Chapter 12 Sorting Algorithms

Contents

Syllabus Learning Outcomes

1.2 Fundamental Algorithms
Understand algorithms for sorting and searching methods such as insertion sort, bubble sort, quicksort, merge sort, linear search, binary search and hash table search, and use examples to explain these methods.

1.2.1 Implement sort algorithms.
– Insertion sort
– Bubble sort
– Quicksort
– Merge sort

1.2.2 Use examples to explain sort algorithms.

1.2.5 Compare and describe the efficiencies of the sort and search algorithms using Big-O notation for time complexity (worst case).
Exclude: space complexity

2.3 Implementing Algorithms and Data Structures
Use programming language elements and constructs to implement sort and search algorithms such as insertion sort, bubble sort, quicksort, merge sort, linear search, binary search, and hash table search, as well as data structures such as stacks, queues, linear linked lists, and binary search trees.

2.3.1 Implement sort programs.
– Insertion sort
– Bubble sort
– Quicksort
– Merge sort

Sorting algorithms are used to put data in order. This data may be numbers, strings, records or objects. The four sorting algorithms you are expected to know are bubble sort, insertion sort, merge sort and quicksort.

Sorting algorithms could be compared based on

- Time taken to complete the work
- Memory requirement to complete the work
- Stability in keeping the input orders of items having the same sorting order

1. Bubble sort

Bubble sort is one of the easiest sorting algorithms to understand and implement; however, it is very inefficient compared to other alternatives.

1.1 Non-optimised bubble sort

Let's look at how a simple bubble sort is implemented.

It works as follows:

Compare the first and second values. If the first value is larger than the second value, swap them.

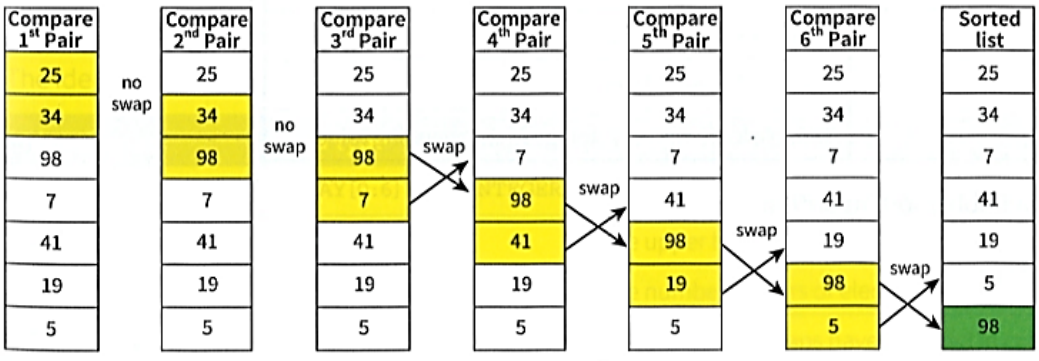
Compare the second and third values. If the second value is larger than the third value, swap them.

Keep on comparing adjacent values, swapping them if necessary, until the last two values in the list have been processed.

When we have completed the first pass through the entire array, the largest value is in the correct position at the end of the array. The other values may or may not be in the correct order.

We need to work through the array again and again with each pass having one less value to compare.

Illustration of bubble sort in one pass:



Swapping values working down the array in one pass

Original list	After pass 1	After pass 2	After pass 3	After pass 4	After pass 5	After pass 6
25	25	25	7	7	7	5
34	34	7	25	19	5	7
98	7	34	19	5	19	19
7	41	19	5	25	25	25
41	19	5	34	34	34	34
19	5	41	41	41	41	41
5	98	98	98	98	98	98

States of the array after each pass

The algorithm of non-optimised bubble sort in pseudocode is:

```
FOR passes ← 0 TO length of data - 2
  FOR j ← 0 TO length of data - 2 - passes
    IF data[j] > data[j + 1]
        THEN
        data[j], data[j + 1] ← data[j + 1], data[j]
        ENDIF
  ENDFOR
ENDFOR
```

The values to be sorted may already be in the correct order before the outer loop has been through all its iterations. Look at the list of values below.

