

Sorting for Algorithm Design

option	applications
sorting algorithm	<ul style="list-style-type: none"> when all elements to sort are known at one time many tasks can reduce to sorting, or can be done more efficiently if the elements are sorted – “when in doubt, sort” e.g. searching, closest pair, element uniqueness, finding the mode, selection e.g. greedy algorithms such as event scheduling and Kruskal's algorithm
PriorityQueue	<ul style="list-style-type: none"> for sorting in a dynamic environment, where the elements are not all known at once e.g. greedy algorithms such as Dijkstra's algorithm and Prim's algorithm

Pragmatics –

- can handle increasing vs decreasing order, keys vs records, and non-numerical data by abstracting a *comparator* from the sorting algorithm

Sorting

Key algorithms –

- | | | |
|------------------|---|--|
| comparison-based | { | <ul style="list-style-type: none"> selection sort – $O(n^2)$ <ul style="list-style-type: none"> – heapsort – $O(n \log n)$ insertion sort – $O(n^2)$ <ul style="list-style-type: none"> – shellsort can be as good as $O(n \log^2 n)$ with the right gap sequence
<small>https://en.wikipedia.org/wiki/Shell_sort</small> |
| | | <ul style="list-style-type: none"> quicksort, merge sort – $O(n \log n)$ |
| distribution | { | <ul style="list-style-type: none"> bucket sort – average $O(n)$ when $b \approx n$, $O(n+n^2/b+b)$ otherwise worst $O(n^2)$ [or $O(n \log n)$ depending on algorithm used for buckets] <ul style="list-style-type: none"> – b is the number of buckets
<small>ADM 4.7 https://en.wikipedia.org/wiki/Bucket_sort</small> radix sort – $O(nd)$ <ul style="list-style-type: none"> – d is the number of “digits”
<small>https://en.wikipedia.org/wiki/Radix_sort</small> |
| | | <ul style="list-style-type: none"> external sorting
<small>https://en.wikipedia.org/wiki/External_sorting</small> – when not all of the data fits in memory at once |

Sorting – Key Algorithms

adapt for different sort orders by translating “less than” to “comparator says comes first”

selection sort	repeatedly pick the smallest remaining element in the unsorted portion of the array and swap it with the current position
heapsort	use a priority queue to repeatedly pick the smallest remaining element

```
SelectionSort(A)
  For i = 1 to n do
    Sort[i] = Find-Minimum from A
  Delete-Minimum from A
  Return(Sort)
```

heapsort is a variation of selection sort that selects from a heap instead of an unsorted array

```
void selection_sort(item_type s[], int n) {
  int i, j; /* counters */
  int min; /* index of minimum */

  for (i = 0; i < n; i++) {
    min = i;
    for (j = i + 1; j < n; j++) {
      if (s[j] < s[min]) {
        min = j;
      }
    }
    swap(&s[i], &s[min]);
  }
}
```

```
void heapsort_(item_type s[], int n) {
  int i; /* counters */
  priority_queue q; /* heap for heapsort */

  make_heap(&q, s, n);

  for (i = 0; i < n; i++) {
    s[i] = extract_min(&q);
  }
}
```

can be implemented in-place – use a max heap instead $s[0..n-i-1]$ is the heap, $s[n-i..n]$ sorted extract max swaps the root with the last element in the array, which is exactly where it should go

in-place sort – $s[0..i-1]$ is the sorted part of the array, $s[i..n]$ is unsorted final swap takes advantage of $s[i..n]$ unsorted to remove an element without shifting

Sorting – Key Algorithms

adapt for different sort orders by translating “less than” to “comparator says comes first”

insertion sort	repeatedly insert the next element into the correct position in the sorted part of the array
shellsort	repeatedly sort elements at different gap intervals, reducing the gap size over iterations

```
# Sort an array a[0..n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1] # Ciura gap sequence

# Start with the largest gap and work down to a gap of 1
# similar to insertion sort but instead of 1, gap is being used in each step
foreach (gap in gaps)
{
  # Do a gapped insertion sort for every element in gaps
  # Each loop leaves a[0..gap-1] in gapped order
  for (i = gap; i < n; i += 1)
  {
    # save a[i] in temp and make a hole at position i
    temp = a[i]
    # shift earlier gap-sorted elements up until the correct location for a[i] is found
    for (j = i; j >= gap; && (a[j - gap] > temp); j -= gap)
    {
      a[j] = a[j - gap]
    }
    # put temp (the original a[i]) in its correct location
    a[j] = temp
  }
}
```

shellsort is a variation of insertion sort that sorts all elements *gap* apart in each pass

the performance depends on the gap sequence, and can be complicated to analyze not as efficient as quicksort, but is significantly better than $O(n^2)$ and avoids recursion

```
insertion sort
for (i = 1; i < n; i++) {
  j = i;
  while ((j > 0) && (s[j] < s[j - 1])) {
    swap(&s[j], &s[j - 1]);
    j = j - 1;
  }
}
```

