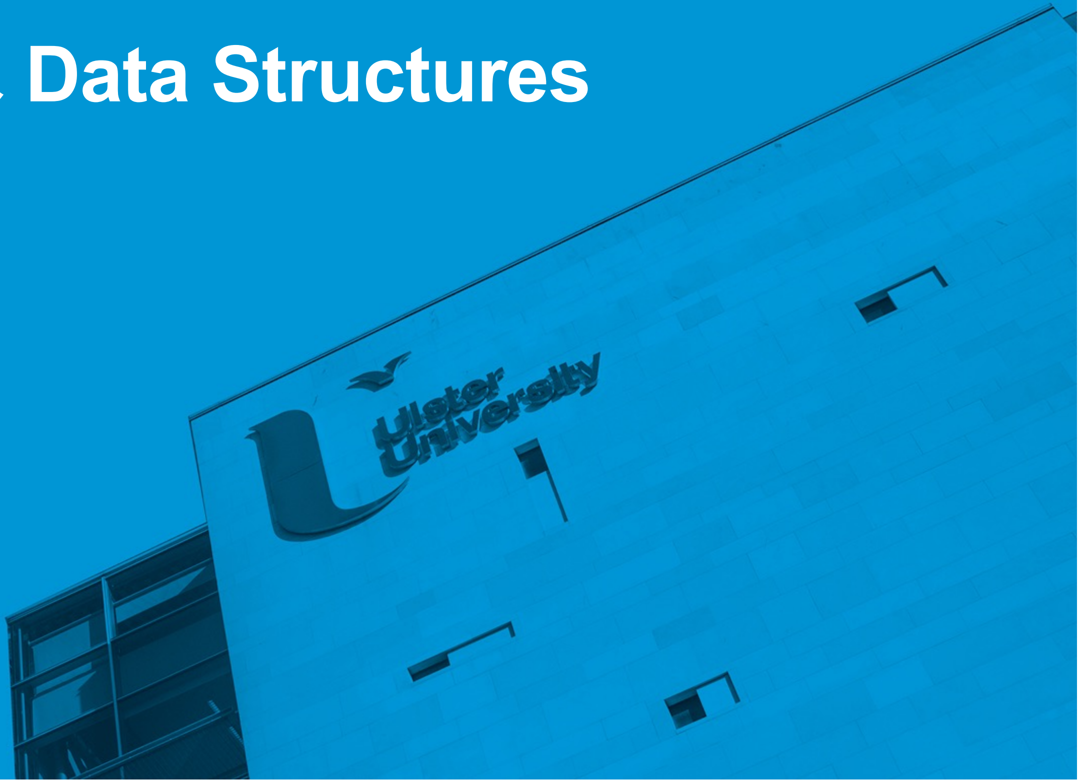
COM498 Algorithms & Data Structures

7.3 Merge Sort

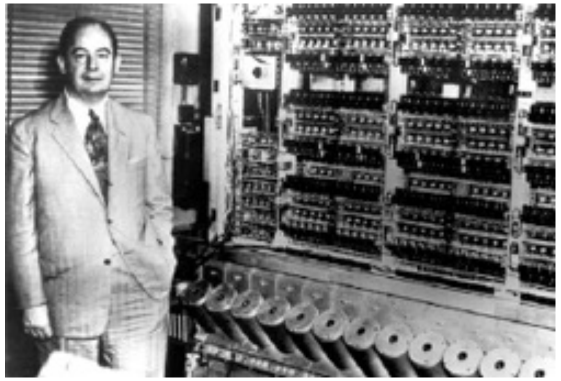


Limitations of O(n^2) methods

- Previous sorting algorithms often sufficient when you want to sort small arrays.
- However, when you need to sort very large arrays frequently *Bubble Sort*, *Selection Sort*, *Insertion Sort* and *Shell Sort* are too inefficient.
- All have an execution time related to the square of the number of elements to be sorted – so the sort time grows rapidly as the number of elements increases.