Original list	After pass 1	After pass 2	After pass 3	After pass 4	After pass 5	After pass 6
5	5	5	5	5	5	5
34	34	7	7	7	7	7
98	7	34	19	19	19	19
7	41	19	25	25	25	25
41	19	25	34	34	34	34
19	25	41	41	41	41	41
25	98	98	98	98	98	98

States of the array after each pass

After the third pass, the values are all in the correct order but the algorithm will continue to run. This means that we are making comparisons when no further swaps need to be made.

1.2 Optimised bubble sort

If we have gone through one pass without swapping any values, this means that the values must be in the correct order. So, we can improve the algorithm by using a variable `swapped` to store whether a swap has taken place during the current pass.

When we swap a pair of values, we set `swapped` to `True`. This indicates that the list was not sorted. If `swapped` is `False` at the end of one pass through the list, this indicates that none of the values are swapped. If none of the values are swapped, the list must be sorted, and we can exit the sort algorithm. This allows us to cut down on any unnecessary comparisons and passes which improves the bubble sort algorithm to make it optimised.

The algorithm of optimised bubble sort in pseudocode is:

```
swapped □ TRUE
passes □ length of data – 1

WHILE swapped DO
  swapped □ FALSE
    FOR j □ 0 TO passes - 1
      IF data[j] > data[j + 1]
        THEN
          data[j], data[j + 1] □ data[j + 1], data[j]
          swapped □ TRUE
      ENDIF
    ENDFOR
    passes □ passes – 1
ENDWHILE
```

1.3 Time complexity of bubble sort

Iteration	No. of comparisons
1	n-1
2	n-2
...	...
n-1	1

First, we assume that we will be sorting the array (of size n) using bubble sort in an ascending order. At the first iteration of the bubble sort, we are iterating through n-1 elements, until the largest element 'bubbles' to the end of the array. At the second iteration, we are iterating through n – 2 elements. After the second iteration, the second largest element bubbles up to the second last position of the array. This process continues until the whole array is sorted (or no more swaps occur, in the optimised case).

No. of comparisons = (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1 = $\frac{(n-1)n}{2}$

Best case time complexity: Ω(n)
Given a sorted list, we only need one pass with n-1 comparisons on the values in the list to check that it is sorted. This example will give a time complexity of Ω(n).

Worst case time complexity: $O(n^2)$
The non-optimised bubble sort algorithm will give the worst case time complexity of $O(n^2)$ as it needs to make (n-1)*(n-2) comparisons to complete the sort.

2. Insertion Sort

Insertion sort works by dividing an array into two parts: sorted and unsorted. Elements are inserted one by one into their correct position in the sorted section.

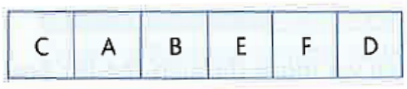
To sort an array in ascending order using insertion sort, we start with the first element in the array. One element by itself is already sorted. Then we consider the next element in the unsorted array. If it is smaller than the first element, we insert this element to the left of the first element, else we place it to the right. Next, we consider the third element in the array. We make comparisons with the elements in the sorted array leftwards until we find the correct position to insert this element into the sorted array. We then repeat this for the remaining elements, until the whole array is sorted.

The pseudo code of insertion sort algorithm is:

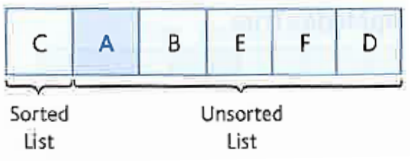
```
FOR i  1 TO length of data -1
      curr  data[i] //Take the first item of the unsorted list
      position  i
      WHILE position > 0 AND curr < data[position – 1] DO
            data[position]  data[position – 1]
            position  position – 1
      ENDWHILE
      data[position]  curr
ENDFOR
```

Illustration of insertion sort:

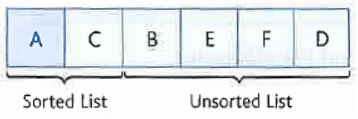
Let's start with a list of unsorted letters.



The first element is C and it becomes a member of the sorted list.

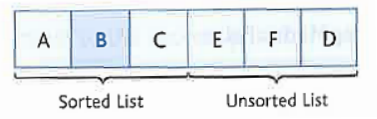

C
A
B
E
F
D
Sorted List
Unsorted List

A, the first item of the unsorted list is smaller than C so is inserted to the left of it.



