46887 – COMPUTATIONAL THINKING WITH ALGORITHMS	
Document Title:	CT Project 1819S2 PYTHON
Student Name:	Gerhard van der Linde
Student Number:	G00364778
Lecturer:	Dr. Patrick Mannion

CT Project 1819S2 PYTHON

1. Introduction

The section will introduce the concept of sorting and sorting algorithms

Introduction to Sorting

Fundamentally, sorting arranges data in a sequence to make searching easier by arranging data, information or things in ascending or descending order. Sorting naturally came into existence, as humans realised the importance of searching more efficiently. There are many things in our lives that we need to search for, like a particular record in database, numbers in list, a telephone number in a directory or a page from a book or electronic document. All this would be very difficuly if the data was kept unordered and unsorted. [1)]

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms, such as search and merge algorithms, which require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output. More formally, the output of any sorting algorithm must satisfy two conditions:

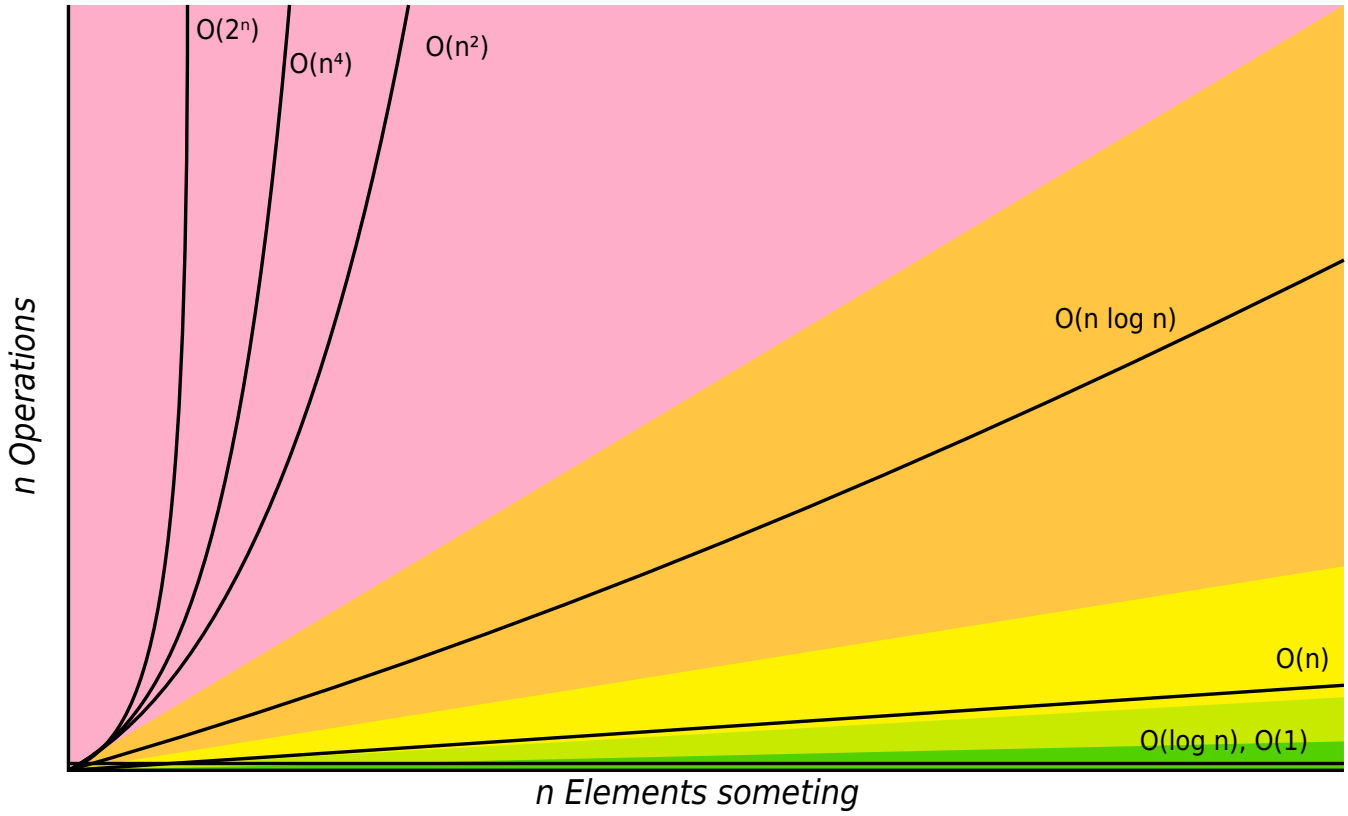
1. The output is in nondecreasing order with each element no smaller than the previous element.
2. The output is a permutation, a reordering, yet retaining all of the original elements.

