/right partition decision is made on successive bits of the key, and is thus O(KN) for N K-bit keys. Note that all comparison sort algorithms effectively assume an ideal K of O(logN) as if k is smaller we can sort in O(N) using a hash table or integer sorting, and if K >> logN but elements are unique within O(logN) bits, the remaining bits will not be looked at by either quicksort or quick radix sort, and otherwise all comparison sorting algorithms will also have the same overhead of looking through O(K) relatively useless bits but quick radix sort will avoid the worst case $O(N^2)$ behaviours of standard quicksort and quick radix

sort, and will be faster even in the best case of those comparison algorithms under these conditions of uniqueprefix(K) >> logN. See Powers [4] for further discussion of the hidden overheads in comparison, radix and parallel sorting.

Comparison with other sorting algorithms

Quicksort is a space-optimized version of the binary tree sort. Instead of inserting items sequentially into an explicit tree, quicksort organizes them concurrently into a tree that is implied by the recursive calls. The algorithms make exactly the same comparisons, but in a different order. An often desirable property of a sorting algorithm is stability - that is the order of elements that compare equal is not changed, allowing controlling order of multikey tables (e.g. directory or folder listings) in a natural way. This property is hard to maintain for in situ (or in place) quicksort (that uses only constant additional space for pointers and buffers, and logN additional space for the management of explicit or implicit recursion). For variant quicksorts involving extra memory due to representations using pointers (e.g. lists or trees) or files (effectively lists), it is trivial to maintain stability. The more complex, or disk-bound, data structures tend to increase time cost, in general making increasing use of virtual memory or disk.

The most direct competitor of quicksort is heapsort. Heapsort's worst-case running time is always $O(n \log n)$. But, heapsort is assumed to be on average somewhat slower than standard in-place quicksort. This is still debated and in research, with some publications indicating the opposite. Introsort is a variant of quicksort that switches to heapsort when a bad case is detected to avoid quicksort's worst-case running time. If it is known in advance that heapsort is going to be necessary, using it directly will be faster than waiting for introsort to switch to it.

Quicksort also competes with mergesort, another recursive sort algorithm but with the benefit of worst-case $O(n \log n)$ running time. Mergesort is a stable sort, unlike standard in-place quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Like mergesort, quicksort can be implemented as an in-place stable sort,[5] but this is seldom done. Although quicksort can easily be implemented as a stable sort using linked lists, it will often suffer from poor pivot choices without random access. The main disadvantage of mergesort is that, when operating on arrays, efficient implementations require $O(n)$ auxiliary space, whereas the variant of quicksort with in-place partitioning and tail recursion uses only $O(\log n)$ space. (Note that when operating on linked lists, mergesort only requires a small, constant amount of auxiliary storage.)

Bucket sort with two buckets is very similar to quicksort; the pivot in this case is effectively the value in the middle of the value range, which does well on average for uniformly distributed inputs.

One-parameter family of Partition sorts

Richard Cole and David C. Kandathil, in 2004, discovered a one-parameter family of sorting algorithms, called Partition sorts, which on average (with all input orderings equally likely) perform at most $n \log n + O(n)$ comparisons (close to the information theoretic lower bound) and $\Theta(n \log n)$ operations; at worst they perform $\Theta(n \log^2 n)$ comparisons (and also operations); these are in-place, requiring only additional $O(\log n)$ space. Practical efficiency and smaller variance in performance were demonstrated against optimised quicksorts (of Sedgewick and Bentley-McIlroy). [6]

Notes

[1] qsort.c in GNU libc: (http://www.cs.columbia.edu/~hgs/teaching/isp/hw/qsort.c), (http://repo.or.cz/w/glibc.git/blob/HEAD:/stdlib/qsort.c)

[2] http://www.ugrad.cs.ubc.ca/~cs260/chnotes/ch6/Ch6CovCompiled.html

[3] David M. W. Powers, Parallelized Quicksort and Radixsort with Optimal Speedup (http://citeseer.ist.psu.edu/327487.html), *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.

[4] David M. W. Powers, Parallel Unification: Practical Complexity (http://david.wardpowers.info/Research/AI/papers/199501-ACAW-PUPC.pdf), Australasian Computer Architecture Workshop, Flinders University, January 1995

[5] A Java implementation of in-place stable quicksort (http://h2database.googlecode.com/svn/trunk/h2/src/tools/org/h2/dev/sort/InPlaceStableQuicksort.java)

[6] Richard Cole, David C. Kandathil: "The average case analysis of Partition sorts" (http://www.cs.nyu.edu/cole/papers/part-sort.pdf), European Symposium on Algorithms, 14-17 September 2004, Bergen, Norway. Published: Lecture Notes in Computer Science 3221, Springer Verlag, pp. 240-251.

References

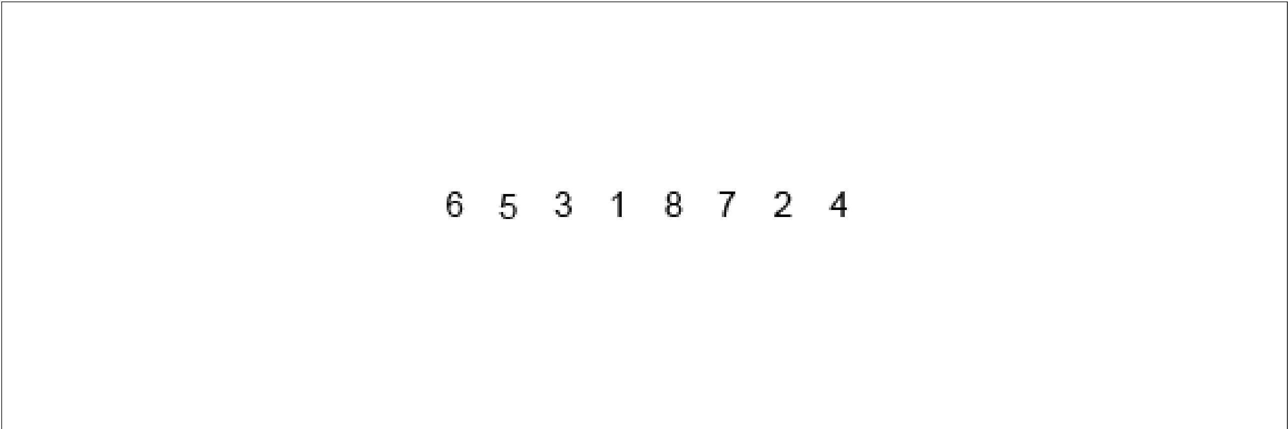
- Sedgewick, R. (1978). "Implementing Quicksort programs". *Comm. ACM* **21** (10): 847–857. doi: 10.1145/359619.359631 (http://dx.doi.org/10.1145/359619.359631).
- Dean, B. C. (2006). "A simple expected running time analysis for randomized "divide and conquer" algorithms". *Discrete Applied Mathematics* **154**: 1–5. doi: 10.1016/j.dam.2005.07.005 (http://dx.doi.org/10.1016/j.dam.2005.07.005).
- Hoare, C. A. R. (1961). "Algorithm 63: Partition". *Comm. ACM* **4** (7): 321. doi: 10.1145/366622.366642 (http://dx.doi.org/10.1145/366622.366642).
- Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". *Comm. ACM* **4** (7): 321. doi: 10.1145/366622.366644 (http://dx.doi.org/10.1145/366622.366644).
- Hoare, C. A. R. (1961). "Algorithm 65: Find". *Comm. ACM* **4** (7): 321–322. doi: 10.1145/366622.366647 (http://dx.doi.org/10.1145/366622.366647).
- Hoare, C. A. R. (1962). "Quicksort". *Comput. J.* **5** (1): 10–16. doi: 10.1093/comjnl/5.1.10 (http://dx.doi.org/10.1093/comjnl/5.1.10). (Reprinted in Hoare and Jones: *Essays in computing science* (http://portal.acm.org/citation.cfm?id=SERIES11430.63445), 1989.)
- Musser, David R. (1997). "Introspective Sorting and Selection Algorithms" (http://www.cs.rpi.edu/~musser/gp/introsort.ps). *Software: Practice and Experience* (Wiley) **27** (8): 983–993. doi: 10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-# (http://dx.doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#).
- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 113–122 of section 5.2.2: Sorting by Exchanging.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 7: Quicksort, pp. 145–164.
- A. LaMarca and R. E. Ladner. "The Influence of Caches on the Performance of Sorting." Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997. pp. 370–379.
- Faron Moller. Analysis of Quicksort (http://www.cs.swan.ac.uk/~csfm/Courses/CS_332/quicksort.pdf). CS 332: Designing Algorithms. Department of Computer Science, Swansea University.
- Martínez, C.; Roura, S. (2001). "Optimal Sampling Strategies in Quicksort and Quickselect". *SIAM J. Comput.* **31** (3): 683–705. doi: 10.1137/S0097539700382108 (http://dx.doi.org/10.1137/S0097539700382108).
- Bentley, J. L.; McIlroy, M. D. (1993). "Engineering a sort function". *Software: Practice and Experience* **23** (11): 1249–1265. doi: 10.1002/spe.4380231105 (http://dx.doi.org/10.1002/spe.4380231105).

External links

- Animated Sorting Algorithms: Quick Sort (http://www.sorting-algorithms.com/quick-sort) – graphical demonstration and discussion of quick sort
- Animated Sorting Algorithms: 3-Way Partition Quick Sort (http://www.sorting-algorithms.com/quick-sort-3-way) – graphical demonstration and discussion of 3-way partition quick sort
- Interactive Tutorial for Quicksort (http://pages.stern.nyu.edu/~panos/java/Quicksort/index.html)
- Quicksort applet (http://www.yorku.ca/sychen/research/sorting/index.html) with "level-order" recursive calls to help improve algorithm analysis
- Open Data Structures - Section 11.1.2 - Quicksort (http://opendatastructures.org/versions/edition-0.1e/ods-java/11_1_Comparison_Based_Sorti.html#SECTION001412000000000000000)
- Multidimensional quicksort in Java (http://fiehnlab.ucdavis.edu/staff/wohlgemuth/java/quicksort)
- Literate implementations of Quicksort in various languages (http://en.literateprograms.org/Category:Quicksort) on LiteratePrograms
- A colored graphical Java applet (http://coderaptors.com/?QuickSort) which allows experimentation with initial state and shows statistics

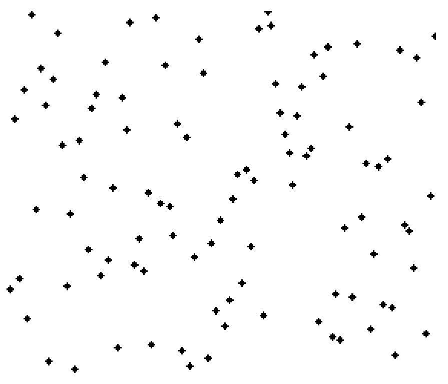
Merge sort

Merge sort



An example of merge sort. First divide the list into the smallest unit (1 element), then compare each element with the adjacent list to sort and merge the two adjacent lists. Finally all the elements are sorted and merged.

Class	Sorting algorithm
Data structure	Array
Worst case performance	O(*n* log *n*)
Best case performance	O(*n* log *n*) typical, O(*n*) natural variant
Average case performance	O(*n* log *n*)
Worst case space complexity	O(*n*) auxiliary



Merge sort animation. The sorted elements are represented by dots.

In computer science, a **merge sort** (also commonly spelled **mergesort**) is an *O*(*n* log *n*) comparison-based sorting algorithm. Most implementations produce a stable sort, which means that the implementation preserves the input order of equal elements in the sorted output. Mergesort is a divide and conquer algorithm that was invented by John von Neumann in 1945. A detailed description and analysis of bottom-up mergesort appeared in a report by Goldstine and Neumann as early as 1948.

