

**“AZƏRBAYCAN HAVA YOLLARI” CJSC NATIONAL AVIATION ACADEMY**

**Individual Work № 9:**

**Topic: Array sorting in Python**

**Subject: Obyektyönümlü proqramlaşdırma**

**Teacher: Mehemmed Shahmaliyev**

**Group: 1459i Student: Ilaha Aliyeva**

**Date: Signature**

**Baku 2022**

Python arrays can be sorted using different sorting algorithms, varying in their runtime and efficiency based on the algorithm chosen. We investigate some of these approaches towards sorting array elements.

Using sorted() on Python iterable objects

Python uses some extremely efficient algorithms for performing sorting. The sorted() method, for example, uses an algorithm called Timsort (which is a combination of Insertion Sort and Merge Sort) for performing highly optimized sorting.

Any Python iterable object such as a list or an array can be sorted using this method.

import array

# Declare a list type object

list\_object = [3, 4, 1, 5, 2]

# Declare an integer array object

array\_object = array.array('i', [3, 4, 1, 5, 2])

print('Sorted list ->', sorted(list\_object))

print('Sorted array ->', sorted(array\_object))

Output:

Sorted list -> [1, 2, 3, 4, 5]

Sorted array -> [1, 2, 3, 4, 5]

Implementing MergeSort and QuickSort

Here, we investigate two other commonly used Sorting techniques used in actual practice, namely the MergeSort and the QuickSort algorithms.

1. MergeSort Algorithm

The algorithm uses a bottom-up Divide and Conquer approach, first dividing the original array into subarrays and then merging the individually sorted subarrays to yield the final sorted array.

In the below code snippet, the mergesort\_helper() method does the actual splitting into subarrays and the perform\_merge() method merges two previously sorted arrays into a new sorted array.

*import array*

*def mergesort(a, arr\_type):*

*def perform\_merge(a, arr\_type, start, mid, end):*

*# Merges two previously sorted arrays*

*# a[start:mid] and a[mid:end]*

*tmp = array.array(arr\_type, [i for i in a])*

*def compare(tmp, i, j):*

*if tmp[i] <= tmp[j]:*

*i += 1*

*return tmp[i-1]*

*else:*

*j += 1*

*return tmp[j-1]*

*i = start*

*j = mid + 1*

*curr = start*

*while i<=mid or j<=end:*

*if i<=mid and j<=end:*

*if tmp[i] <= tmp[j]:*

*a[curr] = tmp[i]*

*i += 1*

*else:*

*a[curr] = tmp[j]*

*j += 1*

*elif i==mid+1 and j<=end:*

*a[curr] = tmp[j]*

*j += 1*

*elif j == end+1 and i<=mid:*

*a[curr] = tmp[i]*

*i += 1*

*elif i > mid and j > end:*

*break*

*curr += 1*

*def mergesort\_helper(a, arr\_type, start, end):*

*# Divides the array into two parts*

*# recursively and merges the subarrays*

*# in a bottom up fashion, sorting them*

*# via Divide and Conquer*

*if start < end:*

*mergesort\_helper(a, arr\_type, start, (end + start)//2)*

*mergesort\_helper(a, arr\_type, (end + start)//2 + 1, end)*

*perform\_merge(a, arr\_type, start, (start + end)//2, end)*

*# Sorts the array using mergesort\_helper*

*mergesort\_helper(a, arr\_type, 0, len(a)-1)*

*Test Case:*

*a = array.array('i', [3, 1, 2, 4, 5, 1, 3, 12, 7, 6])*

*print('Before MergeSort ->', a)*

*mergesort(a, 'i')*

*print('After MergeSort ->', a)*

**Output:**

Before MergeSort -> array('i', [3, 1, 2, 4, 5, 1, 3, 12, 7, 6])

After MergeSort -> array('i', [1, 1, 2, 3, 3, 4, 5, 6, 7, 12])

**Implementing MergeSort and QuickSort**

Here, we investigate two other commonly used Sorting techniques used in actual practice, namely the MergeSort and the QuickSort algorithms.

**1. MergeSort Algorithm**

The algorithm uses a bottom-up Divide and Conquer approach, first dividing the original array into subarrays and then merging the individually sorted subarrays to yield the final sorted array.

In the below code snippet, the mergesort\_helper() method does the actual splitting into subarrays and the perform\_merge() method merges two previously sorted arrays into a new sorted array.