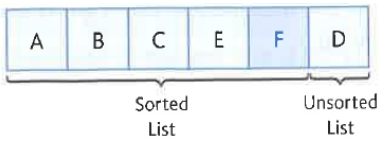
A and C are now both in the sorted list.

B is now the first item in the unsorted list. B is less than C so C moves to a position ahead of its original position. B is not less than A, so B is inserted between A and C.



E is now the first item of the unsorted list.
E is not less than C so it joins the sorted list without shifting any elements.

Similarly, F is not less than E, so it then joins the sorted list without shifting any elements.



D is now the only member of the unsorted list. It is less than F, so F shifts to the right.

D is less than E so E shifts to the right.

D is not less than C so D is inserted into the position between C and E.
All items in the list are now members of the sorted list.

2.1 Time complexity of insertion sort

The outer for-loop in insertion sort always iterates n-1 times.
The inner while loop will make (n – 1)*1 comparisons in best case if all values are in correct order. The inner while loop will make 1 + 2 + 3 + … + (n – 2) + (n – 1) comparisons in worst case.

Best case time complexity: Ω(n)

Worst case time complexity: $O(n^2)$

3. Merge Sort

A divide-and-conquer algorithm solves problem by breaking it into subproblems, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Some key benefits of using a divide and conquer approach are speed and it simplifies complexity. A well constructed divide and conquer algorithm is typically very fast. It breaks a complex topic down into small and easy to manage chunks, which is most likely easy to solve.

Merge sort is a divide and conquer algorithm, which has elegant recursive formulations and is highly efficient. The implementation of the merge sort algorithm needs two functions:
1. A function that recursively splits the input array into half (`merge_sort(array)`)
2. A function that merges both halves, producing a sorted array. (`merge(list1,list2)`)

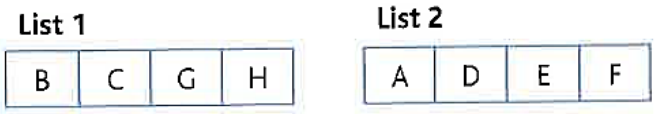
To understand merge sort, we first need to understand how we merge lists. If we have two sorted lists in ascending order, List1 and List2, we compare the first item in both lists. If the first item in List1 is smaller than that in List2, the first item is removed from List1 and added to a new list. If the first item in List2 is smaller than that in List1, the first item is removed from List2 and added to the new list. This comparison is repeated until either List1 or List2 becomes empty. The remaining items of the other list are added to the new list and merge sort is complete.

The merge algorithm in pseudo code:

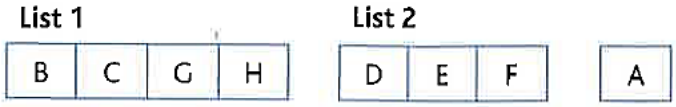
```
FUNCTION merge (list1 : ARRAY, list2 : ARRAY) RETURNS ARRAY
    i □ 0
    j □ 0
    new_list □ []
    WHILE i < length of list1 AND j < length of list2 DO
      IF list1[i] < list2[j]
        THEN
            Append list1[i] to new_list
            i □ i + 1
        ELSE
            Append list2[j] to new_list
            j □ j + 1
      ENDIF
    ENDWHILE

    //If list1 still has remaining items, add remaining items in list1 to
    //new_list, else add remaining items in list2 to new_list
    IF i < length of list1
      THEN
        new_list □ new_list + list1[i: ]
      ELSE
```

```
        new_list  new_list + list2[j: ]
    ENDIF
  RETURN new_list
ENDFUNCTION
```

Illustration of merge function:



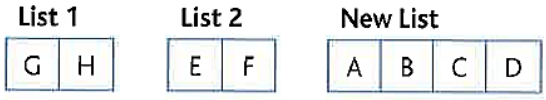
List1 and List2 are sorted. The first item of List2 (A) is lower than the first item of List1 (B), so the first item of List2 is added to the new list.

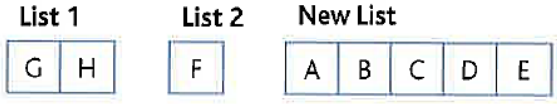


Now the first item in List1 (B) is smaller than first item in List2 (D), so the first item of List1 is added to the new list.

Again, the first item of List1 is smaller than first item of List2, so the first item of List1 is added to the new list.