Algorithm

Conceptually, a merge sort works as follows

1. Divide the unsorted list into *n* sublists, each containing 1 element (a list of 1 element is considered sorted).
2. Repeatedly merge sublists to produce new sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Top-down implementation

Example C like code using indices for top down merge sort algorithm that recursively splits the list (called *runs* in this example) into sublists until sublist size is 1, then merges those sublists to produce a sorted list. The copy back step could be avoided if the recursion alternated between two functions so that the direction of the merge corresponds with the level of recursion.

```
TopDownMergeSort(A[], B[], n)
{
    TopDownSplitMerge(A, 0, n, B);
}

TopDownSplitMerge(A[], iBegin, iEnd, B[])
{
    if(iEnd - iBegin < 2)                       // if run size == 1
        return;                                 //   consider it sorted
    // recursively split runs into two halves until run size == 1,
    // then merge them and return back up the call chain
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // split / merge left
half
    TopDownSplitMerge(A, iMiddle, iEnd,    B);  // split / merge right
half
    TopDownMerge(A, iBegin, iMiddle, iEnd, B);  // merge the two half
runs
    CopyArray(B, iBegin, iEnd, A);              // copy the merged runs
 back to A
}

TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i0 = iBegin, i1 = iMiddle;

    // While there are elements in the left or right runs
    for (j = iBegin; j < iEnd; j++) {
        // If left run head exists and is <= existing right run head.
        if (i0 < iMiddle && (i1 >= iEnd || A[i0] <= A[i1]))
            B[j] = A[i0++];  // Increment i0 after using it as an
index.
        else
            B[j] = A[i1++];  // Increment i1 after using it as an
index.
    }
}
```

Bottom-up implementation

Example code for C using indices for bottom up merge sort algorithm which treats the list as an array of *n* sublists (called *runs* in this example) of size 1, and iteratively merges sub-lists back and forth between two buffers:

```
/* array A[] has the items to sort; array B[] is a work array */
BottomUpSort(int n, int A[], int B[])
{
  int width;

  /* Each 1-element run in A is already "sorted". */

  /* Make successively longer sorted runs of length 2, 4, 8, 16...
until whole array is sorted. */
  for (width = 1; width < n; width = 2 * width)
    {
      int i;

      /* Array A is full of runs of length width. */
      for (i = 0; i < n; i = i + 2 * width)
        {
          /* Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1]
to B[] */
          /* or copy A[i:n-1] to B[] ( if(i+width >= n) ) */
          BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B);
        }

      /* Now work array B is full of runs of length 2*width. */
      /* Copy array B to array A for next iteration. */
      /* A more efficient implementation would swap the roles of A and
B */
      CopyArray(A, B, n);
      /* Now array A is full of runs of length 2*width. */
    }
}

BottomUpMerge(int A[], int iLeft, int iRight, int iEnd, int B[])
{
  int i0 = iLeft;
  int i1 = iRight;
  int j;

  /* While there are elements in the left or right lists */
  for (j = iLeft; j < iEnd; j++)
    {
      /* If left list head exists and is <= existing right list head */
      if (i0 < iRight && (i1 >= iEnd || A[i0] <= A[i1]))
        {
          B[j] = A[i0];
```

```
            i0 = i0 + 1;
          }
        else
          {
            B[j] = A[i1];
            i1 = i1 + 1;
          }
      }
}
```

Top-down implementation using lists

Example pseudocode for top down merge sort algorithm which uses recursion to divide the list into sub-lists, then merges sublists during returns back up the call chain.

```
function merge_sort(list m)
    // if list size is 0 (empty) or 1, consider it sorted and return it
    // (using less than or equal prevents infinite recursion for a zero length m)
    if length(m) <= 1
        return m
    // else list size is > 1, so split the list into two sublists
    // 1. DIVIDE Part...
    var list left, right
    var integer middle = length(m) / 2
    for each x in m before middle
         add x to left
    for each x in m after or equal middle
         add x to right
    // recursively call merge_sort() to further split each sublist
    // until sublist size is 1
    left = merge_sort(left)
    right = merge_sort(right)
    // merge the sublists returned from prior calls to merge_sort()
    // and return the resulting merged sublist
    // 2. CONQUER Part...
    return merge(left, right)
```

In this example, the `merge` function merges the left and right sublists.

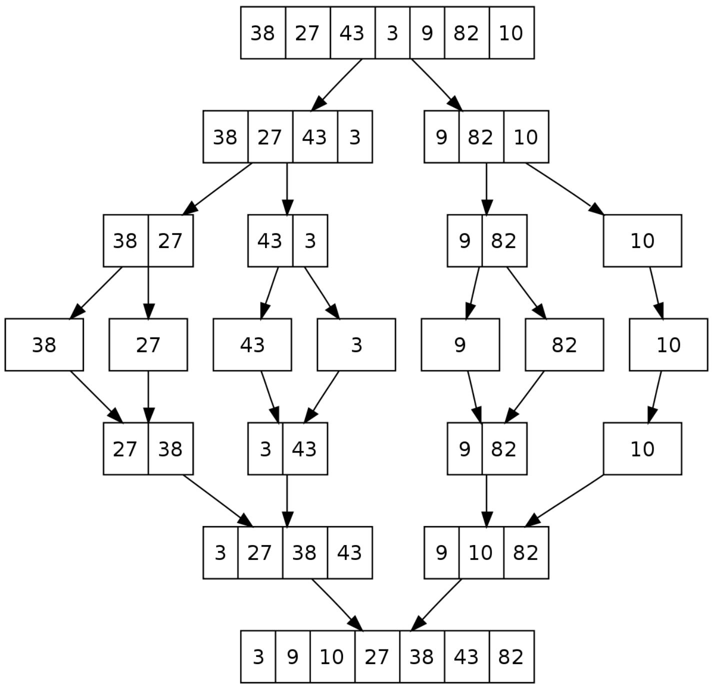
```
function merge(left, right)
    // receive the left and right sublist as arguments.
    // 'result' variable for the merged result of two sublists.
    var list result
    // assign the element of the sublists to 'result' variable until there is no element to merge.
    while length(left) > 0 or length(right) > 0
        if length(left) > 0 and length(right) > 0
           // compare the first two element, which is the small one, of each two sublists.
            if first(left) <= first(right)
                // the small element is copied to 'result' variable.
                // delete the copied one(a first element) in the sublist.
                append first(left) to result
                left = rest(left)
            else
                // same operation as the above(in the right sublist).
                append first(right) to result
                right = rest(right)
        else if length(left) > 0
            // copy all of remaining elements from the sublist to 'result' variable,
            // when there is no more element to compare with.
            append first(left) to result
            left = rest(left)
        else if length(right) > 0
            // same operation as the above(in the right sublist).
            append first(right) to result
            right = rest(right)
    end while
    // return the result of the merged sublists(or completed one, finally).
    // the length of the left and right sublists will grow bigger and bigger, after the next call of this function.
    return result
```

Natural merge sort

A natural merge sort is similar to a bottom up merge sort except that any naturally occurring runs (sorted sequences) in the input are exploited. In the bottom up merge sort, the starting point assumes each run is one item long. In practice, random input data will have many short runs that just happen to be sorted. In the typical case, the natural merge sort may not need as many passes because there are fewer runs to merge. In the best case, the input is already sorted (i.e., is one run), so the natural merge sort need only make one pass through the data. Example:

```
Start       : 3--4--2--1--7--5--8--9--0--6
Select runs : 3--4  2  1--7  5--8--9  0--6
Merge       : 2--3--4  1--5--7--8--9  0--6
Merge       : 1--2--3--4--5--7--8--9  0--6
Merge       : 0--1--2--3--4--5--6--7--8--9
```

Analysis



A recursive merge sort algorithm used to sort an array of 7 integer values. These are the steps a human would take to emulate merge sort (top-down).

In sorting n objects, merge sort has an average and worst-case performance of O(n log n). If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.

In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than ($n \lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$), which is between ($n \lg n - n + 1$) and ($n \lg n + n + O(\lg n)$).[1]

For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches $\alpha \cdot n$ fewer than the worst case where $\alpha = -1 + \sum_{k=0}^{\infty} \frac{1}{2^k+1} \approx 0.2645.$

In the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case. In terms of moves, merge sort's worst case complexity is O(n log n)—the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.[*citation needed*]

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort as long as the merge operation is implemented properly.

Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only n/2 extra spaces).

Merge sort also has some demerits. One is its use of 2n locations; the additional n locations are commonly used because merging two sorted sets in place is more complicated and would need more comparisons and move operations. But despite the use of this space the algorithm still does a lot of work: The contents of *m* are first copied into *left* and *right* and later into the list *result* on each invocation of *merge_sort* (variable names according to the pseudocode above).

Variants

Variants of merge sort are primarily concerned with reducing the space complexity and the cost of copying.

A simple alternative for reducing the space overhead to n/2 is to maintain *left* and *right* as a combined structure, copy only the *left* part of *m* into temporary space, and to direct the *merge* routine to place the merged output into *m*. With this version it is better to allocate the temporary space outside the *merge* routine, so that only one allocation is needed. The excessive copying mentioned previously is also mitigated, since the last pair of lines before the *return result* statement (function *merge* in the pseudo code above) become superfluous.

In-place sorting is possible, and still stable, but is more complicated, and slightly slower, requiring non-linearithmic quasilinear time O(n log^2 n) (Katajainen, Pasanen & Teuhola 1996). One way to sort in-place is to merge the blocks recursively.[2] Like the standard merge sort, in-place merge sort is also a stable sort. Stable sorting of linked lists is simpler. In this case the algorithm does not use more space than that already used by the list representation, but the O(log(k)) used for the recursion trace.

An alternative to reduce the copying into multiple lists is to associate a new field of information with each key (the elements in *m* are called keys). This field will be used to link the keys and any associated information together in a sorted list (a key and its related information is called a record). Then the merging of the sorted lists proceeds by changing the link values; no records need to be moved at all. A field which contains only a link will generally be smaller than an entire record so less space will also be used. This is a standard sorting technique, not restricted to merge sort.

Use with tape drives

An external merge sort is practical to run using disk or tape drives when the data to be sorted is too large to fit into memory. External sorting explains how merge sort is implemented with disk drives. A typical tape drive sort uses four tape drives. All I/O is sequential (except for rewinds at the end of each pass). A minimal implementation can get by with just 2 record buffers and a few program variables.

Merge sort type algorithms allowed large data sets to be sorted on early computers that had small random access memories by modern standards. Records were stored on magnetic tape and processed on banks of magnetic tape drives, such as these IBM 729s.

Naming the four tape drives as A, B, C, D, with the original data on A, and using only 2 record buffers, the algorithm is similar to Bottom-up implementation, using pairs of tape drives instead of arrays in memory. The basic algorithm can be described as follows:

1. Merge pairs of records from A; writing two-record sublists alternately to C and D.
2. Merge two-record sublists from C and D into four-record sublists; writing these alternately to A and B.
3. Merge four-record sublists from A and B into eight-record sublists; writing these alternately to C and D
4. Repeat until you have one list containing all the data, sorted --- in log2(n) passes.

Instead of starting with very short runs, usually a hybrid algorithm is used, where the initial pass will read many records into memory, do an internal sort to create a long run, and then distribute those long runs onto the output set. The step avoids many early passes. For example, an internal sort of 1024 records will save 9 passes. The internal sort is often large because it has such a benefit. In fact, there are techniques that can make the initial runs longer than the available internal memory.[3]

A more sophisticated merge sort that optimizes tape (and disk) drive usage is the polyphase merge sort.

Optimizing merge sort

On modern computers, locality of reference can be of paramount importance in software optimization, because multilevel memory hierarchies are used. Cache-aware versions of the merge sort algorithm, whose operations have been specifically chosen to minimize the movement of pages in and out of a machine's memory cache, have been proposed. For example, the **tiled merge sort** algorithm stops partitioning subarrays when subarrays of size S are reached, where S is the number of data items fitting into a CPU's cache. Each of these subarrays is sorted with an in-place sorting algorithm such as insertion sort, to discourage memory swaps, and normal merge sort is then completed in the standard recursive fashion. This algorithm has demonstrated better performance on machines that benefit from cache optimization. (LaMarca & Ladner 1997)

Kronrod (1969) suggested an alternative version of merge sort that uses constant additional space. This algorithm was later refined. (Katajainen, Pasanen & Teuhola 1996).

Also, many applications of external sorting use a form of merge sorting where the input get split up to a higher number of sublists, ideally to a number for which merging them still makes the currently processed set of pages fit into main memory.