Sorting – Key Algorithms

adapt for different sort orders by translating "less than" to "comparator says comes first"

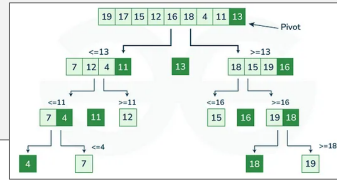
quicksort	use a pivot element to partition the array, then recursively sort the two subarrays
-----------	---

```
void quicksort(item_type s[], int l, int h) {
    int p; /* index of partition */

    if (l < h) {
        p = partition(s, l, h);
        quicksort(s, l, p - 1);
        quicksort(s, p + 1, h);
    }
}
```

```
int partition(item_type s[], int l, int h) {
    int i; /* counter */
    int p; /* pivot element index */
    int firsthigh; /* divider position for pivot element */

    p = h; /* select last element as pivot */
    firsthigh = l;
    for (i = l; i < h; i++) {
        if (s[i] < s[p]) {
            swap(&s[i], &s[firsthigh]);
            firsthigh++;
        }
    }
    swap(&s[p], &s[firsthigh]);
    return(firsthigh);
}
```



leads to worst-case $O(n^2)$ performance for sorted, reverse sorted

instead pick random element as pivot for $O(n \log n)$ expected performance (low probability of worst case)

avoid worst case from sorted and nearly sorted arrays by shuffling first – shuffling is $O(n)$

Sorting – Key Algorithms

adapt for different sort orders by translating "less than" to "comparator says comes first"

merge sort	divide the array in half, recursively sort the two halves, and then combine them into a single sorted array
------------	---

```
void merge_sort(item_type s[], int low, int high) {
    int middle; /* index of middle element */

    if (low < high) {
        middle = (low + high) / 2;
        merge_sort(s, low, middle);
        merge_sort(s, middle + 1, high);
    }

    merge(s, low, middle, high);
}
```

```
void merge(item_type s[], int low, int middle, int high) {
    int i; /* counter */
    queue buffer1, buffer2; /* buffers to hold elements for merging */

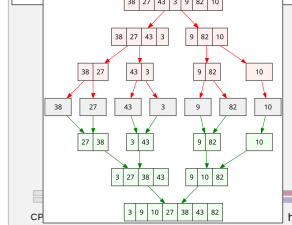
    init_queue(&buffer1);
    init_queue(&buffer2);

    for (i = low; i <= middle; i++) enqueue(&buffer1, s[i]);
    for (i = middle + 1; i <= high; i++) enqueue(&buffer2, s[i]);

    i = low;
    while (!empty_queue(&buffer1) || empty_queue(&buffer2)) {
        if (headq(&buffer1) <= headq(&buffer2)) {
            s[i++] = dequeue(&buffer1);
        } else {
            s[i++] = dequeue(&buffer2);
        }
    }

    while (!empty_queue(&buffer1)) {
        s[i++] = dequeue(&buffer1);
    }

    while (!empty_queue(&buffer2)) {
        s[i++] = dequeue(&buffer2);
    }
}
```

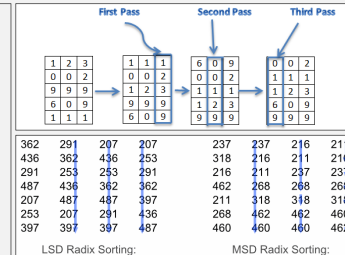


$O(n \log n)$ best/worst

multiway mergesort is suitable for external sorting – sort chunks of the data, then use 2- or k-way merging in a series of passes

Sorting – Key Algorithms

bucket sort	partition the elements into bins, then sort each bin
radix sort	process elements character-by-character or digit-by-digit, typically from right to left, sorting according to the current character/digit in each pass



buckets are based on a range of values for numbers, prefixes for strings

insertion sort can be used to sort buckets as each element is added, or another sorting algorithm to sort bucket contents after distribution step

each pass of radix sort sorts by distributing into buckets based on current digit/character