Sorting is, without doubt, the most fundamental algorithmic problem that was faced in the early days on computing. In fact, most of the computer science research was centered on finding a best way to sort a set of data. There is probably a good reason to make sorting that important. Supposedly, 25% of all CPU cycles are spent sorting. Sorting is fundamental to most other algorithmic problems, for example binary search. Many different approaches lead to useful sorting algorithms, and these ideas can be used to solve many other problems. [2) 3)]

Bubble sort was analyzed as early as 1956.[4)] Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

The relevance of sorting concepts

Complexity in Time and Space



Generated using embedded svg, source in appendix

Rating	Sort Type
Excellent	1
Good	$\log n$
Fair	n
Bad	$n \log n$
Horrible	n^2 n^4 2^n

The complexity of an algorithm is a function describing the *efficiency* of the algorithm *in terms of the amount of data* the algorithm must process. There are two main complexity measures of the efficiency of an algorithm:

Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. In simple terms, we can say time complexity is the sum of the number of times each statements gets executed.

Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. When we say “this algorithm takes constant extra space,” because the amount of extra memory needed doesn’t vary with the number of items processed. [5)]

When evaluating the complexity of an algorithm, keep in mind that you must identify the most expensive computation within an algorithm to determine its classification. For example, consider an algorithm that is subdivided into two tasks, a task classified as linear followed by a task classified as quadratic. The overall performance of the algorithm must therefore be classified as quadratic.

Comparing the relative efficiency of algorithms by evaluating the running time complexity on input data size, the graphical representation above show the typically, algorithmic complexity of a number of families, i.e. the growth in its execution time with respect to increasing input size of the dataset to sort. The effect of higher order growth functions becomes more significant as the size of the input set is increased.

Performance

Performance is all about how much time, memory and disk space is consumed and is actually used when a program is run. This depends on the computer specification, the compiler used to run and build the code, as well as the efficiency of the code itself.

The table below gives a very good summary of performance and if read in conjunction with the graphical representation of various algorithm family types discussed in the time and space complexity section above.

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Simple comparison					
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Efficient comparision					
Merge Sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$	Yes
Quicksort	$n \log n$	n^2	$n \log n$	n (worst case)	No *
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Non comparison					
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Bucket Sort	$n + k$	n^2	$n + k$	$n \times k$	Yes
Hybrids					
Timsort [6]	n	$n \log n$	$n \log n$	n	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No

**The standard Quicksort algorithm is unstable, although stable variations do exist*

Complexity affects performance but not the other way around. Therefore, empirical comparisons of algorithm complexity are of limited use if we wish to draw general conclusions about the relative performance of different algorithms, as the results obtained are highly dependent on the specific platform which is used to execute the algorithm.

By knowing and understanding the performance of an algorithm under various use cases, you can determine whether an algorithm is appropriate to use in your specific requirement. This also explains how slow the program could be in any situation, and provides a lower bound on possible performance.

O(n^2) represents an algorithm whose worst case performance is directly proportional to the square of the size of the input data set.

Fundamental result in algorithm analysis is that no algorithm that sorts by comparing elements can do better than $n \log n$ performance in the average or worst cases.