Parallel processing

Merge sort parallelizes well due to use of the divide-and-conquer method. A parallel implementation is shown in pseudo-code in the third edition of Cormen, Leiserson, Rivest, and Stein's *Introduction to Algorithms*. This algorithm uses a parallel merge algorithm to not only parallelize the recursive division of the array, but also the merge operation. It performs well in practice when combined with a fast stable sequential sort, such as insertion sort, and a fast sequential merge as a base case for merging small arrays.[4] Merge sort was one of the first sorting algorithms where optimal speed up was achieved, with Richard Cole using a clever subsampling algorithm to ensure $O(1)$ merge. Other sophisticated parallel sorting algorithms can achieve the same or better time bounds with a lower constant. For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in $O(\log n)$ time on a CRCW PRAM with n processors by performing partitioning implicitly.[5] Powers[6] further shows that a pipelined version of Batcher's Bitonic Mergesort at $O(\log^2 n)$ time on a butterfly sorting network is in practice actually faster than his $O(\log n)$ sorts on a PRAM, and he provides detailed discussion of the hidden overheads in comparison, radix and parallel sorting.

Comparison with other sort algorithms

Although heapsort has the same time bounds as merge sort, it requires only $\Theta(1)$ auxiliary space instead of merge sort's $\Theta(n)$, and is often faster in practical implementations. On typical modern architectures, efficient quicksort implementations generally outperform mergesort for sorting RAM-based arrays. On the other hand, merge sort is a stable sort and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl). In Java, the Arrays.sort() [7] methods use merge sort or a tuned quicksort depending on the datatypes and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.[8] Python uses timsort, another tuned hybrid of merge sort and insertion sort, that has become the standard sort algorithm in Java SE 7, on the Android platform, and in GNU Octave.

Utility in online sorting

Merge sort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning. In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list — a better approach in this case is to insert elements into a binary search tree as they are received.[*citation needed*]

Notes

[1] The worst case number given here does not agree with that given in Knuth's *Art of Computer Programming, Vol 3*. The discrepancy is due to Knuth analyzing a variant implementation of merge sort that is slightly sub-optimal

[2] A Java implementation of in-place stable merge sort (http://h2database.googlecode.com/svn/trunk/h2/src/tools/org/h2/dev/sort/InPlaceStableMergeSort.java)

[3] Selection sort. Knuth's snowplow. Natural merge.

[4] V. J. Duvanenko, "Parallel Merge Sort", Dr. Dobb's Journal, March 2011 (http://drdobbs.com/high-performance-computing/229400239)

[5] Powers, David M. W. Parallelized Quicksort and Radixsort with Optimal Speedup (http://citeseer.ist.psu.edu/327487.html), *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.

[6] David M. W. Powers, Parallel Unification: Practical Complexity (http://david.wardpowers.info/Research/AI/papers/199501-ACAW-PUPC.pdf), Australasian Computer Architecture Workshop, Flinders University, January 1995

[7] http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html

[8] OpenJDK Subversion (https://openjdk.dev.java.net/source/browse/openjdk/jdk/trunk/jdk/src/share/classes/java/util/Arrays.java?view=markup)

References

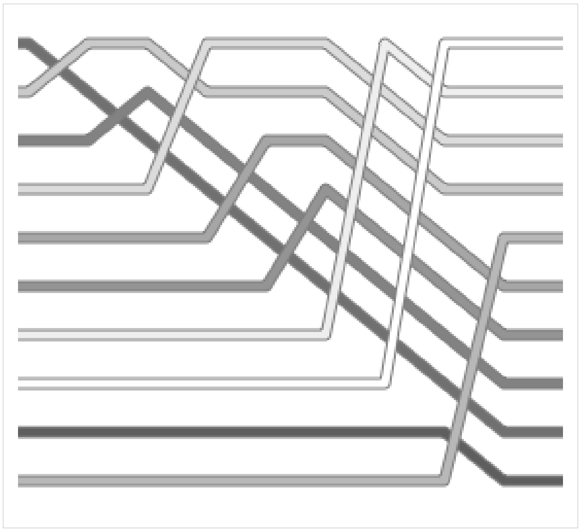
- Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
- Katajainen, Jyrki; Pasanen, Tomi; Teuhola, Jukka (1996). "Practical in-place mergesort" (http://www.diku.dk/hjemmesider/ansatte/jyrki/Paper/mergesort_NJC.ps). *Nordic Journal of Computing* **3**. pp. 27–40. ISSN 1236-6064 (http://www.worldcat.org/issn/1236-6064). Retrieved 2009-04-04.. Also Practical In-Place Mergesort (http://citeseer.ist.psu.edu/katajainen96practical.html). Also (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.8523)
- Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". *Sorting and Searching*. The Art of Computer Programming **3** (2nd ed.). Addison-Wesley. pp. 158–168. ISBN 0-201-89685-0.
- Kronrod, M. A. (1969). "Optimal ordering algorithm without operational field". *Soviet Mathematics - Doklady* **10**. p. 744.
- LaMarca, A.; Ladner, R. E. (1997). "The influence of caches on the performance of sorting". *Proc. 8th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA97)*: 370–379.
- Sun Microsystems. "Arrays API" (http://java.sun.com/javase/6/docs/api/java/util/Arrays.html). Retrieved 2007-11-19.
- Sun Microsystems. "java.util.Arrays.java" (https://openjdk.dev.java.net/source/browse/openjdk/jdk/trunk/jdk/src/share/classes/java/util/Arrays.java?view=markup). Retrieved 2007-11-19.

External links

- Animated Sorting Algorithms: Merge Sort (http://www.sorting-algorithms.com/merge-sort) – graphical demonstration and discussion of array-based merge sort
- Dictionary of Algorithms and Data Structures: Merge sort (http://www.nist.gov/dads/HTML/mergesort.html)
- Mergesort applet (http://www.yorku.ca/sychen/research/sorting/index.html) with "level-order" recursive calls to help improve algorithm analysis
- Open Data Structures - Section 11.1.1 - Merge Sort (http://opendatastructures.org/versions/edition-0.1e/ods-java/11_1_Comparison_Based_Sorti.html#SECTION001411000000000000000)

Insertion sort

Insertion sort


Graphical illustration of insertion sort

Class	Sorting algorithm
Data structure	Array
Worst case performance	O(n^2) comparisons, swaps
Best case performance	O(n) comparisons, O(1) swaps
Average case performance	O(n^2) comparisons, swaps
Worst case space complexity	O(n) total, O(1) auxiliary

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- Simple implementation
- Efficient for (quite) small data sets
- Adaptive (i.e., efficient) for data sets that are already substantially sorted: the time complexity is O($n + d$), where d is the number of inversions
- More efficient in practice than most other simple quadratic (i.e., O(n^2)) algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is O(n)
- Stable; i.e., does not change the relative order of elements with equal keys
- In-place; i.e., only requires a constant amount O(1) of additional memory space
- Online; i.e., can sort a list as it receives it

When humans manually sort something (for example, a deck of playing cards), most use a method that is similar to insertion sort.[1]

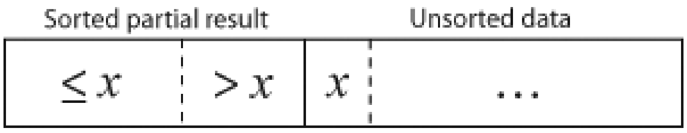
Algorithm

A graphical example of insertion sort.

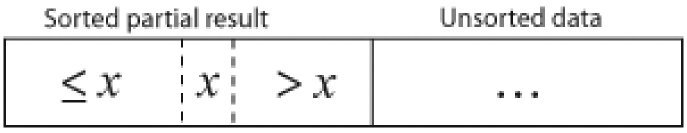
Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after *k* iterations has the property where the first *k* + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:



becomes



with each element greater than *x* copied to the right as it is compared against *x*.

The most common variant of insertion sort, which operates on arrays, can be described as follows:

1. Suppose there exists a function called *Insert* designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.
2. To perform an insertion sort, begin at the left-most element of the array and invoke *Insert* to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

Pseudocode of the complete algorithm follows, where the arrays are zero-based:

```
// The values in A[i] are checked in-order, starting at the second one
for i ← 1 to i ← length(A)
  {
    // at the start of the iteration, A[0..i-1] are in sorted order
    // this iteration will insert A[i] into that sorted order
    // save A[i], the value that will be inserted into the array on this iteration
    valueToInsert ← A[i]
    // now mark position i as the hole; A[i]=A[holePos] is now empty
    holePos ← i
    // keep moving the hole down until the valueToInsert is larger than
    // what's just below the hole or the hole has reached the beginning of the array
    while holePos > 0 and valueToInsert < A[holePos - 1]
      { //value to insert doesn't belong where the hole currently is, so shift
        A[holePos] ← A[holePos - 1] //shift the larger value up
        holePos ← holePos - 1       //move the hole position down
      }
    // hole is in the right position, so put valueToInsert into the hole
    A[holePos] ← valueToInsert
    // A[0..i] are now in sorted order
  }
```

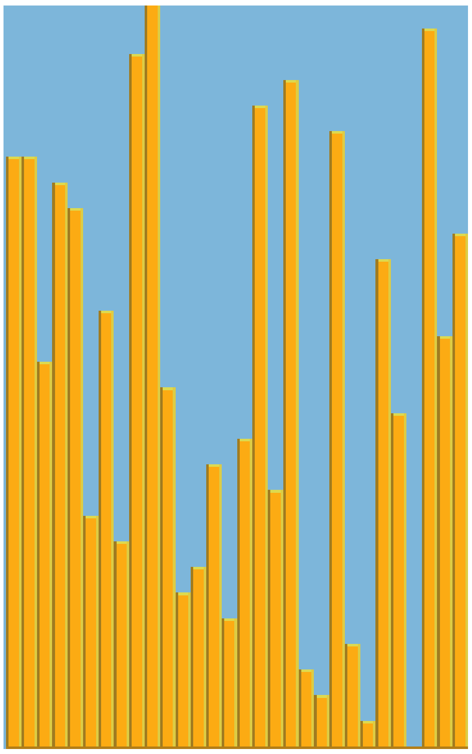
Note that although the common practice is to implement in-place, which requires checking the elements in-order, the order of checking (and removing) input elements is actually arbitrary. The choice can be made using almost any pattern, as long as all input elements are eventually checked (and removed from the input).

Best, worst, and average cases

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., Θ(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n^2)).

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.


Animation of the insertion sort sorting a 30 element array.

Example: The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the item under consideration is underlined. The item that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1

3 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

2 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

1 2 3 4 5 6 7 9

Comparisons to other sorting algorithms

Insertion sort is very similar to selection sort. As in selection sort, after k passes through the array, the first k elements are in sorted order. For selection sort these are the k smallest elements, while in insertion sort they are whatever the first k elements were in the unsorted array. Insertion sort's advantage is that it only scans as many elements as needed to determine the correct location of the k+1st element, while selection sort must scan all remaining elements to find the absolute smallest element.

Calculations show that insertion sort will usually perform about half as many comparisons as selection sort. Assuming the k+1st element's rank is random, insertion sort will on average require shifting half of the previous k elements, while selection sort always requires scanning all unplaced elements. If the input array is reverse-sorted, insertion sort performs as many comparisons as selection sort. If the input array is already sorted, insertion sort performs as few as n-1 comparisons, thus making insertion sort more efficient when given sorted or "nearly sorted" arrays.

While insertion sort typically makes fewer comparisons than selection sort, it requires more writes because the inner loop can require shifting large sections of the sorted portion of the array. In general, insertion sort will write to the array $O(n^2)$ times, whereas selection sort will write only $O(n)$ times. For this reason selection sort may be preferable in cases where writing to memory is significantly more expensive than reading, such as with EEPROM or flash memory.

Some divide-and-conquer algorithms such as quicksort and mergesort sort by recursively dividing the list into smaller sublists which are then sorted. A useful optimization in practice for these algorithms is to use insertion sort for sorting small sublists, where insertion sort outperforms these more complex algorithms. The size of list for which insertion sort has the advantage varies by environment and implementation, but is typically between eight and twenty elements.

Variants

D.L. Shell made substantial improvements to the algorithm; the modified version is called Shell sort. The sorting algorithm compares elements separated by a distance that decreases on each pass. Shell sort has distinctly improved running times in practical work, with two simple variants requiring $O(n^{3/2})$ and $O(n^{4/3})$ running time.