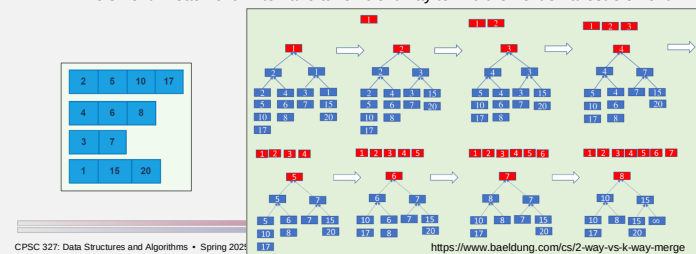
LSD processes entire collection in each pass – distribution must preserve order within buckets (stable)

MSD most suitable for strings or fixed-length integers

Sorting – Key Algorithms

external sorting	any technique used for sorting data sets that are too big to fit into memory
------------------	--

- multiway mergesort
 - sort chunks that are small enough to fit into memory
 - iterative 2-way merge
 - repeatedly merge pairs of chunks
 - direct k-way merge
 - merge k chunks at once, maintaining a heap or tournament tree of the next element in each chunk to have an efficient way to find the next smallest element



Sorting

- best general-purpose sorting algorithm is quicksort
 - use a library implementation because there are many details to achieve maximum performance
- heapsort is a good choice if you have to implement a sorting algorithm
 - $O(n \log n)$
 - easy to implement, especially with a priority queue available
 - can be implemented in-place

"Although other algorithms prove slightly faster in practice, you won't go wrong using heapsort for sorting data that sits in the computer's main memory." [Skienna p. 121]

Sorting

Choosing a sorting algorithm –

ADM 17.1

- how many keys are being sorted?
 - for small n (≤ 100), $O(n^2)$ is fine and simple implementations are preferred
 - for medium and large n , $O(n \log n)$ is needed
 - for very large n ($\geq 100,000,000$), need external sorting
- are there duplicate keys?
 - a *stable* sorting algorithm keeps equal keys in the original order
 - many algorithms are not inherently stable, but can be made so by including the initial position as a secondary key
 - having many equal keys amongst the elements being sorted can lead to performance problems for some algorithms if not explicitly designed for it
- does the data have special properties?
 - is it already partially sorted?
 - are the keys uniformly distributed across the range of values?
 - are the keys long?
 - is the range of keys very small?

Sorting in Java – Arrays

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html>

static void	sort(int[] a)	Sorts the specified array into ascending numerical order.
static void	sort(int[] a, int fromIndex, int toIndex)	Sorts the specified range of the array into ascending order.

- plus similar versions for arrays of the other primitive types

sort

```
public static void sort(int[] a)
```

Sorts the specified array into ascending numerical order.

Implementation Note:

The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers $O(n \log(n))$ performance on all data sets, and is typically faster than traditional (one-pivot) Quicksort implementations.

Parameters:

a - the array to be sorted

Sorting in Java – Arrays

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Arrays.html>

static void	sort(Object[] a)	Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.
static void	sort(Object[] a, int fromIndex, int toIndex)	Sorts the specified range of the specified array of objects into ascending order, according to the natural ordering of its elements.
static <T> void	sort(T[] a, int fromIndex, int toIndex, Comparator<T> c)	Sorts the specified range of the specified array of objects according to the order induced by the specified comparator.
static <T> void	sort(T[] a, Comparator<T> c)	Sorts the specified array of objects according to the order induced by the specified comparator.

- *natural ordering* means the elements must implement the Comparable interface

Sorting in Java – Arrays

```
public static void sort(Object[] a)
```

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the `Comparable` interface. Furthermore, all elements in the array must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the array).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

Implementation note: This implementation is a stable, adaptive, iterative mergesort that requires far fewer than $n \lg(n)$ comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to $n/2$ object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort[®]). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Parameters:

`a` - the array to be sorted

Throws:

`ClassCastException` - if the array contains elements that are not *mutually comparable* (for example, strings and integers)

`IllegalArgumentException` - (optional) if the natural ordering of the array elements is found to violate the `Comparable` contract

Sorting in Java – List

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html>

```
default void sort(Comparator<? super E> c)
```

Sorts this list according to the order induced by the specified `Comparator`. The sort is *stable*: this method must not reorder equal elements.