Merge Sort

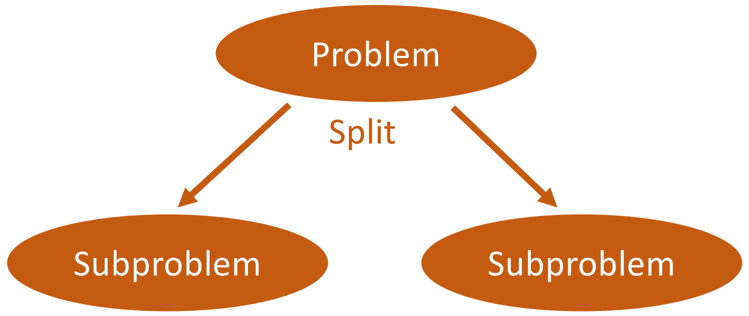
- Invented by John Von Neumann in 1945
- Basic plan is very simple – **divide and conquer**
  - Divide array into two halves
  - Sort each half
  - Merge the two halves

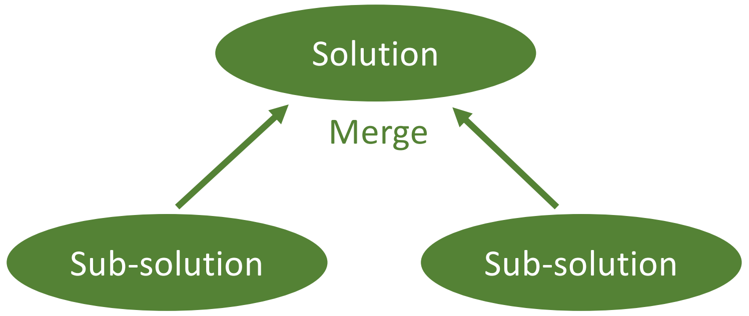


input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Divide and Conquer Algorithms

- Divide and conquer algorithm strategy (to problem solving):

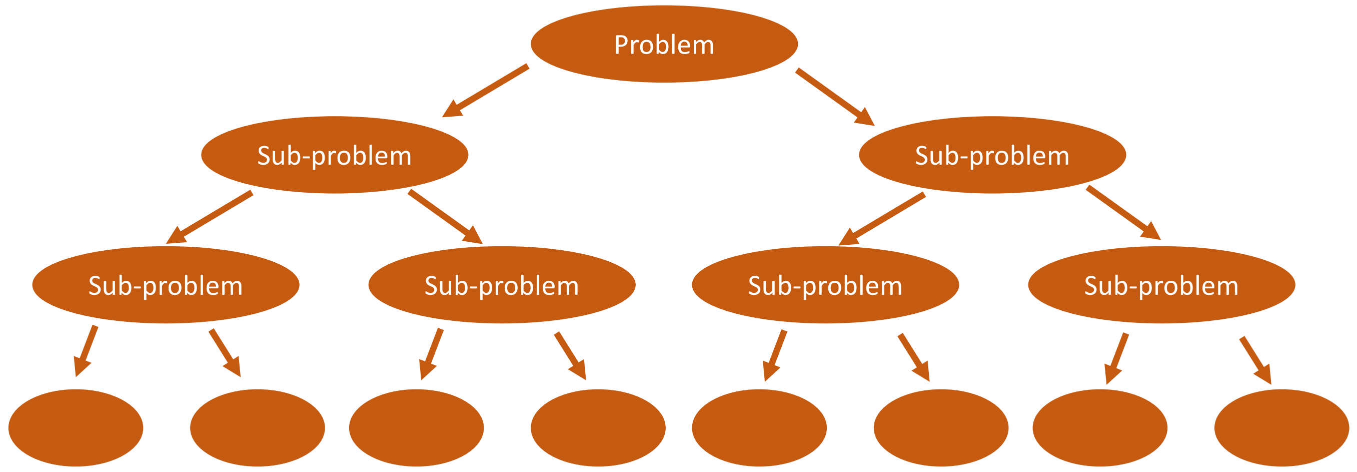




  - divide a problem into two or more small but distinct problems,
    solve each new problem, then combine their solutions to solve the original problem