If the cost of comparisons exceeds the cost of swaps, as is the case for example with string keys stored by reference or with human interaction (such as choosing one of a pair displayed side-by-side), then using *binary insertion sort* may yield better performance. Binary insertion sort employs a binary search to determine the correct location to insert new elements, and therefore performs $\lceil \log_2(n) \rceil$ comparisons in the worst case, which is $O(n \log n)$. The algorithm as a whole still has a running time of $O(n^2)$ on average because of the series of swaps required for each insertion.

The number of swaps can be reduced by calculating the position of multiple elements before moving them. For example, if the target position of two elements is calculated before they are moved into the right position, the number of swaps can be reduced by about 25% for random data. In the extreme case, this variant works similar to merge sort.

To avoid having to make a series of swaps for each insertion, the input could be stored in a linked list, which allows elements to be spliced into or out of the list in constant-time when the position in the list is known. However, searching a linked list requires sequentially following the links to the desired position: a linked list does not have random access, so it cannot use a faster method such as binary search. Therefore, the running time required for searching is O(n) and the time for sorting is O(n^2). If a more sophisticated data structure (e.g., heap or binary tree) is used, the time required for searching and insertion can be reduced significantly; this is the essence of heap sort and binary tree sort.

In 2004 Bender, Farach-Colton, and Mosteiro published a new variant of insertion sort called *library sort* or *gapped insertion sort* that leaves a small number of unused spaces (i.e., "gaps") spread throughout the array. The benefit is that insertions need only shift elements over until a gap is reached. The authors show that this sorting algorithm runs with high probability in O(n log n) time.

If a skip list is used, the insertion time is brought down to O(log n), and swaps are not needed because the skip list is implemented on a linked list structure. The final running time for insertion would be O(n log n).

List insertion sort is a variant of insertion sort. It reduces the number of movements.[*citation needed*]

List insertion sort code in C

If the items are stored in a linked list, then the list can be sorted with O(1) additional space. The algorithm starts with an initially empty (and therefore trivially sorted) list. The input items are taken off the list one at a time, and then inserted in the proper place in the sorted list. When the input list is empty, the sorted list has the desired result.

```
struct LIST * SortList1(struct LIST * pList) {
    // zero or one element in list
    if(pList == NULL || pList->pNext == NULL)
        return pList;
    // head is the first element of resulting sorted list
    struct LIST * head = 0;
    while(pList != NULL) {
        struct LIST * current = pList;
        pList = pList->pNext;
        if(head == NULL || current->iValue < head->iValue) {
            // insert into the head of the sorted list
            // or as the first element into an empty sorted list
            current->pNext = head;
            head = current;
        } else {
            // insert current element into proper position in non-empty
 sorted list
            struct LIST * p = head;
            while(p != NULL) {
                if(p->pNext == NULL || // last element of the sorted
list
                   current->iValue < p->pNext->iValue) // middle of the list
                {
                    // insert into middle of the sorted list or as the
last element
                    current->pNext = p->pNext;
                    p->pNext = current;
                    break; // done
                }
                p = p->pNext;
            }
        }
    }
    return head;
}
```

The algorithm below uses a trailing pointer for the insertion into the sorted list. A simpler recursive method rebuilds the list each time (rather than splicing) and can use O(*n*) stack space.

```
struct LIST
{
  struct LIST * pNext;
  int           iValue;
};

struct LIST * SortList(struct LIST * pList)
{
  // zero or one element in list
  if(!pList || !pList->pNext)
      return pList;

  /* build up the sorted array from the empty list */
  struct LIST * pSorted = NULL;

  /* take items off the input list one by one until empty */
  while (pList != NULL)
  {
      /* remember the head */
      struct LIST *   pHead  = pList;
      /* trailing pointer for efficient splice */
      struct LIST ** ppTrail = &pSorted;

      /* pop head off list */
      pList = pList->pNext;

      /* splice head into sorted list at proper place */
      while (!(*ppTrail == NULL || pHead->iValue < (*ppTrail)->iValue)) /* does head
belong here? */
      {
          /* no - continue down the list */
          ppTrail = &(*ppTrail)->pNext;
      }

      pHead->pNext = *ppTrail;
      *ppTrail = pHead;
  }

  return pSorted;
}
```

References

[1] Robert Sedgewick, *Algorithms*, Addison-Wesley 1983 (chapter 8 p. 95)

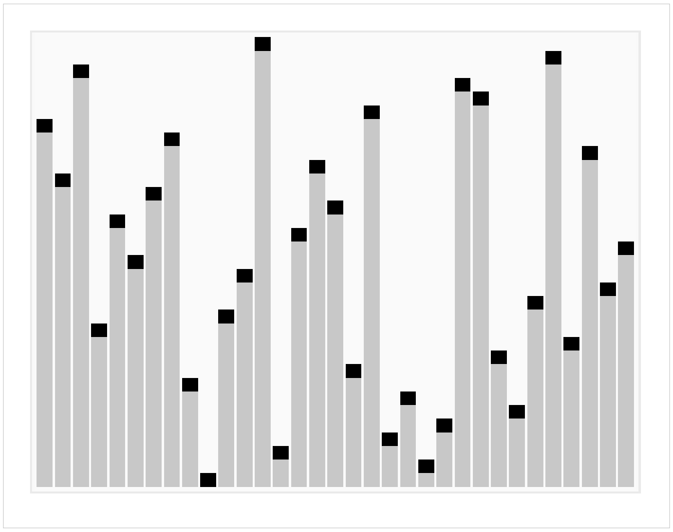
- Bender, Michael A; Farach-Colton, Martín; Mosteiro, Miguel (2006), *Insertion Sort is O(n log n)* (http://www.cs.sunysb.edu/~bender/newpub/BenderFaMo06-librarysort.pdf) (PDF), SUNYSB; also http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.3758; republished? in *Theory of Computing Systems* (ACM) **39** (3), June 2006 http://dl.acm.org/citation.cfm?id=1132705 (http://dl.acm.org/citation.cfm?id=1132705) `|url=` missing title (help) .
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), "2.1: Insertion sort", *Introduction to Algorithms* (second ed.), MIT Press and McGraw-Hill, pp. 15–21, ISBN 0-262-03293-7.
- "5.2.1: Sorting by Insertion", *The Art of Computer Programming*, 3. Sorting and Searching (second ed.), Addison-Wesley, 1998, pp. 80–105, ISBN 0-201-89685-0.
- Sedgewick, Robert (1983), "8", *Algorithms*, Addison-Wesley, pp. 95ff, ISBN 978-0-201-06672-2.

External links

- Adamovsky, John Paul, *Binary Insertion Sort – Scoreboard – Complete Investigation and C Implementation* (http://www.pathcom.com/~vadco/binary.html), Pathcom.
- *Insertion Sort in C with demo* (http://electrofriends.com/source-codes/software-programs/c/sorting-programs/program-to-sort-the-numbers-using-insertion-sort/), Electrofriends.
- *Insertion Sort – a comparison with other O(n^2) sorting algorithms* (http://corewar.co.uk/assembly/insertion.htm), UK: Core war.
- *Animated Sorting Algorithms: Insertion Sort – graphical demonstration and discussion of insertion sort* (http://www.sorting-algorithms.com/insertion-sort), Sorting algorithms.
- *Category:Insertion Sort* (http://literateprograms.org/Category:Insertion_sort) (wiki), LiteratePrograms – implementations of insertion sort in various programming languages
- *InsertionSort* (http://coderaptors.com/?InsertionSort), Code raptors – colored, graphical Java applet that allows experimentation with the initial input and provides statistics
- Harrison, *Sorting Algorithms Demo* (http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html), CA: UBC – visual demonstrations of sorting algorithms (implemented in Java)
- *Insertion sort* (http://www.algolist.net/Algorithms/Sorting/Insertion_sort) (illustrated explanation), Algo list. Java and C++ implementations.

Heapsort

Heapsort



A run of the heapsort algorithm sorting an array of randomly permuted values. In the first stage of the algorithm the array elements are reordered to satisfy the heap property. Before the actual sorting takes place, the heap tree structure is shown briefly for illustration.

Class	Sorting algorithm
Data structure	Array
Worst case performance	$O(n\ \log\ n)$
Best case performance	$\Omega(n), O(n\ \log\ n)$[1]
Average case performance	$O(n\ \log\ n)$
Worst case space complexity	$O(n)$ total, $O(1)$ auxiliary

Heapsort is a comparison-based sorting algorithm to create a sorted array (or list), and is part of the selection sort family. Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case O(*n* log *n*) runtime. Heapsort is an in-place algorithm, but it is not a stable sort. It was invented by J. W. J. Williams in 1964.

Overview

The heapsort algorithm can be divided into two parts.

In the first step, a heap is built out of the data.

In the second step, a sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. Once all objects have been removed from the heap, we have a sorted array. The direction of the sorted elements can be varied by choosing a min-heap or max-heap in step one.

Heapsort can be performed in place. The array can be split into two parts, the sorted array and the heap. The storage of heaps as arrays is diagrammed here. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.

Variations

- The most important variation to the simple variant is an improvement by R. W. Floyd that, in practice, gives about a 25% speed improvement by using only one comparison in each siftup run, which must be followed by a siftdown for the original child. Moreover, it is more elegant to formulate. Heapsort's natural way of indexing works on indices from 1 up to the number of items. Therefore the start address of the data should be shifted such that this logic can be implemented avoiding unnecessary +/- 1 offsets in the coded algorithm. The worst-case number of comparisons during the Floyd's heap-construction phase of Heapsort is known to be equal to $2N - 2s_2(N) - e_2(N)$, where $s_2(N)$ is the sum of all digits of the binary representation of N and $e_2(N)$ is the exponent of 2 in the prime factorization of N.
- Ternary heapsort[2] uses a ternary heap instead of a binary heap; that is, each element in the heap has three children. It is more complicated to program, but does a constant number of times fewer swap and comparison operations. This is because each step in the shift operation of a ternary heap requires three comparisons and one swap, whereas in a binary heap two comparisons and one swap are required. The ternary heap does two steps in less time than the binary heap requires for three steps, which multiplies the index by a factor of 9 instead of the factor 8 of three binary steps. Ternary heapsort is about 12% faster than the simple variant of binary heapsort.[*citation needed*]
- The **smoothsort** algorithm[3][4] is a variation of heapsort developed by Edsger Dijkstra in 1981. Like heapsort, smoothsort's upper bound is O(n log n). The advantage of smoothsort is that it comes closer to O(n) time if the input is already sorted to some degree, whereas heapsort averages O(n log n) regardless of the initial sorted state. Due to its complexity, smoothsort is rarely used.
- Levcopoulos and Petersson describe a variation of heapsort based on a Cartesian tree that does not add an element to the heap until smaller values on both sides of it have already been included in the sorted output. As they show, this modification can allow the algorithm to sort more quickly than O(n log n) for inputs that are already nearly sorted.

Comparison with other sorts

Heapsort primarily competes with quicksort, another very efficient general purpose nearly-in-place comparison-based sort algorithm.

Quicksort is typically somewhat faster due to some factors, but the worst-case running time for quicksort is O(n^2), which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk. See quicksort for a detailed discussion of this problem and possible solutions.

Thus, because of the O(n log n) upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort.

Heapsort also competes with merge sort, which has the same time bounds. Merge sort requires Ω(n) auxiliary space, but heapsort requires only a constant amount. Heapsort typically runs faster in practice on machines with small or slow data caches. On the other hand, merge sort has several advantages over heapsort:

- Merge sort on arrays has considerably better data cache performance, often outperforming heapsort on modern desktop computers because merge sort frequently accesses contiguous memory locations (good locality of reference); heapsort references are spread throughout the heap.
- Heapsort is not a stable sort; merge sort is stable.
- Merge sort parallelizes well and can achieve close to linear speedup with a trivial implementation; heapsort is not an obvious candidate for a parallel algorithm.
- Merge sort can be adapted to operate on linked lists with O(1) extra space.[5] Heapsort can be adapted to operate on doubly linked lists with only O(1) extra space overhead.[*citation needed*]

- Merge sort is used in external sorting; heapsort is not. Locality of reference is the issue.

Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

Pseudocode

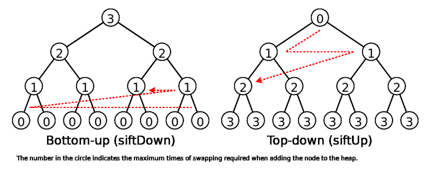
The following is the "simple" way to implement the algorithm in pseudocode. Arrays are **zero-based** and *swap* is used to exchange two elements of the array. Movement 'down' means from the root towards the leaves, or from lower indices to higher. Note that during the sort, the largest element is at the root of the heap at a[0], while at the end of the sort, the largest element is in a[end].

```
function heapSort(a, count) is
    input:  an unordered array a of length count

    (first place a in max-heap order)
    heapify(a, count)

    end := count-1 //in languages with zero-based arrays the children are 2*i+1 and 2*i+2
    while end > 0 do
        (swap the root(maximum value) of the heap with the last element of the heap)
        swap(a[end], a[0])
        (decrease the size of the heap by one so that the previous max value will
        stay in its proper placement)
        end := end - 1
        (put the heap back in max-heap order)
        siftDown(a, 0, end)

function heapify(a, count) is
    (start is assigned the index in a of the last parent node)
    start := (count - 2 ) / 2

    while start ≥ 0 do
        (sift down the node at index start to the proper place such that all nodes below
         the start index are in heap order)
        siftDown(a, start, count-1)
        start := start - 1
    (after sifting down the root all nodes/elements are in heap order)

function siftDown(a, start, end) is
    input:  end represents the limit of how far down the heap
                  to sift.
    root := start

    while root * 2 + 1 ≤ end do          (While the root has at least one child)
        child := root * 2 + 1        (root*2 + 1 points to the left child)
        swap := root        (keeps track of child to swap with)
        (check if root is smaller than left child)
        if a[swap] < a[child]
```

```
            swap := child
        (check if right child exists, and if it's bigger than what we're currently swapping with)
        if child+1 ≤ end and a[swap] < a[child+1]
            swap := child + 1
        (check if we need to swap at all)
        if swap != root
            swap(a[root], a[swap])
            root := swap          (repeat to continue sifting down the child now)
        else
            return
```

The heapify function can be thought of as building a heap from the bottom up, successively shifting downward to establish the heap property. An alternative version (shown below) that builds the heap top-down and sifts upward may be conceptually simpler to grasp. This "siftUp" version can be visualized as starting with an empty heap and successively inserting elements, whereas the "siftDown" version given above treats the entire input array as a full, "broken" heap and "repairs" it starting from the last non-trivial sub-heap (that is, the last parent node).

Also, the "siftDown" version of heapify has $O(n)$ time complexity, while the "siftUp" version given below has $O(n \log n)$ time complexity due to its equivalence with inserting each element, one at a time, into an empty heap. This may seem counter-intuitive since, at a glance, it is apparent that the former only makes half as many calls to its logarithmic-time sifting function as the latter; i.e., they seem to differ only by a constant factor, which never has an impact on asymptotic analysis.


Bottom-up (siftDown)
Top-down (siftUp)
The number in the circle indicates the maximum times of swapping required when adding the node to the heap.
Difference in time complexity between the "siftDown" version and the "siftUp" version.

To grasp the intuition behind this difference in complexity, note that the number of swaps that may occur during any one siftUp call *increases* with the depth of the node on which the call is made. The crux is that there are many (exponentially many) more "deep" nodes than there are "shallow" nodes in a heap, so that siftUp may have its full logarithmic running-time on the approximately linear number of calls made on the nodes at or near the "bottom" of the heap. On the other hand, the number of swaps that may occur during any one siftDown call *decreases* as the depth of the node on which the call is made increases. Thus, when the "siftDown" heapify begins and is calling siftDown on the bottom and most numerous node-layers, each sifting call will incur, at most, a number of swaps equal to the "height" (from the bottom of the heap) of the node on which the sifting call is made. In other words, about half the calls to siftDown will have at most only one swap, then about a quarter of the calls will have at most two swaps, etc.

The heapsort algorithm itself has $O(n \log n)$ time complexity using either version of heapify.

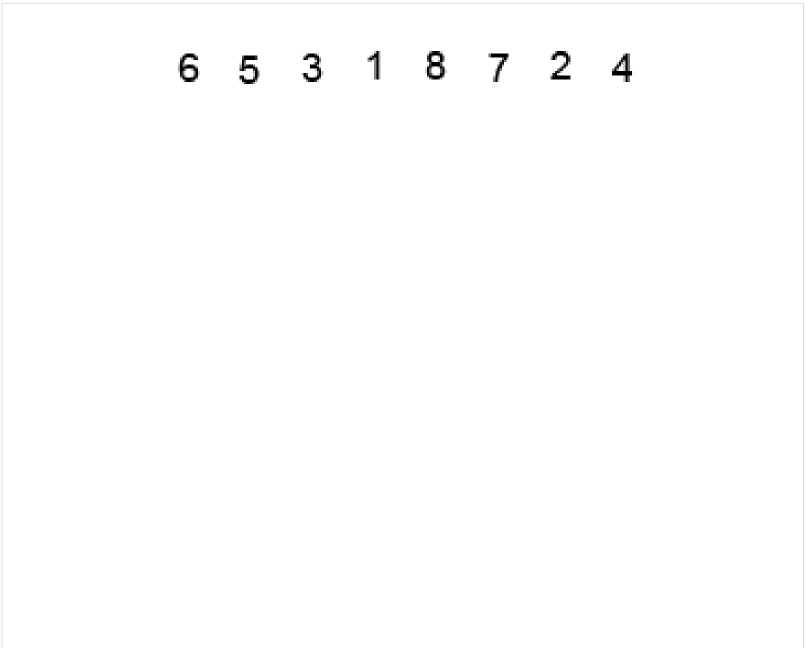
```
function heapify(a,count) is
    (end is assigned the index of the first (left) child of the root)
    end := 1

    while end < count
        (sift up the node at index end to the proper place such that all nodes above
         the end index are in heap order)
        siftUp(a, 0, end)
        end := end + 1
    (after sifting up the last node all nodes are in heap order)
```

```
function siftUp(a, start, end) is
    input:  start represents the limit of how far up the heap to sift.
                  end is the node to sift up.
    child := end
    while child > start
        parent := floor((child - 1) / 2)
        if a[parent] < a[child] then (out of max-heap order)
            swap(a[parent], a[child])
            child := parent (repeat to continue sifting up the parent now)
        else
            return
```

Example

Let { 6, 5, 3, 1, 8, 7, 2, 4 } be the list that we want to sort from the smallest to the largest. (NOTE, for 'Building the Heap' step: Larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap binary tree.)

1. Build the heap



An example on heapsort.

Heap	newly added element	swap elements
nil	6	
6	5	
6, 5	3	
6, 5, 3	1	
6, 5, 3, 1	8	
6, **5**, 3, 1, **8**		5, 8
6, **8**, 3, 1, 5		6, 8
8, 6, 3, 1, 5	7	
8, 6, **3**, 1, 5, **7**		3, 7

8, 6, 7, 1, 5, 3	2	
8, 6, 7, 1, 5, 3, 2	4	
8, 6, 7, **1**, 5, 3, 2, **4**		1, 4
8, 6, 7, 4, 5, 3, 2, 1		

2. Sorting.

Heap	swap elements	delete element	sorted array	details
8, 6, 7, 4, 5, 3, 2, **1**	8, 1			swap 8 and 1 in order to delete 8 from heap
1, 6, 7, 4, 5, 3, 2, **8**		8		delete 8 from heap and add to sorted array
1, 6, **7**, 4, 5, 3, 2	1, 7		8	swap 1 and 7 as they are not in order in the heap
7, 6, **1**, 4, 5, **3**, 2	1, 3		8	swap 1 and 3 as they are not in order in the heap
7, 6, 3, 4, 5, 1, **2**	7, 2		8	swap 7 and 2 in order to delete 7 from heap
2, 6, 3, 4, 5, 1, **7**		7	8	delete 7 from heap and add to sorted array
2, **6**, 3, 4, 5, 1	2, 6		7, 8	swap 2 and 6 as they are not in order in the heap
6, **2**, 3, 4, **5**, 1	2, 5		7, 8	swap 2 and 5 as they are not in order in the heap
6, 5, 3, 4, 2, **1**	6, 1		7, 8	swap 6 and 1 in order to delete 6 from heap
1, 5, 3, 4, 2, **6**		6	7, 8	delete 6 from heap and add to sorted array
1, **5**, 3, 4, 2	1, 5		6, 7, 8	swap 1 and 5 as they are not in order in the heap
5, **1**, 3, **4**, 2	1, 4		6, 7, 8	swap 1 and 4 as they are not in order in the heap
5, 4, 3, 1, **2**	5, 2		6, 7, 8	swap 5 and 2 in order to delete 5 from heap
2, 4, 3, 1, **5**		5	6, 7, 8	delete 5 from heap and add to sorted array
2, **4**, 3, 1	2, 4		5, 6, 7, 8	swap 2 and 4 as they are not in order in the heap
4, 2, 3, **1**	4, 1		5, 6, 7, 8	swap 4 and 1 in order to delete 4 from heap
1, 2, 3, **4**		4	5, 6, 7, 8	delete 4 from heap and add to sorted array
1, 2, **3**	1, 3		4, 5, 6, 7, 8	swap 1 and 3 as they are not in order in the heap
3, 2, **1**	3, 1		4, 5, 6, 7, 8	swap 3 and 1 in order to delete 3 from heap
1, 2, **3**		3	4, 5, 6, 7, 8	delete 3 from heap and add to sorted array
1, **2**	1, 2		3, 4, 5, 6, 7, 8	swap 1 and 2 as they are not in order in the heap
2, **1**	2, 1		3, 4, 5, 6, 7, 8	swap 2 and 1 in order to delete 2 from heap
1, **2**		2	3, 4, 5, 6, 7, 8	delete 2 from heap and add to sorted array
1		1	2, 3, 4, 5, 6, 7, 8	delete 1 from heap and add to sorted array
			1, 2, 3, 4, 5, 6, 7, 8	completed

Notes

[1] http://dx.doi.org/10.1006/jagm.1993.1031

[2] "Data Structures Using Pascal", 1991, page 405, gives a ternary heapsort as a student exercise. "Write a sorting routine similar to the heapsort except that it uses a ternary heap."