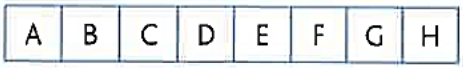
This process of comparing the first items of the two lists continues until either list has no more remaining items to compare.





Finally, List2 has no more items left to compare. We therefore, append the remaining items of List1 to the new list.

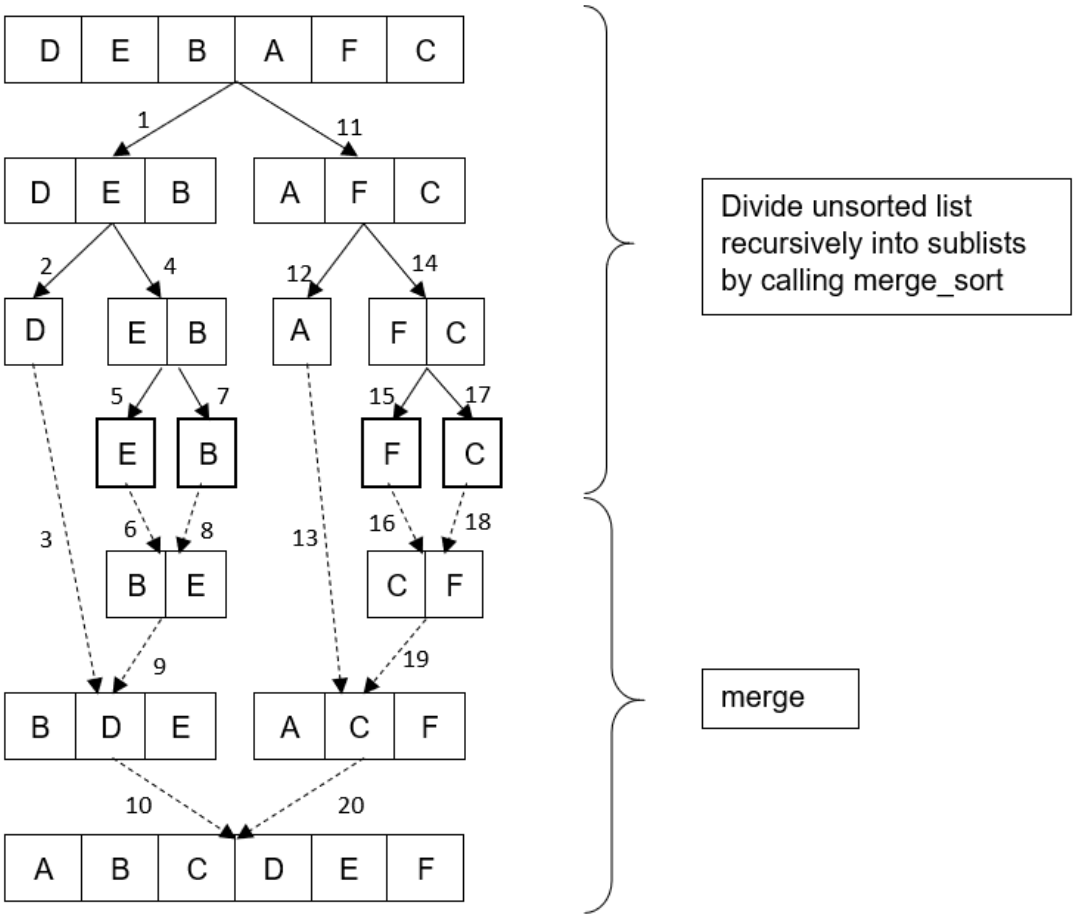
List 1 **List 2** **New List**



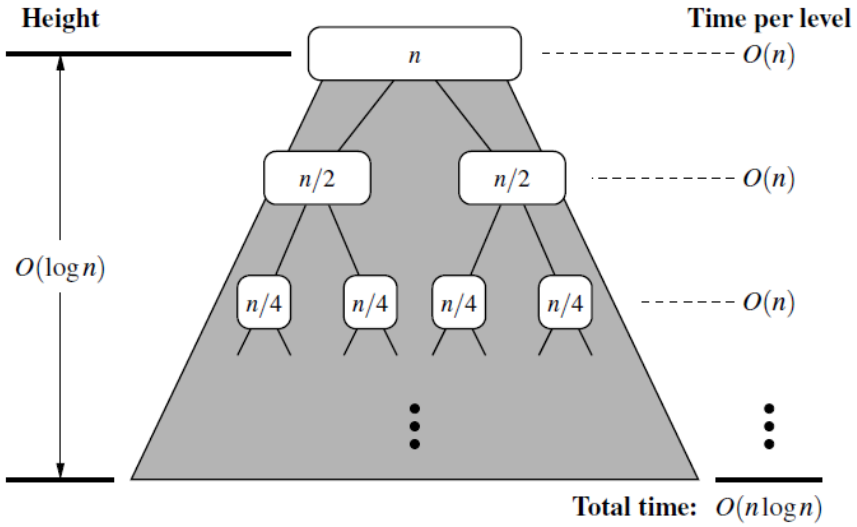
Merge sort recursively divides an unsorted array into subarrays, until each containing one element (an array of one element is considered a sorted array). Thereafter, the algorithm repeatedly pairs up the subarrays and merge each pair into a new sorted subarray, using the above process of merging arrays, until there is only one subarray remaining and that will be the sorted array.

```
FUNCTION merge_sort(A : ARRAY) RETURNS ARRAY
  DECLARE mid : INTEGER
  IF Len(A) > 1
     mid  length of A // 2
     list1  A[: Mid] //copy of first half of A
     list2  A[Mid :] //copy of second half of A
     sorted_list1  merge_sort(list1) //sort copy of first half
     sorted_list2  merge_sort(list2) //sort copy of second half
     RETURN merge(sorted_list1, sorted_list2) //merge sorted halves
  ENDIF
  RETURN A
ENDFUNCTION
```

Illustration of merge sort:



3.1 Time complexity of merge sort



As we have already learned in binary search that whenever we divide a number into half in every step, the time complexity can be represented using a logarithmic function, which is O(log n).

The time complexity of merging the subarrays of n elements is O(n)

Hence, merge sort will give a time complexity of O(n*log n) = O(n log n).

The time complexity for both worst and best cases is O(n log n) as merge sort always divides the array into two halves and take linear time to merge two halves.