All elements in this list must be *mutually comparable* using the specified comparator (that is, `c.compare(e1, e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

If the specified comparator is `null` then all elements in this list must implement the `Comparable` interface and the elements' natural ordering should be used.

This list must be modifiable, but need not be resizable.

Implementation Requirements:

The default implementation obtains an array containing all elements in this list, sorts the array, and iterates over this list resetting each element from the corresponding position in the array. (This avoids the $n^2 \log(n)$ performance that would result from attempting to sort a linked list in place.)

Implementation Note:

This implementation is a stable, adaptive, iterative mergesort that requires far fewer than $n \lg(n)$ comparisons when the input array is partially sorted, while offering the performance of a traditional mergesort when the input array is randomly ordered. If the input array is nearly sorted, the implementation requires approximately n comparisons. Temporary storage requirements vary from a small constant for nearly sorted input arrays to $n/2$ object references for randomly ordered input arrays.

The implementation takes equal advantage of ascending and descending order in its input array, and can take advantage of ascending and descending order in different parts of the same input array. It is well-suited to merging two or more sorted arrays: simply concatenate the arrays and sort the resulting array.

The implementation was adapted from Tim Peters's list sort for Python (TimSort[®]). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Sorting in Java – Collections

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html>

```
static <T extends Comparable<? super T>> void sort(List<T> list)
static <T> void sort(List<T> list, Comparator<? super T> c)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements.

Sorts the specified list according to the order induced by the specified comparator.

- **natural ordering** means the elements must implement the `Comparable` interface

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sorts the specified list into ascending order, according to the natural ordering of its elements. All elements in the list must implement the `Comparable` interface. Furthermore, all elements in the list must be *mutually comparable* (that is, `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the list).

This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort.

The specified list must be modifiable, but need not be resizable.

Implementation Note:

This implementation defers to the `List.sort(Comparator)` method using the specified list and a `null` comparator.

Type Parameters:

Comparing in Java

- when there is a natural ordering, the type itself implements `Comparable`

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html>

- when there is not a natural ordering, define a standalone *comparator*
 - separate class implementing `Comparator`

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Comparator.html>

Comparable

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure $\text{signum}(x.\text{compareTo}(y)) == -\text{signum}(y.\text{compareTo}(x))$ for all x and y . (This implies that $x.\text{compareTo}(y)$ must throw an exception if and only if $y.\text{compareTo}(x)$ throws an exception.)

The implementor must also ensure that the relation is transitive: $(x.\text{compareTo}(y) > 0 \ \&\& \ y.\text{compareTo}(z) > 0)$ implies $x.\text{compareTo}(z) > 0$.

Finally, the implementor must ensure that $x.\text{compareTo}(y)=0$ implies that $\text{signum}(x.\text{compareTo}(z)) == \text{signum}(y.\text{compareTo}(z))$, for all z .

API Note:

It is strongly recommended, but *not* strictly required that $(x.\text{compareTo}(y)=0) == (x.\text{equals}(y))$. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Parameters:

`o` - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

`NullPointerException` - if the specified object is null

`ClassCastException` - if the specified object's type prevents it from being compared to this object.

Comparator

```
int compare(T o1,  
           T o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

The implementor must ensure that $\text{signum}(\text{compare}(x, y)) == -\text{signum}(\text{compare}(y, x))$ for all x and y . (This implies that $\text{compare}(x, y)$ must throw an exception if and only if $\text{compare}(y, x)$ throws an exception.)

The implementor must also ensure that the relation is transitive: $((\text{compare}(x, y)>0) \ \&\& \ (\text{compare}(y, z)>0))$ implies $\text{compare}(x, z)>0$.

Finally, the implementor must ensure that $\text{compare}(x, y)=0$ implies that $\text{signum}(\text{compare}(x, z))=\text{signum}(\text{compare}(y, z))$ for all z .

API Note:

It is generally the case, but *not* strictly required that $(\text{compare}(x, y)=0) == (x.\text{equals}(y))$. Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

Parameters:

`o1` - the first object to be compared.

`o2` - the second object to be compared.

Returns:

a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Throws:

`NullPointerException` - if an argument is null and this comparator does not permit null arguments

`ClassCastException` - if the arguments' types prevent them from being compared by this comparator.