[3] http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF

[4] http://www.cs.utexas.edu/~EWD/transcriptions/EWD07xx/EWD796a.html

[5] Merge sort Wikipedia page

References

- Williams, J. W. J. (1964), "Algorithm 232 - Heapsort", *Communications of the ACM* **7** (6): 347–348
- Floyd, Robert W. (1964), "Algorithm 245 - Treesort 3", *Communications of the ACM* **7** (12): 701
- Carlsson, Svante (1987), "Average-case results on heapsort", *BIT* **27** (1): 2–17
- Knuth, Donald (1997), "§5.2.3, Sorting by Selection", *Sorting and Searching*, The Art of Computer Programming **3** (third ed.), Addison-Wesley, pp. 144–155, ISBN 0-201-89685-0
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapters 6 and 7 Respectively: Heapsort and Priority Queues
- A PDF of Dijkstra's original paper on Smoothsort (http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF)
- Heaps and Heapsort Tutorial (http://cis.stvincent.edu/html/tutorials/swd/heaps/heaps.html) by David Carlson, St. Vincent College
- Heaps of Knowledge (http://www.nicollet.net/2009/01/heaps-of-knowledge/)

External links

- Animated Sorting Algorithms: Heap Sort (http://www.sorting-algorithms.com/heap-sort) – graphical demonstration and discussion of heap sort
- Courseware on Heapsort from Univ. Oldenburg (http://olli.informatik.uni-oldenburg.de/heapsort_SALA/english/start.html) - With text, animations and interactive exercises
- NIST's Dictionary of Algorithms and Data Structures: Heapsort (http://www.nist.gov/dads/HTML/heapSort.html)
- Heapsort implemented in 12 languages (http://www.codecodex.com/wiki/Heapsort)
- Sorting revisited (http://www.azillionmonkeys.com/qed/sort.html) by Paul Hsieh
- A color graphical Java applet (http://coderaptors.com/?HeapSort) that allows experimentation with initial state and shows statistics
- A PowerPoint presentation demonstrating how Heap sort works (http://employees.oneonta.edu/zhangs/powerPointPlatform/index.php) that is for educators.
- Open Data Structures - Section 11.1.3 - Heap-Sort (http://opendatastructures.org/versions/edition-0.1e/ods-java/11_1_Comparison_Based_Sorti.html#SECTION001413000000000000000)

Article Sources and Contributors

Search algorithm *Source*: https://en.wikipedia.org/w/index.php?oldid=544783109 *Contributors*: Abutorsam007, Adityapasalapudi, Alex.g, Alexius08, Alexmorter, Altenmann, Andromeda, AngleWyrm, AnotherPerson 2, Barneca, Baruneju, Bbik, Beland, Bogdangiusca, Boleslav Bobcik, Bonadea, Bsilverthorn, CRGreathouse, CYD, Capricorn42, CharlesGillingham, Chris G, Christian75, ClansOfIntrigue, Colin Barrett, Conversion script, Cspooner, Dariopy, Daveagp, Dcoetzee, Decrypt3, DevastatorIIC, Diego Moya, Doc glasgow, Famtaro, Flandidlydanders, Fmicaeli, Frikle, GPHemsley, Gattom, Gene.arboit, Giftlite, Gregman2, HamburgerRadio, JNW, Jitse Niesen, Jorge Stolfi, Jschnur, Kareekacha, Kc03, Kdakin, Kenyon, Kgeza7, Kku, Kragen, Lamdk, Ligulem, MEOGLOBAL, Materialscientist, MattGiuca, Meredyth, Mezzanine, Mikeblas, Mild Bill Hiccup, Mironearth, Mlpkr, Mundhenk, Nabla, Nanshu, Neil Smithline, Nixdorf, Nohat, Ohnoitsjamie, Pgr94, Phils, Plasticup, Quadrescence, R'n'B, RW Marloe, Ramzysamman, Rgoodermote, RichF, Richardj311, RichiH, Robinh, Ruud Koot, SLi, Saaska, Sam Hocevar, Shuroo, SiobhanHansa, Snowolf, Spidern, Stefano, Taral, Taw, Tcncv, The12game, ThomHImself, Tide rolls, Tijfo098, Tomaxer, Vassloff, Ww, Xijiahe, Zzedar, 159 anonymous edits

Linear search *Source*: https://en.wikipedia.org/w/index.php?oldid=580523191 *Contributors*: Akhan.iipsmca, Alexander, Altay8, Andreas Kaufmann, Bporopat, CanisRufus, Conversion script, Dcoetzee, Download, Dze27, Ecb29, Eleschinski2000, Escape Orbit, Fredrik, Ghettoblaster, Giftlite, Glenn, Glrx, GregorB, Hirzel, Incnis Mrsi, Isis, Jdh30, Jogloran, Jorge Stolfi, M.O.X, Modster, Nanshu, NickyMcLean, Ninenines, Nixdorf, Nuno Tavares, Oleg Alexandrov, Oxaric, Patrick, Petrb, Pinethicket, RedWolf, Reedy, Richardj311, SERINE, SLi, Shuroo, Simeon, Stevietheman, StewartMH, Userabc, Utcursch, Xeno8, Ybungalobill, Zundark, 84 anonymous edits

Binary search algorithm *Source*: https://en.wikipedia.org/w/index.php?oldid=586913913 *Contributors*: 0, 1exec1, AANaimi, Agcala, Aj8uppal, Alansohn, Alex.koturanov, Allstarecho, Altenmann, Andrewrp, Ardnew, Arjarj, Atlantia, Balabiot, Baruneju, Beetstra, Beland, BenFrantzDale, BiT, BigDwiki, Biker Biker, Bill william compton, Black Falcon, Bloghog23, Bob1312, Boemanneke, Briandamgaard, Brianga, BrokenSegue, Caesura, CanOfWorms, CarminPolitano, ChaosCon, Charles Matthews, ChrisGualtieri, Chutzpan, CiaPan, Codethinkers, Coemgenus, ColdFusion650, Comp123, Curb Chain, DOwenWilliams, Daiyuda, Dasboe, David Cooke, David Eppstein, Dcoetzee, DevilsAdvocate, Devourer09, Df747jet, Dillard421, Diomidis Spinellis, Dodno, Doug Bell, Drtom, Dze27, ESkog, Ed Poor, EdH, Edward, El C, EmilJ, Ericamick, Ericl234, Ewx, Fabian Steeg, FactoidCow, Flyer22, Forderud, Fredrik, Fsiler, Fuzheado, Garyzx, Gene Thomas, Gerbrant, Giftlite, Gilderien, Glrx, Googl, Gpvos, Haigee2007, Hannes Hirzel, Hariva, Harriv, Harry0xBd, Hasanadnantaha, Hashar, Heineman, HenryLi, Hkleinnl, Htmnssn, Hv, ICrann15, IRockStone, Iain.dalton, Ieopo, Imjooseo, Ironmagma, J4 james, JLaTondre, Jamesx12345, Jan Winnicki, Jeffrey Mall, Jerryobject, Jfmantis, Jim1138, Jk2q3jrklse, Jonny Diamond, Joshgilkerson, JustAHappyCamper, Justin W Smith, Kcrca, Kdakin, Kinkydarkbird, Kylemcinnes, L Kensington, LA2, Lavaka, Liao, Ligulem, Loisel, Lourakis, Lugia2453, M, Macrakis, Macy, MarioS, Mariolj, Mark Martinec, Materialscientist, Maximaximax, Mboverload, McKay, MelbourneStar, Melmann, Merritt.alex, Messy Thinking, Mitch Ames, Mlpkr, Mononomic, MoreNet, Mr flea, Msswp, Muffincorp, Musiphil, Mxn, Neuralwarp, NickyMcLean, Ninaddb, Nithin.A.P, Nixdorf, Njanani, Nnarasimhakaushik, Nullzero, Nuno Tavares, Oli Filth, OverlordQ, Oylenshpeegul, Pakaran, Pappu0007, Patrick, Peter Karlsen, Peter Winnberg, Phil Boswell, Photonique, Plugwash, Pm215, Pmlineditor, Pne, Pnm, Pol098, Predator106, Psherm85, Quasipalm, Quuxplusone, R.e.b., Ranching, Robert Dober, Robrohan, Rocketrod1960, Rodion Gork, Ryajinor, Rynishere, SPTWriter, Sangameshh, Scandum, Scarian, Seaphoto, Sephiroth BCR, Shirik, SigmaEpsilon, SiobhanHansa, Sioux.cz, SirJective, SoCalSuperEagle, Solidpoint, Spoon!, Stan Shebs, Stephenb, Svick, Svivian, Swanyboy2, TYelliot, Tabletop, Taeshadow, TakuyaMurata, Taw, Tentinator, TestPilot, The Cave Troll, Tim32, TripleF, Trusilver, Two Bananas, Ukexpat, Userabc, Verdy p, Vipinhari, Wavelength, Wbm1058, WhiteOak2006, Widr, Wikipelli, WilliamThweatt, Wstomv, Wullschj, XP1, Zachwlewis, Zeno Gantner, Zzedar, 555 anonymous edits

Sorting algorithm *Source*: https://en.wikipedia.org/w/index.php?oldid=585943435 *Contributors*: -OOPSIE-, 124Nick, 132.204.27.xxx, A3 nm, A5b, A930913, AManWithNoPlan, Aaron Rotenberg, Abhishekupadhya, Accelerometer, Adair2324, AdamProcter, Advance512, Aeons, Aeonx, Aeriform, Agateller, Agorf, Aguydude, Ahoerstemeier, Ahshabazz, Ahy1, Alain Amiouni, Alansohn, AlexPlank, Alksub, AllyUnion, Altenmann, Alvestrand, Amirmalekzadeh, Anadverb, Andre Engels, Andy M. Wang, Ang3lboy2001, Angela, Arpi0292, Artoonie, Arvindn, Astronouth7303, AxelBoldt, Ayushyogi, BACbKA, Bachrach44, Balabiot, Baltar, Gaius, Bartoron2, Bbi5291, Beland, Ben Standeven, BenFrantzDale, BenKovitz, Bender2k14, Bento00, Bidabadi, Bkell, Bobo192, Boleyn, Booyabazooka, Bradyoung01, Brendanl79, Bryan Derksen, BryghtShadow, Bubba73, BurtAlert, C. A. Russell, C7protal, CJLL Wright, Caesura, Calculuslover, Calixte, CambridgeBayWeather, Canthusus, Carey Evans, Ccn, Charles Matthews, Chenopodiaceous, Chinju, Chris the speller, Ciaccona, Circular17, ClockworkSoul, Codeman38, Cole Kitchen, Compfreak7, Conversion script, Cpl Syx, Crashmatrix, Crumpuppet, Cuberoot31, Cwolfsheep, Cyan, Cybercobra, Cymbalta, Cyrius, DHN, DIY, DaVinci, Daiyuda, Damian Yerrick, Danakil, Daniel Quinlan, DarkFalls, Darkwind, DarrylNester, Darth Panda, David Eppstein, Dcirovic, Dcoetzee, Deanonwiki, Debackerl, Decrypt3, Deepakjoy, Deskana, DevastatorIIC, Dgse87, Diannaa, Dihard, Domingos, Doradus, Duck1123, Duvavic1, Dybdahl, Dysprosia, EdC, Eddideigel, Efansoftware, Egomes jason, Eliz81, Energy Dome, Etopocketo, Fagstein, Falcon8765, Fastily, Fawcett5, Firsfron, Foobarnix, Foot, Fragglet, Frankenviking, Fred Bauder, Fredrik, Frencheigh, Fresheneesz, Fuzzy, GanKeyu, GateKeeper, Gavia immer, Gdr, Giftlite, Glrx, Grafen, Graham87, Graue, GregorB, H3nry, HJ Mitchell, Hadal, Hagerman, Hairhorn, Hamaad.s, Hannes Hirzel, Hashar, Hede2000, Hgranqvist, Hirzel, Hobart, HolyCookie, Hpa, I am One of Many, I dream of horses, IMalc, Indefual, InverseHypercube, Iridescent, Itsameen-bc103112, J.delanoy, JBakaka, JLaTondre, JRSpriggs, JTN, Jacektomas, Jachto, Jaguaraci, Jamesday, Japo, Jay Litman, Jbonneau, Jeffq, Jeffrey Mall, Jeronimo, Jesin, Jirka6, Jj137, Jll, Jmw02824, Jokes Free4Me, JonGinny, Jonadab, Jonas Kölker, Josh Kehn, Joshk, Jprg1966, Jthemphill, Justin W Smith, Jwoodger, Kalraritz, Kevinsystrom, Khazar2, Kievite, Kingjames iv, KlappCK, Knivd, Knutux, Kragen, KyuubiSeal, LC, Ldoron, Lee J Haywood, LilHelpa, Lowercase Sigma, Luna Santin, Lzap, Makeemlighter, Malcolm Farmer, Mandarax, Mark Renier, MarkisLandis, MartinHarper, Marvon7Newby, MarvonNewby, Mas.morozov, Materialscientist, Mathmensch, MattGiuca, Matthew0028, Mav, Maximus Rex, Mbernard707, Mdd4696, Mdtr, Medich1985, Methecooldude, Michael Greiner, Michael Hardy, Michaelbluejay, Mike Rosoft, Mike00500, Mina86, Mindmatrix, Mountain, Mr Elmo, Mrck@charter.net, Mrjeff, Mudd1, Musiphil, Myanw, NTF, Nanshu, Nayuki, Nbarth, Nevakee11, NewEnglandYankee, NickT988, Nicolaum, Nikai, Nish0009, Nixdorf, Nknight, Nomen4Omen, NostinAdrek, OfekRon, Olathe, Olivier, Omegatron, Ondra.pelech, OoS, Oskar Sigvardsson, Oğuz Ergin, Pablo.cl, Pajz, Pamulapati, Panarchy, Panu-Kristian Poiksalo, PatPeter, Patrick, Paul Murray, Pbassan, Pcap, Pce3@ij.net, Pelister, Perstar, Pete142, Peter Flass, Petri Krohn, Pfalstad, Philomathoholic, PierreBoudes, Piet Delport, Populus, Pparent, PsyberS, Pyfan, Quaeler, RHaworth, RJFJR, RapidR, Rasinj, Raul654, RaulMetumtam, RazorICE, Rcbarnes, Rcgldr, Reyk, Riana, Roadrunner, Robert L, Robin S, RobinK, Rodspade, Roman V. Odaisky, Rsathish, Rursus, Ruud Koot, Ryguasu, Scalene, Schnozzinkobenstein, Shadowjams, Shanes, Shredwheat, SimonP, SiobhanHansa, Sir Nicholas de Mimsy-Porpington, Slashme, Sligocki, Smartech, Smjg, Snickel11, Sophus Bie, Soultaco, SouthernNights, Spoon!, Ssd, StanfordProgrammer, Staplesauce, Starwiz, Staszek Lem, Stephen Howe, Stephenb, StewieK, Suanshsinghal, Summentier, Sven nestle2, Swamp Ig, Swift, T4bits, TBingmann, TakuyaMurata, Tamfang, Taw, Tawker, Techie007, Teles, Templatetypedef, The Anome, The Thing That Should Not Be, TheKMan, TheRingess, Thefourlinestar, Thinking of England, ThomasTomMueller, Thumperward, Timwi, Titodutta, Tobias Bergemann, Tortoise 74, TowerDragon, Travuun, Trixter, Twikir, Tyler McHenry, UTSRelativity, Udirock, Ulfben, UpstateNYer, User A1, VTBassMatt, Vacation9, Valenciano, Veganfanatic, VeryVerily, Veryangrypenguin, Verycuriousboy, Vrenator, Wantnot, Wazimuko, Wei.cs, Wfunction, Wiki.ryansmith, Wikiwonky, WillNess, Wimt, Worch, Writtenonsand, Ww, Yansa, Yintan, Yuval madar, Zawersh, Zipcodeman, Ztothefifth, Zundark, 945 anonymous edits