Best case time complexity: Ω(n log n)
Worse case time complexity: O(n log n)

4. Quicksort

Like merge sort, quicksort is also based on the divide and conquer algorithm, but it uses this technique in an opposite manner, as all the hard work is done before the recursive calls.

4.1 Out-of-place quicksort

Quicksort first selects a pivot element. The pivot element can be first, last, middle or any random element in the array. The algorithm then partitions the array around the pivot, putting every element less than the pivot in a `less_than` array, and every element greater than and equal to the pivot in a `greater_equal` array. The `equal` array will just hold one element – the pivot itself. The process repeats for the `less_than` array and `greater_equal` array. Eventually, the elements are combined in order by first inserting the elements of `less_than`, then those of `equal`, and finally those of `greater_equal`.

The algorithm of out-of-place quicksort in pseudo code:

```
FUNCTION quicksort (A : ARRAY) RETURNS ARRAY
    IF NOT A  // If A is empty
        RETURN []

    less_than  an array of all elements in A less than pivot
    equal  pivot (element in the middle of the array)
    greater_equal  an array of all elements in A greater than or
                    equal to pivot

    result  quicksort(less_than) + [equal] + quicksort(greater_equal)

    RETURNS result
ENDFUNCTION
```

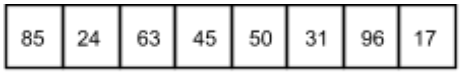
This works well if all elements are unique. If the original array contains elements with duplicates, the order of occurrence of two similar elements in the array may be altered and this makes the algorithm an unstable sorting algorithm.

