# Sorting Algorithms

## Quicksort

**Why is quicksort often the best practical choice for sorting (two reasons)?**

* It is very efficient on average as its expected running time is
* It also sorts in place.

**What is the worst-case and average case running time of quicksort?**

in worst case, average is

**What is the space complexity of QuickSort?**

**Does quicksort sort in place?**

Yes.

**What is the stability of Quicksort?**

It is not stable

**What strategy does quicksort use?**

Divide and conquer

**Describe the three step process that quicksort uses for sorting a typical subarray A[p..r]**

1. **Divide:** divide the Array A[p..r] into two (possibly empty) arrays consisting of A[p..q-1] and A[p+1..r] where each item in A[p..q-1] is less than equal to A[q] which is less than or equal to each item in A[q+1..r]. Compute the index q as part of this procedure.
2. **Conquer:** Sort the two subarrays A[p..q-1] and A[q+1..r] by recursive calls to quicksort
3. **Combine:** Because the subarrays are already sorted in-place there is no extra work required to combine the results the whole array A[p..r] is now sorted.

**What is the pseudocode for quicksort and what does the initial call look like to sort an entire array A?**

QUICKSORT (A, p, r)

if p < r

q = PARTITION (A, p, r)

QUICKSORT (A, p, q-1)

QUICKSORT (A, q+1, r)

Initial call to sort an entire array A is QUICKSORT(A, 1, A.length).

**What is the pseudocode for PARTITION?**

PARTITION (A, p, r)

x = A[r] *// select the pivot element*

i = p – 1 *// upper boundary of the less than region*

**for** j = p to r – 1 *// j defines the upper boundary of the greater than region*

**if** A[j] x *// compare the current element with the pivot*

i = i + 1 *// if current element is less than pivot, expand the less than region by incrementing i*

exchange A[i] with A[j] *// and exchange the values at i and j*

exchange A[i+1] with A[r] *// at the end, exchange the pivot with the value ahead of the less than region*

**return** i+1

**What are the four regions at the beginning of each iteration of the loop for any array index k?**

1. If (A[k] is less than the pivot element, **less than region**)
2. If (A[k] is greater than the pivot element, **greater than region**)
3. If (A[k] is equal to the pivot element, **equal to region**)
4. The indices between j and r-1 are not covered and have no relationship to the pivot. **Undefined region**.

**Identify the four regions in figure (d) and explain how PARTITION transforms the array from (d) to (e) and from (e) to (f)**

**Diagram

Description automatically generated**

The lightly shaded region is the elements which are less than the pivot (4). The heavily shaded region contain those elements which are greater than the pivot. The non-shaded region is the undefined region. j is incremented each time we loop and defines the upper boundary of the greater than region while i is only incremented when we add an item to the less than region. i defines the upper boundary of the less than region.

Partition works likes this. Select the pivot element to be the last element in the array. Then loop through each element in the array from p to r-1 and compare the currently selected element (j) to the pivot.

* If the selected element is less than or equal to the pivot, expand the less than region by incrementing i and exchange the current item with the item that is in the i slot after i has been incremented.
* If the selected element is greater than the pivot, don’t do anything. Move on to the next selected element.
* After we have iterated through the array from r to p-1, then swap the pivot with the element in the i+1 slot.

**What is the Java code implementation for quicksort and what language specific details do you need to remember?**

Particularly for the PARTITION method, remember to use a List<> for the array so that you can use the Collections.swap() method.



## Merge Sort

**What is the running time of merge sort in the worst case and average case?**

Average and worst case running time is

**What is the space complexity of merge sort?**

**What is the stability of Merge Sort?**

It is stable

**What strategy does Merge Sort use?**

Divide and conquer

**Does merge sort work in place?**

No.

**Describe the three step process that merge sort uses for sort**

1. **Divide:** divide the n-element sequence into two sequences of length n/2
2. **Conquer:** Sort the two subsequences recursively using merge sort
3. **Combine:** merge the two sorted subsequences to produce the sorted answer.

**Describe how the MERGE operation works in terms of two decks of sorted cards**

We call MERGE(A, p, q, r) where A is an array to be sorted and p, q, r are indices into the array such at The procedure assumes that the subarrays A[p..q] and A[q+1..r] are already sorted and combines them into a single sorted array.

Suppose there are two decks of cards faced up on the table. The basic step in merging the two decks of cards is to take the smaller card of the two decks and place it in the output pile, exposing a new card on the deck the smallest card was removed from. We repeat this step until one pile is empty and simply put all the other cards on top of the output pile since they are already sorted.

**What is the pseudocode for the MERGE operation?**

MERGE(A, p, q, r)

= q – p + 1 *//length of left pile*

= r – q *//length of right pile*

Let L[1..+1] and R[1.. +1] be new arrays

For i = 1 to

L[i] = A[p+i-1] *//Copy values from A to left array*

For j = 1 to

R[j] = A[q + j] *//Copy values from A to right array*

L[+1] = *// Set sentinel value*

R[] = *// Set sentinel value*

i = 1

j = 1

for k = p to r *// p to r is the total number of items to merge*

if L[i] R[j] *// if the item in the left pile is smaller than in the right*

A[k] = L[i] *//add the left item to the output pile*

i = i + 1 *// and increment the left pile index*

else A[k] = R[j] *// else add the right item to the output pile*

j = j + 1 *//and increment the right pile index*

**What is the pseudocode for the whole MERGE-SORT algorithm?**

MERGE-SORT(A, p, r)

if p < r

q =

MERGE-SORT(A, p, q)

MERGE-SORT(A, q+1, r)

MERGE(A, p, q, r)

To sort the entire array A, we make the initial call MERGE-SORT(A, 1, A.length)

**Compare and contrast MergeSort with QuickSort. Why is QuickSort generally preferable to MergeSort? In what scenarios would MergeSort be a good choice?**

Quicksort sorts in place and thus requires less space. It is also very easy to avoid QuickSort’s worse case running time of by choosing the pivot randomly. Quicksort also has a small hidden constant compared to MergeSort. If data has to be sorted on disk, you really want to use some variation of MergeSort. MergeSort is worth considering if speed is important, bad worst-case performance cannot be tolerated, and extra space is available. Studies show that QuickSort is better for smaller datasets while MergeSort is better on larger datasets.

**What is the Java code implementation for MergeSort and what language specific details do you need to remember?**

* What does stability mean for sorting algorithms? Why is it important?
  + A sorting algorithm is stable if it preserves the order of equal items. It is important because you can’t stack unstable sorts.
* Know the following sorting algorithms:
  + Quicksort
  + Merge sort
  + Insertion Sort
  + Heapsort
  + Radix/counting/bucket sort
  + Selection sort
* For each algorithm know the following:
  + Conceptually how it works
  + Code implementation
  + Time complexity
  + Space complexity
  + Stability of results
* What are the use cases for insertion, bucket, counting, and radix sort?
* How does the heap sort work?
* Why shouldn’t you use bubble sort or selection sort?