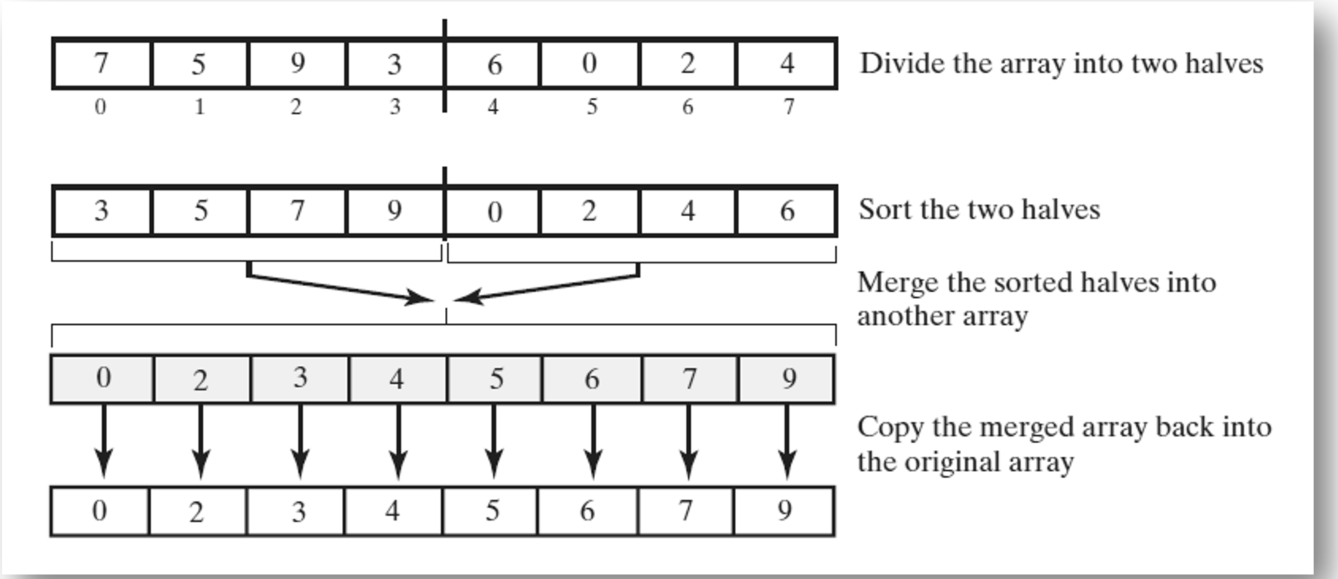
Divide and Conquer Algorithms

- Often applied recursively – sub-problems are themselves divided until the smallest possible problem is found



Recursive Merge Sort

- Merge two sorted arrays into a temporary array, then copy the temporary array back to the original array



7 5 9 3 6 0 2 4
0 1 2 3 4 5 6 7
Divide the array into two halves
3 5 7 9 0 2 4 6
Sort the two halves
Merge the sorted halves into another array
0 2 3 4 5 6 7 9
Copy the merged array back into the original array
0 2 3 4 5 6 7 9