An improvement can be made to make the above algorithm into a stable one. Instead of adding elements greater than and equal to the pivot into the `greater_equal` array, elements equal to the pivot can be stored in an `equal` array and elements greater than pivot can be stored in a `greater` array. This will preserve the original order of occurrence of the elements.

```
pivot  element in the middle of the array
less  an array of all elements in A less than pivot
equal  an array of all elements in A equal to pivot
```

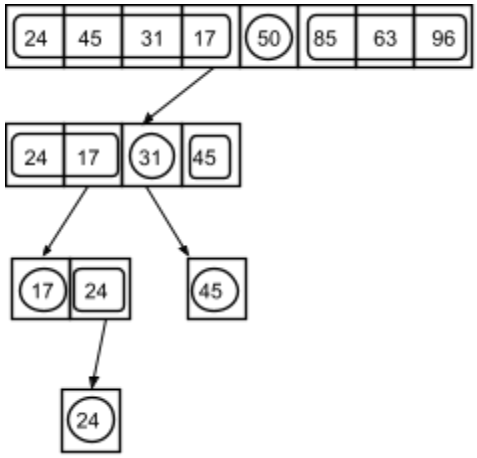
```
greater  an array of all elements in A greater than pivot
```

Illustration of quicksort:

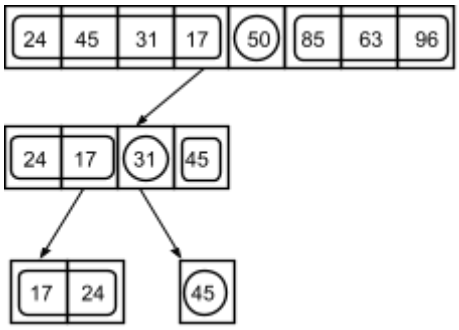


1. Starting with the array above, we take the middle element as pivot (it does not need to be the middle element, it can be any). We then divide the input array into 3 subarrays – `less`, `greater` and `equal`. Elements less than pivot will be appended to the `less` subarray, elements equal to pivot will be appended to the `equal` subarray and elements greater than pivot will be appended to the `greater` subarray. There is no attempt in sorting the arrays. Elements are just added in order.

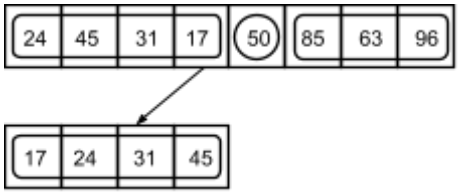
2. The same process is repeated for the `Less` subarray



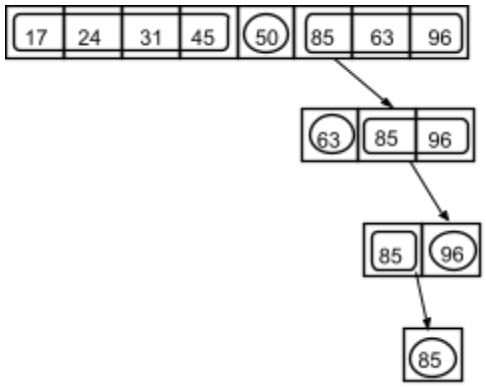
3. The subarrays are concatenated and returned.



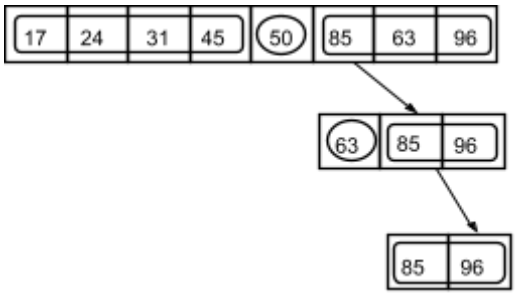
4. The subarray continues to concatenate and return.



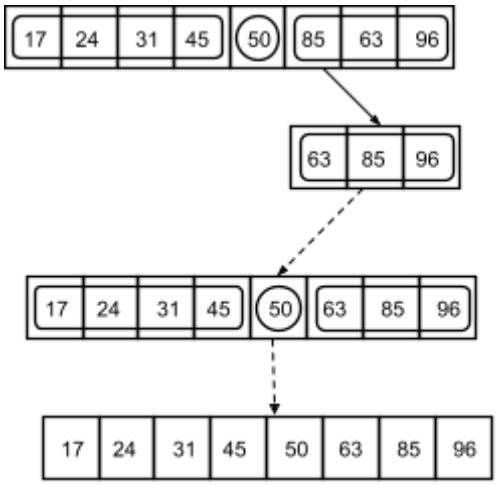
5. The same process is repeated for the `Greater` subarray.



6. The subarrays are concatenated and returned.



7. The subarray continues to concatenate and return. Eventually we get our sorted array.



4.2 In-place quicksort

Whilst a tremendously powerful method, using recursion on large data sets can be problematic. The computer can run out of memory, causing the dreaded 'stack overflow' error.