Bubble sort *Source*: https://en.wikipedia.org/w/index.php?oldid=587127194 *Contributors*: 4wajzkd02, A bit iffy, Abu adam, Adam Zivner, Adamuu, AdmN, Adrian.hawryluk, Agorf, Ahy1, Akhan.iipsmca, Alansohn, Aleksa Lukic, Alexius08, Alfie66, Allen3, Ambassador1, Amrish deep, Andre Engels, Andrejj, Andres, Anti-Nationalist, Arvindn, AxelBoldt, Balabiot, Beetstra, Billaaa, Bk314159, Black Falcon, Blaisorblade, Booyabazooka, Borice, Bpapa2, Breawycker, BrianWilloughby, BrokenSegue, Bubble snipe, Buddhikaeport, CJLL Wright, Cacophony, Calabe1992, Carey Evans, Centrx, Ceros, Charles yiming, Christian75, Chuckmasterrhymes, Cl2ha, Cloudrunner, Clx321, Conversion script, Copysan, CountingPine, Crashmatrix, Cwolfsheep, Cybercobra, DBigXray, DVdm, Daisymoobeer, Daniel Brockman, Daniel5Ko, Dantheox, David Eppstein, Dcoetzee, Deconvolution, Delog, DevastatorIIC, Dfarrell07, DivideByZero14, Djh2400, Donner60, Drjan, Dudesleeper, E2eamon, EEMIV, ESkog, Ebehn, Edward, Eequor, Encyclopediamonitor, Erikthomp, Everard Proudfoot, Foobar, Fraggle81, Fredrik, FvdP, GLmathgrant, Garas, Gargaj, Garyzx, Gdavidp, Gerweck, Giftlite, Ginsuloft, Glrx, Grendelkhan, Gruauder, Gsantor, Hannan1212, Happypal, Hari, Hashar, Hate123veteran, Hefo, Hertz1888, I dream of horses, INkubusse, IanOsgood, Imjustin, Intel4004, InverseHypercube, Isis, J.delanoy, Jafet, Jdaudier, Jdb67usa, Jellyworld, Jeltz, Jerk111, Jeronimo, Jevy1234, Jim1138, Jmallios, Jmartinezot, Jni, John lilburne, Josecgomez, Josh Kehn, Jossi, Justin W Smith, Kaare, Kan8eDie, Kizor, Klrste, Knutux, Kragen, Krithin, Kuru, L Kensington, LOL, Lailsonbm, LarsMarius, Lee Daniel Crocker, Liao, Lioinnisfree, Looxix, Louis Kyu Won Ryu, Madanpiyush, Mahanthi1, Mantipula, Marc van Leeuwen, Mark viking, Materialscientist, Mattbash, Mediran, Melonkelon, Merovingian, Michael Hardy, Minna Sora no Shita, Miscode, Mizaoku, Moogwrench, Mortense, MrOllie, Mshonle, Nayuki, Nedaljo, NeilFraser, NellieBly, Newuserwiki, NickShaforostoff, Nixdorf, Nmnogueira, Noisylo65, Nuno Tavares, O.Koslowski, Octahedron80, Octavdruta, Oddity-, Officialhopsof, Ohnoitsjamie, Olaolaola, Oli Filth, Oskar Sigvardsson, Oxaric, Oğuz Ergin, Palit Suchitto, Panzi, Paranomia, PauliKL, Pedro, Pfalstad, Philip Trueman, Pinethicket, Prestonmag, Pshirishreddy, Psym, Pt, QFlyer, Quaestor, Quang thai, Quasipalm, Qxz, R. J. Mathar, RTBarnard, Reedy, Rhanekom, Richie, Ronhjones, Ronzii, Ruolin59, Ryk, SCriBu, SPTWriter, Saibo, Saifhhasan, Sam Hocevar, Sandyjee, Santhoshreddym, Satya61229, Shearsongs78, Shreevatsa, Silly rabbit, Simeon, Simpsons contributor, SiobhanHansa, Sjakkalle, Skamecrazy123, Sligocki, Snickel11, Spammyammy, Spiff, Stebbins, Stephenb, StewieK, Super-Magician, Surfer43, Swfung8, Swift, THEN WHO WAS PHONE?, Tentinator, Themania, Tide rolls, Timwi, Tobias Bergemann, Tolly4bolly, Tom714uk, TommyG, Toxin1000, TroncTest, Trunks175, Two Bananas, Udirock, UncleDouggie, Uranographer, Userabc, Vdaghan, Vexis, Virus326, Voyevoda, VzjrZ, Waldir, Wangbing7928, Washa rednecks, Wayne Slam, Who then was a gentleman?, WikHead, WormNut, Ww, XP1, Xanchester, Ycl6, Yuval madar, Zaphod Beeblebrox, ZeroOne, Ztothefifth, 693 anonymous edits

Quicksort *Source*: https://en.wikipedia.org/w/index.php?oldid=587222685 *Contributors*: 0, 1exec1, 4v4l0n42, A Meteorite, Aaditya 7, Aaronbrick, Abednigo, Abu adam, Adicarlo, AdmN, Aerion, Ahy1, Akaloger, Alain Amiouni, AlanBarrett, Alcauchy, Alexander256, Allan McInnes, Altenmann, Andrei Stroe, Apoorva181192, Arcturus4669, Aroben, Arto B, Arvindn, AxelBoldt, BarretB, Bartosz, Bastetswarrior, Bayard, Beetstra, Benbread, Bengski68, Berlinguyinca, Bertrc, Biker Biker, Bkell, Black Falcon, Blaisorblade, Bluear, Bluemoose, Booyabazooka, Boulevardier, Bputman, BrokenSegue, Btwied, Bubba73, C. lorenz, C7protal, CJLL Wright, Calwiki, Carlj7, CarlosHoyos, Cdiggins, Cedar101, Centrx, CesarB, Chameleon, Chinju, Chrischan, Chrisdone, Chrislk02, CiaPan, Cmcfarland, CobaltBlue, ColdShine, Composingliger, Compotatoj, Comrade009, Connelly, Croc hunter, Crowst, Cumthsc, Cybercobra, Cyde, Czarkoff, Daekharel, Dan100, DanBishop, Danakil, Dancraggs, Daniel5Ko, Darrel francis, Darren Strash, David Bunde, Dbfirs, Dcoetzee, Decrypt3, Diego UFCG, Dila, Dinomite, Diomidis Spinellis, Discospinster, Dissident, Dmccarty, Dmcq, Dmwpowers, Doccolinni, DomQ, Domokato, Donhalcon, Doradus, Dr.RMills, Dysprosia, ESkog, Eequor, Elcielo917, Eleassar, Elharo, Empaisley, Entropy, Eric119, Ettrig, Evercat, Evil saltine, Faridani, Fastilysock, Fennec, Feradz, Ferkelparade, Forderud, Francis Tyers, Frankrod44, Frantisek.jandos, François Robere, Fredrik, Fresheneesz, Fryed-peach, Func, Furrfu, Fx2, Fxer, Gdo01, Giftlite, Glane23, Glrx, Gpollock, Graham87, GregorB, Gscshoyru, HJ Mitchell, Haker4o, HalHal, Hamza1886, Hannes Hirzel, Hdante, Helios2k6, Hfastedge, Indeed123, Intangir, Integr8e, Ipeirotis, Jadrian, Jamesday, Jao, Jason Davies, Jazmatician, Jeff3000, Jevy1234, Jfmantis, Jnoring, John Comeau, John lindgren, John of Reading, Jokes Free4Me, Josh Kehn, Jpbowen, Juliand, Jusjih, JustAHappyCamper, Jóna Þórunn, Kanie, Kapildalwani, Karnan, Kbk, Kbolino, Kingturtle, KittyKAY4, Knutux, Kragen, Krishna553, Kuszi, LOL, Larosek, Lars Trebing, Lhuang3, Liftarn, LilHelpa, LittleDan, LodeRunner, Lord Emsworth, Lordmetroid, Lugia2453, MC10, MH, Magioladitis, Magister Mathematicae, Maju wiki, Mark L MacDonald, Mathiastck, McKay, Mecej4, Meekohi, Mh26, Michael Shields, Mihai Damian, Mike Rosoft, Mikeblas, Modster, MrOllie, Mswen, Murray Langton, N Vale, NTF, Narendrak, Naveenkodali123, NawlinWiki, Nbarth, Nearffxx, Necklace, Neilc, Neohaven, NevilleDNZ, NickW557, NickyMcLean, Nik 0 0, Ninjatummen, Nixdorf, Nmnogueira, Nneonneo, Noisylo65, Nonsanity, Norm mit, Notheruser, Nothing1212, NovellGuy, Nwerneck, Obscuranym, Ocolon, Ohnoitsjamie, Oli Filth, Oskar Sigvardsson, OverlordQ, Owen214, Oğuz Ergin, PGSONIC, Pakaran, Pako, Panzi, Patmorin, PatrickFisher, Perpetuo2, Pete142, Petrb, Pgan002, Phe, Philip Trueman, Pifactorial, Pmcjones, Populus, Postrach, Pushingbits, Quantumelixir, Quarl, Quuxplusone, Qwertyus, R3m0t, Rami R, Randywombat, Raul654, Rdargent, RedWolf, Rhanekom, Richss, Roger Hui, RolandH, Romanm, Rony fhebrian, Rrufai, Rsanchezsaez, Rspeer, Runefurb, Rursus, Ruud Koot, SKATEMANKING, SPTWriter, Sabb0ur, Sam Hocevar, SamuelRiv, Sandare, Scebert, Scott Paeth, Seaphoto, Senfo, Sf222, Shanes, Shashmik11, Shellreef, Shentino, Simetrical, SiobhanHansa, Sir Edward V, Sir Nicholas de Mimsy-Porpington, Sjakkalle, Skapur, Sligocki, Snow Blizzard, Snowolf, Solde9, Soliloquial, Spearhead, Spiff, Stdazi, StephenDow, Stevietheman, Sverdrup, Svick, Swfung8, Swift, Swifty slow, Sychen, TakuyaMurata, Tanadeau, Tavianator, The Anome, TheCois, Theclawen, Themania, ThomasTomMueller, Tide rolls, Tim32, Timneu22, Timwi, Tobias Bergemann, Tolly4bolly, Tompagenet, Too Old, Udirock, Ungzd, UrsaFoot, User A1, Vacation9, Versus22, Vikreykja, Vladkornea, Weedwhacker128, Wfaxon, Wik, Wikid77, Wisgary, Worch, Ww, Xezbeth, Yandman, Ylloh, Yulin11, Zarrandreas, Zeno Gantner, ZeroOne, Znupi, Михајло Анђелковић, 857 anonymous edits