- How do we sort the two halves of the array? **Use a merge sort!**

Recursive Merge Sort

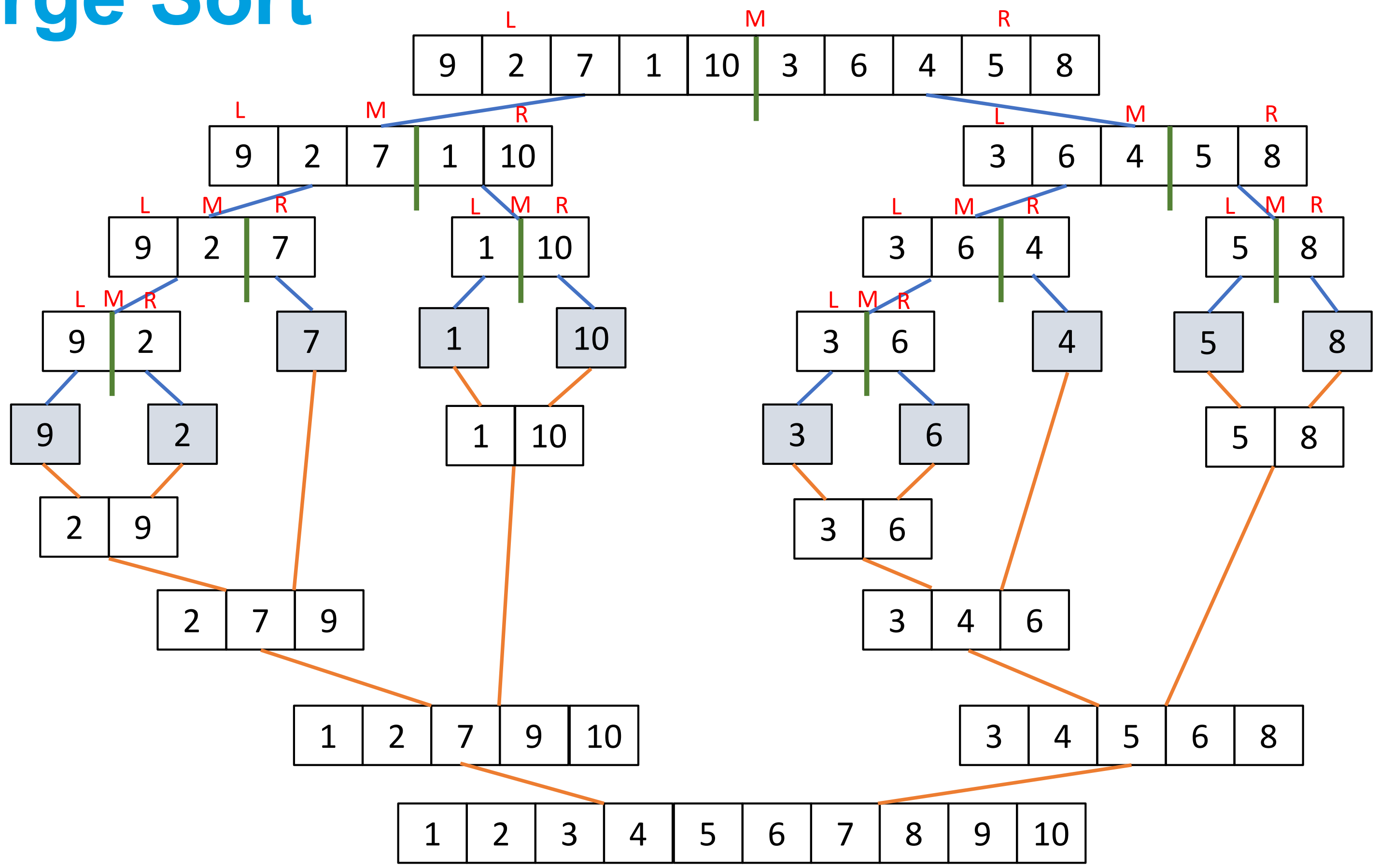
- Pseudocode of recursive algorithm for Merge Sort:

```
Algorithm mergeSort(a, first, last)
// Sorts the entries of an array a between positions first and last.

if first < last
    set mid to the approximate midpoint between first and last

    // find the gap
    call mergeSort(a, first, mid)
    call mergeSort(a, mid + 1, last)
    merge the sorted arrays a[first] to a[mid] and a[mid + 1] to a[last]
```