To avoid this problem, an ***in-place*** algorithm can be used. It uses only a small amount of memory in addition to that needed for the original input. The previous implementation of quicksort does not qualify as ***in-place*** because we use additional arrays `less`, `equal` and `greater` when dividing a sequence within each recursive call.

Assuming that the input sequence is given as a Python list of elements, ***in-place*** quicksort modifies the input sequence using element swapping and does not explicitly create subarrays.

Illustration of in-place quicksort:

Taking the last element as pivot.

- The divide step is performed by scanning the array using left index as *l*, which advances forward, and right index as *r*, which advances backward.

2. A swap is performed when *l* is at an element larger than the pivot and *r* is at an element smaller than the pivot.

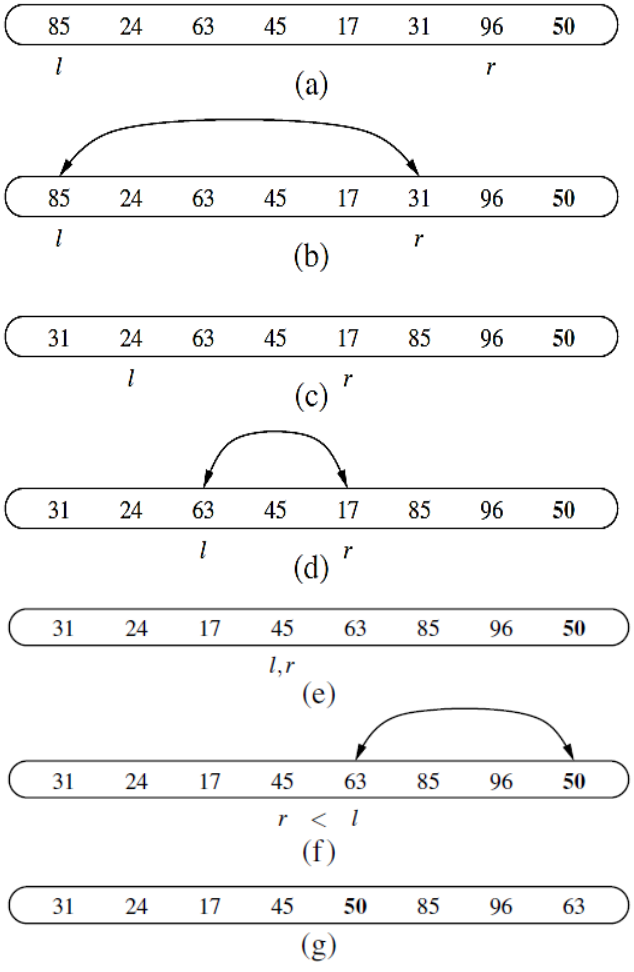
3. *l* continues advances forward and *r* advances backwards

4. Swap is performed again as *l* is at an element larger than the pivot and *r* is at an element smaller than the pivot.

5. *l* continues advances forward and *r* advances backwards

6. When the left index and the right index pass each other, a final swap with the pivot completes the divide step of in-place quick sort.

7. The algorithm completes by recurring on these two subarrays.



4.3 Time Complexity of quicksort

We would expect quicksort to run quickly, and it often does in practice. The best case for quick sort on a sequence of distinct elements occurs when the subarrays have roughly the same size. In that case, as we saw with merge sort, the tree has height O(log n) and therefore quicksort runs in O(n log n) time.

However, the worst case scenario is when the pivot chosen is always either the smallest or the largest element. This would create partitions of size n-1, causing recursive calls n-1 times. This leads to a worst case time complexity of $O(n^2)$.

Best case time complexity: $\Omega(n \log n)$
Worse case time complexity: $O(n^2)$

5 Merge sort vs Quick sort

Quicksort is an unstable sorting technique. It might change the occurrence of two similar elements in the array while sorting. Merge sort is a stable algorithm. It doesn't change the occurrence of similar elements (i.e., equal elements are ordered in the same order in the sorted list) and preserve the order in which these elements appeared in the original array.

In-place quicksort does not need additional memory space to perform sorting. Merge sort requires a temporary array to merge the sorted arrays. Hence, merge sort needs more memory space compared to in-place quicksort.