In-place Sorting

In place sorting is a desirable property for sorting algorithms as it reduces complexity and memory usage.

Sorting algorithms have different memory requirements, which depend on how the specific algorithm works. A sorting algorithm is called in-place if it uses only a fixed additional amount of working space, independent of the input size. Other sorting algorithms may require additional working memory, the amount of which is often related to the size of the input set to be sorted. In-place sorting is a desirable property if the availability of memory is a concern.

Bubble sort, selection sort and insertion sort are all examples of in-place sorting algorithms.

Stable sorting

Stability in sorting algorithms indicates the ability of the algorithm to preserve the order of an already sorted input.

Put in another way, a sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.[7)]

When stability in sorting is a requirement and the comparator function determines that two elements in the original unordered collection are equal, it may be important to maintain their relative ordering in the sorted set then the final location for the pair must be maintained. Sorting algorithms that guarantee this property are termed to be stable.

Unstable sorting algorithms do not preserve this property. Using an unstable sorting algorithm means that if you sort an already sorted array, the ordering of elements which are considered equal may be altered!

Comparator functions

Sorting collections of custom objects may require a custom ordering scheme. In general, we could have some function compare a pair of values which returns:

- -1 if a < b
- 0 if a = b
- 1 if a > b

Sorting algorithms are independent of the definition of “less than” which is to be used, therefore we need not concern ourselves with the specific details of the comparator function used when designing sorting algorithms.

Sorting a collection of items according to a comparator function with some definition of “less than”, can improve the performance of search queries.

Comparison-based sorting

A sorting algorithm is called comparison-based if the way to gain information about the total order is by comparing a pair of elements at a time. Comparison-based sorts are the most widely applicable to diverse types of input data.

Bubble Sort is Comparison-based and named for the way larger values in a list “Bubble up” to the end as sorting takes place. **Bubble sort** was first analysed as early as 1956 with time complexity of n in best case, and n^2 in worst and average cases.

Another example of comparison-based sorts is **Selection Sort**.

$O\left(n \log n\right)$ time is the ideal “worst-case” scenario for a comparison-based sort, i.e $O\left(n \log n\right)$ is the smallest penalty you can hope for in the worst case. **Heapsort** has this type of behaviour.

IntroSort is also comparison-based.

Comparison-based sorts are the most widely applicable, but are limited to $n \log n$ running time in the best cases.

Non-comparison-based Sorting

Examples of non-comparison based sorts are:

- Counting sort
- Bucket sort
- Radix Sort

Non-comparison sorting algorithms can have better worst-case times

“Comparison sorts” make no assumptions about the data and compare all elements against each other, in fact, the majority of sorting algorithms work in this way.

O(n log n) time is the ideal “worst-case” scenario for a comparison-based sort, i.e O is the smallest penalty you can hope for in the worst case. $O(n)$ time is possible if we make assumptions about the data and don't need to compare elements against each other, i.e., we know that data falls into a certain range or has some distribution.

$O(n)$ is the minimum sorting time possible, since we must examine every element at least once.

Non-Comparison sorts can achieve linear n running time in the best case, but are less flexible.

2. Sorting Algorithms

Simple comparison-based sort

Bubble Sort

The table below contains the list pf comparison sorts to select from for benchmarking purposes.

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes

Selection sort is the worst overall performer and unstable with the other two very similar, so **Bubble Sort** is chosen as a classic simple comparison-based sort for this category.

Space and Time Complexity

Bubble sort is not space complex and pretty much consumes the space the size of the array being sorted, so a **space complexity of 1** can be confirmed and no growth occurs on the stack by self calling routines.

From a time complexity perspective, the algorithm essentially creates an inner loop and an outer loop pushing the biggest values to the end of the array every cycle and then reduce the loop count by the sorted group every cycle, so an array of 8 values completes 28 cycles and can save some time skipping over values already in order.