Merge Sort

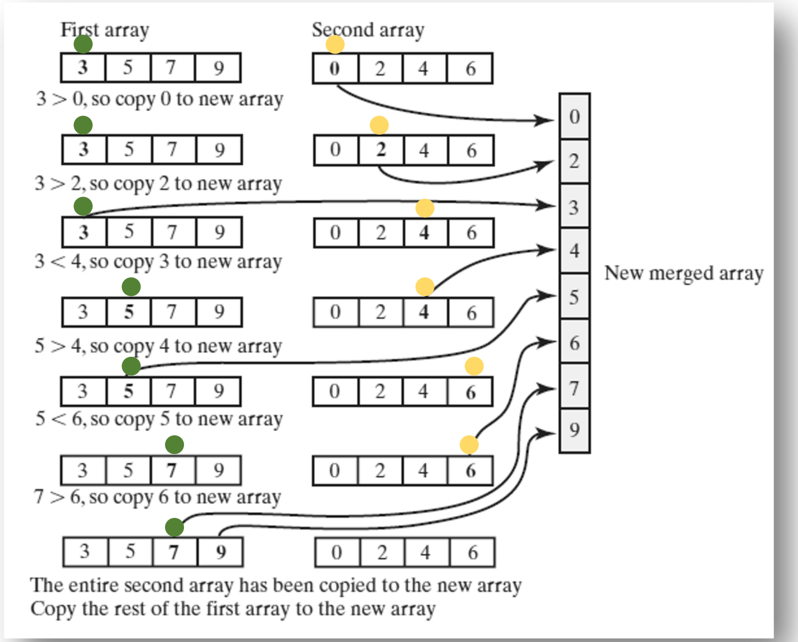


L
M
R
9 2 7 1 10 3 6 4 5 8
9 2 7 1 10
3 6 4 5 8
9 2 7
1 10
3 6 4
5 8
9 2
7
1
10
3 6
4
5
8
9
2
1 10
3
6
5 8
2 9
3 6
2 7 9
3 4 6
1 2 7 9 10
3 4 5 6 8
1 2 3 4 5 6 7 8 9 10

Merge Sort (Merging Arrays)

- The real effort during execution of merge sort occurs during the merge step (involves programming effort)

- Merging two sorted arrays isn't difficult but does require an additional (auxiliary) array
- Processing both arrays from beginning to end:
  - Compare an entry in one array with an entry in the other array
  - Copy smaller entry to the new array
- After reaching end of one array, simply copy remaining entries from other array to the auxiliary array


First array
Second array
3 > 0, so copy 0 to new array
3 > 2, so copy 2 to new array
3 < 4, so copy 3 to new array
5 > 4, so copy 4 to new array
5 < 6, so copy 5 to new array
7 > 6, so copy 6 to new array
New merged array
The entire second array has been copied to the new array
Copy the rest of the first array to the new array

Recursive Merge Sort

- Pseudocode of algorithm to merge 2 sorted arrays:

```
Algorithm merge (a, first, mid, last)
// Merge the array from a[first] to a[mid] with the array from a[mid + 1] to a[last]
// using a temporary array.

set pointers to the first elements in each of the 3 arrays (2 source and temporary)
while both source arrays have elements remaining to be processed
   if the current element in the first source array <= that from the second array
      copy from first source array to the temporary array
      increment pointers into first and temporary arrays
   else
      copy from second source array to the temporary array
      increment pointers into second and temporary arrays
copy elements from remaining source array into the temporary array
copy sorted temporary array values back into a
```

Implementation note

- The algorithm or the **`merge()`** operation suggests that we should create a temporary array into which we merge the values from the pair of source arrays
    - This would require that we create a new array every time that **`merge()`** is called – very expensive and slow!
    - A much better approach is to create the temporary array once in the main sort method and pass it to the **`merge()`** operation as a parameter
    - Hence the **`merge()`** method will have the signature

```
private static void merge(int[] arr, int[] temp,
                          int first, int mid, int last)
```

Scenario

- Implement the Recursive Merge Sort

  - As for the previous sorts, add code to display the number of comparisons and array updates.

  - Test the implementation for an integer array of 10 values where…
    1. the array to be sorted is in random order
    2. the array is already in sorted order
    3. the initial array is in reverse order