Merge sort *Source*: https://en.wikipedia.org/w/index.php?oldid=586327576 *Contributors*: 192.58.206.xxx, 209.157.137.xxx, Ablonus, Abu adam, Acalamari, Adambro, Ahmadsh, Ahy1, Alain Amiouni, Alansohn, Alcat33, Allan McInnes, Amahdy, Amiodusz, Andrei Stroe, Andy M. Wang, Antientropic, Apoorbo, Apoorva181192, ArnoldReinhold, Artagnon, Avb, Bakanov, Base698, Bayard, Bender2k14, Bill wang1234, Black Falcon, Bobrayner, Booyabazooka, Bor75, BrokenSegue, Bubba73, C. A. Russell, CJLL Wright, CRGreathouse, Cactus.man, Caesura, Captain Fortran, Cedar101, Ceros, Chowbok, Cic, CobaltBlue, Comocomocomocomo, Conversion script, Ctxppc, Cuzelac, Cybercobra, DFRussia, DVdm, Damian Yerrick, Dammit, Damonkohler, Danakil, Daniel Geisler, Daniel Quinlan, Daztekk, Dbagnall, Dbingham, Dcljr, Dcoetzee, Deanonwiki, Decrypt3, Deepakabhyankar, Delldot, Destynova, DevastatorIIC, Dima1, Discospinster, Dmcq, Donhalcon, Dr. Gonzo, Duckbill, Duvavic1, EAspenwood, Easwarno1, EdSchouten, Eleschinski2000, EmadIV, Erel Segal, Eric119, Eserra, Ewlyahoocom, Excirial, Faridani, Fashnek, Fizo86, Forderud, Fraggle81, Fred J, Fredrik, Frencheigh, Furrykef, Fyyer, Garas, Garo, Garrettw87, Garyzx, Geeker87, Giftlite, Gilliam, Glrx, GraemeL, GregorB, Haham hanuka, Hariva, Hoof1341, HoserHead, Hyad, Immunize, Intgr, Itmozart, J.delanoy, JF Bastien, Jacobolus, Jafet, Jengelh, JerryFriedman, JimVC3, Jirka6, Jk2q3jrklse, Jleedev, Jogloran, JohnOwens, Josh Kehn, Joshk, Jpl, Kalebdf, Kaushik0402, Kbk, Kenb215, Kenyon, Klrste, Knutux, Kragen, Lee Daniel Crocker, Liko81, Mark viking, Mattjohnson, Mav, Mblumber, Mctpyt, Mecej4, Michele bon, Mikeblas, Mikewebkist, Minesweeper, Mipadi, Miskaton, MisterSheik, Mlhetland, Mntlchaos, MrOllie, Mutinus, N26ankur, Naderra, Naku, Nbarth, Negrulio, Neilc, Nguyen Thanh Quang, Nickls, Ninjatummen, Nixeagle, Nmnogueira, NotARusski, Novas0x2a, Nuno Tavares, Nyenyec, Octotron, Ohnoitsjamie, Oli Filth, Onco p53, OranL, Oskar Sigvardsson, Ossi, Oxaric, Oğuz Ergin, Patmorin, PavelY, Pfalstad, Pgan002, Pkrecker, Pnm, Quasipalm, Radiozilla, Ravishanker.bit, Rcgldr, Renku, ReyBrujo, Rfl, Rhanekom, Rich Farmbrough, Riemann'sZeta, Rjwilmsi, Rl, Romanm, Rspeer, Ruud Koot, SLi, Sam nead, Sanjay742, SashaMarievskaya, Schorzman78, Scott Paeth, ScottBurson, Shadowjams, Shashwat2691, Shellreef, SigbertW, Silly rabbit, SiobhanHansa, Sir Nicholas de Mimsy-Porpington, Sirex98, Sligocki, Soultaco, Sperling, Strcat, StuartBrady, Sven nestle2, Swfung8, Swift, Sychen, T0m, TBloemink, TakuyaMurata, The Anome, Thijswijs, ThomasMueller, Thue, Timeroot, Timwi, Tobias Bergemann, TobiasPersson, Tohd8BohaithuGh1, Tomchiukc, Tommunist, Towopedia, Tuttu4u, UncleDouggie, Vanished user 39948282, Vexis, Villaone56, VineetKumar, Vorn, Wbeek, Widr, Wilagobler, Wintonian, WojciechSwiderski, Worch, Ww, X7q, Xhackeranywhere, Xueshengyao, Yodamgod, Zhaladshar, Zophar1, 500 anonymous edits

Insertion sort *Source*: https://en.wikipedia.org/w/index.php?oldid=585830984 *Contributors*: 1qaz-pl, 24.252.226.xxx, Ahy1, Alansohn, Aleks80, Alex.atkins, Alex.mccarthy, Alexius08, Allan McInnes, Altenmann, Amarpalsinghkapoor, Amaury, Andre Engels, Andres, Apanag, Arthena, Arthur Rubin, Asdquefty, Autumnalmonk, AxelBoldt, Baby123412, Baltar, Gaius, Barras, Baruneju, BiT, Billinghurst, Billylikeswikis, BioPupil, Black Falcon, Blaisorblade, Bobo192, Booyabazooka, Brambleclawx, BrokenSegue, Buddhikaeport, CJLL Wright, Carey Evans, Cedar101, Ceros, Chet Gray, ChrisGualtieri, Colinb, Conversion script, Coshoi, Crashmatrix, Cyde, DFRussia, Damian Yerrick, Daniel Brockman, Daniel Quinlan, Dardasavta, Dastoger Bashar, David Eppstein, DavidGrayson, Davinci8x8, Dcoetzee, DevastatorIIC, Don4of4, Doradus, E.ruzi, EAtsala, Eequor, El C, Emdtechnology, Eric119, Faizbash, Fangyuan1st, Freakingtips, Fredrik, Frozenport, Garyzx, Giftlite, Glrx, Gmazeroff, Goutamrocks, GregorB, Groffles, H.ehsaan, HJ Mitchell, Hannes Hirzel, Hardmath, Hari, Harrisonmetz, HereToHelp, Hydrogen Iodide, I do not exist, Int19h, InverseHypercube, Ivan Pozdeev, Jarajapu, Jarble, Jashar, JerryFriedman, Jesin, Jfmantis, Jonas Kölker, Jordanbray, JosephMDecock, Josh Kehn, Jpmelos, KBKarma, Kangaroosrule, Karl Dickman, Keynell, Killiondude, Kiwi137, Klrste, Knuckles, Knutux, Kostmo, Kpjas, Kragen, Kri, LOL, Lawlzlawlz, Lidden, Looxix, Love debian, MC10, Magicbronson, Mahlon, Marvon7Newby, Materialscientist, Mblumber, Merendoglu, Mess, Michael Hardy, Michael Slone, Mike1242, Mike1341, Minchenko Michael, Mollmerx, MorganGreen, MrOllie, Nasradu8, Nayuki, Neel basu, Nillerdk, Ninjakannon, Nixdorf, Nmnogueira, Nordald, Nuggetboy, Nuxnut, Ohconfucius, Ohnoitsjamie, Okosenkov, Oli Filth, Oskar Sigvardsson, Oxaric, Oğuz Ergin, Pcp071098, Pdvyas, Pharaoh of the Wizards, Phil Boswell, PhilippWeissenbacher, Piet Delport, Pinball22, Pion, Pipedreambomb, Player 03, Ponggr, Pratyya Ghosh, Pred, PrimeHunter, Pulveriser, Quentonamos, RA0808, Rajarammallya, Ratheesh nan, Rax2095, Reidhoch, Rhanekom, Rlneumiller, Ruud Koot, SLi, Sapeur, Scandum, Seanhalle, Seizethedave, Sfan00 IMG, SheldonYoung, Silly rabbit, Simetrical, Simpsons contributor, SiobhanHansa, Sligocki, Smalljim, Smjg, SmokingCrop, Steven Zhang, Strcat, Svick, Swfung8, Swift, Tamer ih, Ted Longstaffe, ThomasMueller, Tiddly Tom, Timwi, Tjwood, Tobias Bergemann, Trivialist, Trunks175, Tryptophan4, VTBassMatt, Vasiľ, Villaone56, Virtualblackfox, Vpshastry, Wavelength, Werdna, Will Faught, WookieInHeat, Wtshymanski, Ww, Ww2censor, XP1, XreDuex, Yandman, Zaradaqaw, ZeroOne, Zowayix, Zvar, 434 anonymous edits

Heapsort *Source*: https://en.wikipedia.org/w/index.php?oldid=583961200 *Contributors*: 1exec1, 4wajzkd02, A.Ou, ANONYMOUS COWARD0xC0DE, Ahy1, Ajitk, Alain Amiouni, Alansohn, Alexius08, AlistairMcMillan, Apoorva181192, Asacarny, Avatarmonkeykirby, Aveeare, Awesomeness, Ayman, Barefootguru, Beland, Black Falcon, Boatsdesk, Booyabazooka, BrokenSegue, CJLL Wright, CRGreathouse, Carey Evans, Cataxxx, Cerasti, Chricho, Circular17, Ck lostsword, Cliff smith, Conversion script, Cwolfsheep, Cybercobra, Cynddl, Damian Yerrick, Danakil, Daniel Quinlan, David Eppstein, DavidGrayson, Dbenbenn, Dcoetzee, Delldot, DisasterManX, Dmcq, Don4of4, Dysprosia, Eequor, Eggman64, Elias, Evershade, Ffangs, Finem, Foobar, Fredrik, Garyzx, Giftlite, Glrx, GregorB, HJ Mitchell, HairyFotr, Hamaad.s, Hariva, Hdante, Herry1234, High-quality32, Hiiiiiiiiiiiiiiiiiiiii, ItsProgrammable, J04n, JRSpriggs, JerseyChewi, Jirka6, Jogloran, JohnOwens, Joncop, Josh Kehn, Joshi1983, Jrwhitehill, Juliancolton, Juliano, Justin W Smith, Jóna Þórunn, Karl Dickman, Kingpin13, Kipb9, KnightRider, Knutux, KuroiShiroi, LOL, Lamb99, Lark ascending, Laurens, Lauwr, Leberbaum, LiDaobing, Liao, Looxix, Luk, MC10, MONGO, Mahanga, Mahanthi1, Manish181192, Masamage, Masao, MattGiuca, Michael Ross, Michaelhilton, Mike1341, MisterSheik, Mona.tehrani, Mortense, MrOllie, NTF, Negrulio, Neilc, NubKnacker, Oaf2, Ohnoitsjamie, Oli Filth, Opelio, Oskar Sigvardsson, Parallelized, Patmorin, Paul Murray, Pdelong, Peristarkawan, Peruvianllama, Piperh, Pit, Postrach, Puckly, Quasipalm, Quaternion, Reedy, Rekinser, Revolus, RolandH, RomanZeyde, Rummelsworth, Ruud Koot, Ryanmeeru, Saintrain, Salvar, Scandum, Sculliam, Seaphoto, Serge9393, Shadowjams, SharShar, Shashmik11, Silly rabbit, SiobhanHansa, Sir Nicholas de Mimsy-Porpington, Smjg, Spencer, Starwiz, Stelian Dumitrascu, Stephen C. Carlson, Stephen Gilbert, StrawberryBlondy, Styner32, Svick, Swfung8, Swift, Taw, Teles, TheKMan, Timwi, Trevor Andersen, Tristanb, Updeshgarg, Uselesswarrior, V.Sindhuja, Vikrum, Wayward, Wheresthebrain, WillNess, Worch, Wsloand, Ww, Ycl6, Yparjis, ZeroOne, Zvar, Zzedar, Олександр Кравчук, یویا, 339 anonymous edits

Image Sources, Licenses and Contributors

License