So the best case scenario is a list already sorted and the code still completes 28 cycles but no swops.

The worst case is a list sorted in reverse order, so runs 28 cycles on 8 values and completes 26 position swops.

The average case is that some values are sorted and typically fall in 50% of values being in order when working with normal distributions of random data.

So analyzing the bubble sort code and looking at datasets of increasing size the algorithm work in line with the theoretical time complexity of n^2.

How algorithm works

- using diagrams
- different example inputs

```
# Bubble Sort
verbose = False
def printArray(arr):
  return (' '.join(str(i) for i in arr))


def bubblesort(arr):
  # create an outer loop the size of the array
  for i in range(len(arr)):
    # create an inner loop thep the size of the array
    for j in range(len(arr) - i - 1):
      # starting at the left compare the two consecutive
      # elements and swop if the one to the left is bigger.
      # Skip the elemnts on the right by the outer loop
      # value to skip already sorted ones
      if arr[j] > arr[j + 1]:
        temp = arr[j]
        arr[j] = arr[j + 1]
        arr[j + 1] = temp
    # Print array after every pass to visualise progress
    if verbose ==True:
      print ("After pass " + str(i) + "  :", printArray(arr))
  return arr

if __name__ == '__main__':
  verbose = True
  arr = [10, 7, 3, 1, 9, 7, 4, 3]
  print ("Initial Array :", printArray(arr))
  arrs=bubblesort(arr)
  print (" Sorted Array :", printArray(arrs))
```

So looking at the code to the left and the diagram above, the code loops through the array with outer loop i and inner loop j that is decreasing by the position of the outerloop progress, so essentially not re-sorting already sorted values. The *if statement* compares the two consecutive values indicated by the inner loop variable j and swops the values around if bigger or proceeds to the next pair if not. This effectively results in the number 10 as shown in the example making it's way to the end of the array.

The loop then repeats to run over the array again in the second cycle and looking at the values in “pass 1” proceeds to move 7 up the line until if finds 9, drops 7 in the place of 9 and moves 9 into place in front of 10 that is not re-visted, because the array count has been reduced by 1.

And so the process repeats until we have a filly sorted array.

Efficient comparison-based sort

Merge Sort

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Merge Sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$	Yes
Quicksort	$n \log n$	n^2	$n \log n$	n (worst case)	No *

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No

**The standard Quicksort algorithm is unstable, although stable variations do exist*

Quicksort has the worst worse case efficiency and is not stable, so **Merge Sort** is the choice for this category.

Space and Time Complexity

Merge sort was proposed by John von Neumann in 1945. This algorithm exploits a recursive divide-and conquer approach resulting in a worst-case running time of $O\left(n \log\left(n\right)\right)$, the best asymptotic behavior which we have seen so far. It's best, worst, and average cases are very similar, making it a very good choice if predictable runtime is important - Merge Sort gives good all-round performance. Stable sort versions of merge Sort are particularly good for sorting data with slow access times, such as data that cannot be held in internal memory(RAM) or are stored in linked lists.

How algorithm works

Mergesort is based on the following basic idea: If the size of the list is 0 or 1, return, otherwise, separate the list into two lists of equal or nearly equal size and recursively sort the first and second halves separately. Finally, merge the two sorted halves into one sorted list.

Clearly, almost all the work is in the merge step, which should be as efficient as possible. Any merge must take at least time that is linear in the total size of the two lists in the worst case, since every element must be looked at in order to determine the correct ordering.

Referring the mergesort example to the right, the list is divided in half down the middle and repeated until left with two values 9 and 4. They are then merged in size order and the previous pair 8 and 3 is then popped of the stack and merged too in sorted order. The groups are then merged in sorted order leaving us with 3,4,8,9 and the right hand side is the processed. This starts with the processing of 1 and 2 that remains in their original order and then merged with 5 as 1,2,5 and then the last step that proceeds to merge 3,4,8,9 with 1,2,5 resulting in the final sorted list of 1,2,3,4,5,8,9.