*import array*

*def mergesort(a, arr\_type):*

*def perform\_merge(a, arr\_type, start, mid, end):*

*# Merges two previously sorted arrays*

*# a[start:mid] and a[mid:end]*

*tmp = array.array(arr\_type, [i for i in a])*

*def compare(tmp, i, j):*

*if tmp[i] <= tmp[j]:*

*i += 1*

*return tmp[i-1]*

*else:*

*j += 1*

*return tmp[j-1]*

*i = start*

*j = mid + 1*

*curr = start*

*while i<=mid or j<=end:*

*if i<=mid and j<=end:*

*if tmp[i] <= tmp[j]:*

*a[curr] = tmp[i]*

*i += 1*

*else:*

*a[curr] = tmp[j]*

*j += 1*

*elif i==mid+1 and j<=end:*

*a[curr] = tmp[j]*

*j += 1*

*elif j == end+1 and i<=mid:*

*a[curr] = tmp[i]*

*i += 1*

*elif i > mid and j > end:*

*break*

*curr += 1*

*def mergesort\_helper(a, arr\_type, start, end):*

*# Divides the array into two parts*

*# recursively and merges the subarrays*

*# in a bottom up fashion, sorting them*

*# via Divide and Conquer*

*if start < end:*

*mergesort\_helper(a, arr\_type, start, (end + start)//2)*

*mergesort\_helper(a, arr\_type, (end + start)//2 + 1, end)*

*perform\_merge(a, arr\_type, start, (start + end)//2, end)*

*# Sorts the array using mergesort\_helper*

*mergesort\_helper(a, arr\_type, 0, len(a)-1)*

**Test Case:**

a = array.array('i', [3, 1, 2, 4, 5, 1, 3, 12, 7, 6])

print('Before MergeSort ->', a)

mergesort(a, 'i')

print('After MergeSort ->', a)

**Output:**

Before MergeSort -> array('i', [3, 1, 2, 4, 5, 1, 3, 12, 7, 6])

After MergeSort -> array('i', [1, 1, 2, 3, 3, 4, 5, 6, 7, 12])

**2. QuickSort Algorithm**

This algorithm also uses a Divide and Conquer strategy, but uses a top-down approach instead, first partitioning the array around a pivot element (here, we always choose the last element of the array to be the pivot).

Thus ensuring that after every step, the pivot is at its designated position in the final sorted array.

After ensuring that the array is partitioned around the pivot (Elements lesser than the pivot are to the left, and the elements which are greater than the pivot are to the right), we continue applying the partition function to the rest of the array, until all the elements are at their respective position, which is when the array is completely sorted.

Note: There are other approaches to this algorithm for choosing the pivot element. Some variants choose the median element as the pivot, while others make use of a random selection strategy for the pivot.

*def quicksort(a, arr\_type):*

*def do\_partition(a, arr\_type, start, end):*

*# Performs the partitioning of the subarray a[start:end]*

*# We choose the last element as the pivot*

*pivot\_idx = end*

*pivot = a[pivot\_idx]*

*# Keep an index for the first partition*

*# subarray (elements lesser than the pivot element)*

*idx = start - 1*

*def increment\_and\_swap(j):*

*nonlocal idx*

*idx += 1*

*a[idx], a[j] = a[j], a[idx]*

*[increment\_and\_swap(j) for j in range(start, end) if a[j] < pivot]*

*# Finally, we need to swap the pivot (a[end] with a[idx+1])*

*# since we have reached the position of the pivot in the actual*

*# sorted array*

*a[idx+1], a[end] = a[end], a[idx+1]*

*# Return the final updated position of the pivot*

*# after partitioning*

*return idx+1*

*def quicksort\_helper(a, arr\_type, start, end):*

*if start < end:*

*# Do the partitioning first and then go via*

*# a top down divide and conquer, as opposed*

*# to the bottom up mergesort*

*pivot\_idx = do\_partition(a, arr\_type, start, end)*

*quicksort\_helper(a, arr\_type, start, pivot\_idx-1)*

*quicksort\_helper(a, arr\_type, pivot\_idx+1, end)*

*quicksort\_helper(a, arr\_type, 0, len(a)-1)*

Here, the quicksort\_helper method does the step of the Divide and Conquer approach, while the do\_partition method partitions the array around the pivot and returns the position of the pivot, around which we continue to recursively partition the subarray before and after the pivot until the entire array is sorted.

**Test Case:**

b = array.array('i', [3, 1, 2, 4, 5, 1, 3, 12, 7, 6])

print('Before QuickSort ->', b)

quicksort(b, 'i')

print('After QuickSort ->', b)

**Output:**

Before QuickSort -> array('i', [3, 1, 2, 4, 5, 1, 3, 12, 7, 6])

After QuickSort -> array('i', [1, 1, 2, 3, 3, 4, 5, 6, 7, 12])