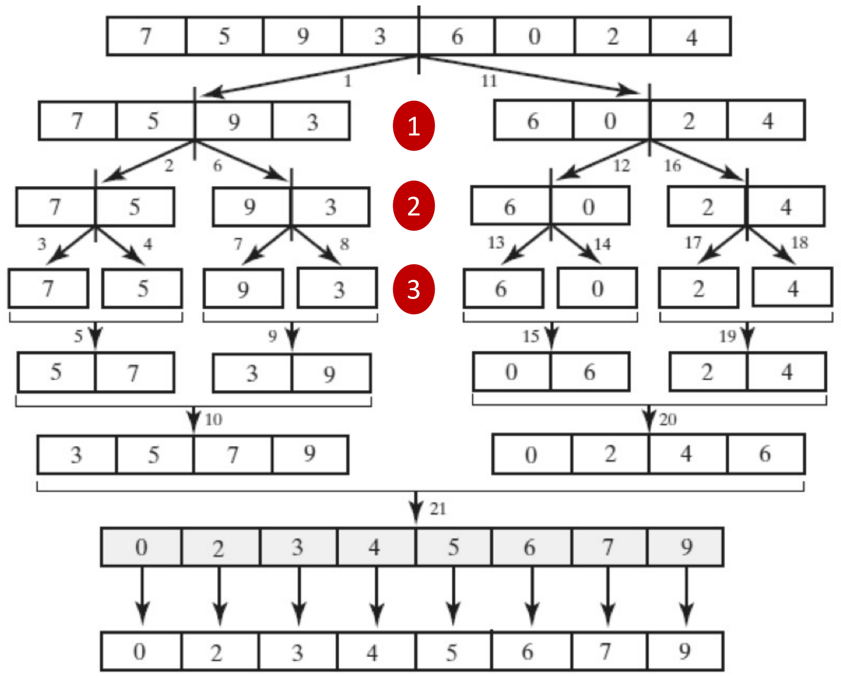
Challenge

1. Measure the performance of the recursive Merge Sort technique
    - Revisit the previous challenge and measure the execution time to sort 1000 arrays of 100, 200, 400, 800, 1600, 3200 and 6400 elements.
    - Report the results as a series of 2 values per line – array size and sort time.

2. The Implementation note earlier suggested that the temporary array be passed to the `merge()` method as a parameter. Measure how effective this tip is in practice, by timing 1000 sorts of arrays of 5,000 and 10,000 integers
    i) With the `temp` array passed as a parameter (as in our implementation)
    ii) With the `temp` array generated anew each time the `merge()` method is called. Remember to remove `temp` from all parameter lists for this test.

Efficiency of Merge Sort

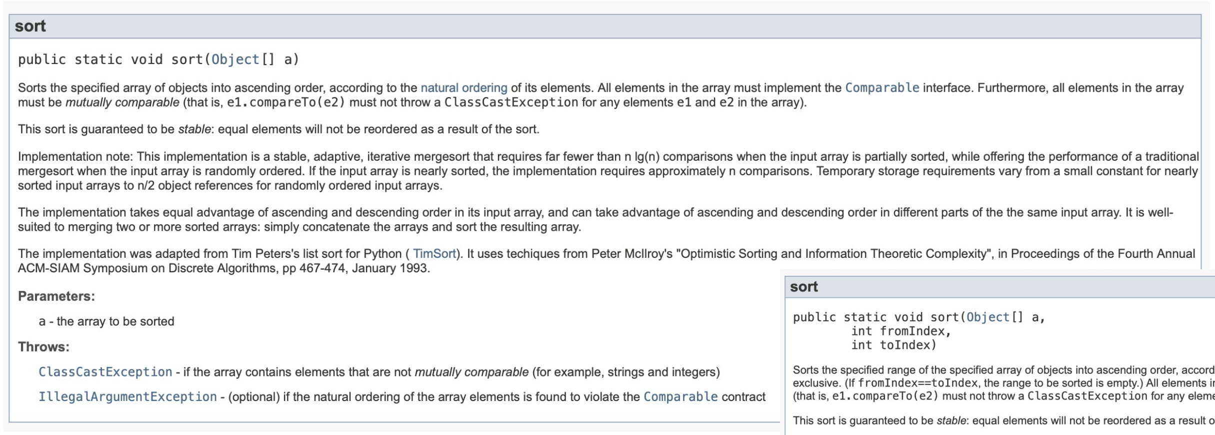
- In worst-case merge sort has an order of O(n log n)



- Assume n is a power of 2 (such as shown)
- In general, if n is 2^k, k levels of recursive calls occur, where $k = \log_2 n$
- At each call to the merge step, at most $n - 1$ comparisons are required
- Each merge also requires n moves to the temporary array and n moves back to the original array
- In total each merge requires $3n - 1$ operations

Merge Sort in the Java Class Library

- Class **`Arrays`** in the package **`java.util`** defines versions of a static method **`sort`**



sort

```
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 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 techiques 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.

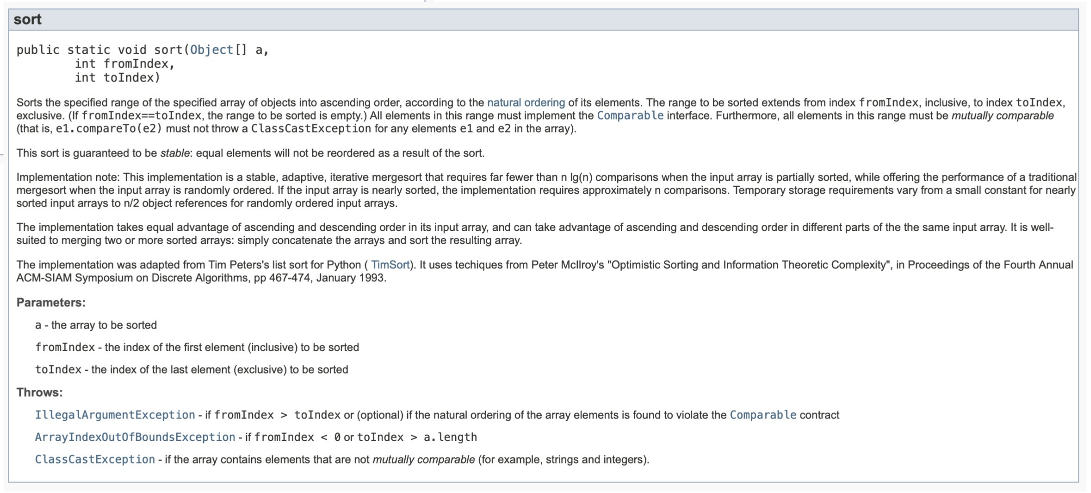
Parameters:

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



sort

```
public 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. The range to be sorted extends from index `fromIndex`, inclusive, to index `toIndex`, exclusive. (If `fromIndex==toIndex`, the range to be sorted is empty.) All elements in this range must implement the `Comparable` interface. Furthermore, all elements in this range 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 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 techiques 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

`fromIndex` - the index of the first element (inclusive) to be sorted

`toIndex` - the index of the last element (exclusive) to be sorted

Throws:

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

`ArrayIndexOutOfBoundsException` - if `fromIndex < 0` or `toIndex > a.length`

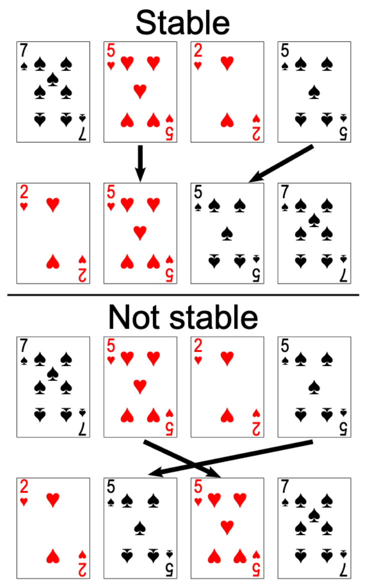
`ClassCastException` - if the array contains elements that are not *mutually comparable* (for example, strings and integers).

Stable Sorting Algorithms

- A sorting algorithm is **stable** if it does not change the relative order of objects that are equal

- For example, if object **x** appears before object **y** in a collection of data, and **`x.compareTo(y) = 0`**, a stable sorting algorithm will leave object **x** before object **y** after sorting the data

- Stability is important for certain applications - for example, sort a group of people first by name then by age, a stable sorting algorithm will ensure that people of the same age will remain in alphabetical order


Stable
Not stable