#				
	Initial Array:	[9, 4, 8, 3, 1, 2, 5]		
1	Left:	[9]	Right:	[4]
2	After Merge:	[4, 9]		
3	Left:	[8]	Right:	[3]
4	After Merge:	[3, 8]		
5	Left:	[4, 9]	Right:	[3, 8]
6	After Merge:	[3, 4, 8, 9]		
7	Left:	[1]	Right:	[2]
8	After Merge:	[1, 2]		
9	Left:	[1, 2]	Right:	[5]
10	After Merge:	[1, 2, 5]		
11	Left:	[3, 4, 8, 9]	Right:	[1, 2, 5]
12	After Merge:	[1, 2, 3, 4, 5, 8, 9]		
13	Sorted Array:	[1, 2, 3, 4, 5, 8, 9]		

merge_sort.py

```
        # Merge sort
        debug = False
        def mergesort(arr):
          # create a top level function with a single parameter and calculate the
          # merge sort parameters so that the function is similar to the other sort
          # functions and can be called from the benchmark fucntion
          return mergesort_p(arr, 0, len(arr)-1)

        def mergesort_p(arr, i, j):
          # main merge sort function
          # this splits the array in left and right untill there are just a single pair
          # of values left and the pass the vales to merge to put together is a sorted way.
          # This section also creates the stacked space complexity by calling itself and
          # placing sections of the stack unitill it cannot be split further. Coming off
```

```
    # the stack left and right are then merged by reference.
    mid = 0
    if i < j:
      mid = int((i + j) / 2)
      # split part down the middle and stack untill one
      mergesort_p(arr, i, mid)
      # split right down the middle and stach untill one
      mergesort_p(arr, mid + 1, j)
      r_arr=merge(arr, i, mid, j)
      return r_arr

  def merge(arr, i, mid, j):
    # this code merges the referenced sections on the stack back together in sorted
    # order and return it to the calling function
    if debug == True: # display this if debugging is enabled
      print ("Left: " + str(arr[i:mid + 1]), "Right: " + str(arr[mid + 1:j + 1]))
    N = len(arr)
    temp = [0] * N
    l = i
    r = j
    m = mid + 1
    k = l
    while l <= mid and m <= r:
      if arr[l] <= arr[m]:
        temp[k] = arr[l]
        l += 1
      else:
        temp[k] = arr[m]
        m += 1
      k += 1

    while l <= mid:
      temp[k] = arr[l]
      k += 1
      l += 1
    while m <= r:
      temp[k] = arr[m]
      k += 1
      m += 1
    for i1 in range(i, j + 1):
      arr[i1] = temp[i1]
    if debug == True: # display this if debugging is enabled
      print (" After Merge: " + str(arr[i:j + 1]))
    return arr

  if __name__ == '__main__':
      # debug = True # default disabled, uncomment to enable
      # diffrent arrays for testing and comparison purposes
      arr1 = [0, 1, 2, 3, 4, 5, 6]
      arr2 = [0, 1, 2, 3, 4, 5, 6, 7]
      arr3 = [9, 4, 8, 3, 1, 2, 5]
      arr4 = [4, 8, 3, 1, 2, 5, 9]
      arr5 = [i for i in range(100,0,-1)]

      # a list of the arrays to loop through is the test
      #s = [arr1,arr2,arr3,arr4,arr5]
      s = [arr3]
      for arr in s:
        print ("Initial Array: " + str(arr))
        arrs=mergesort(arr)
        print (" Sorted Array: " + str(arrs)+"\n")
```

Non-comparison sort

Counting Sort

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Counting Sort	$n+k$	$n+k$	$n+k$	$n+k$	Yes

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Bucket Sort	$n+k$	n^2	$n+k$	$n \times k$	Yes

So again in this instance between the two, bucket sort is the worst in the worst case scenario, so between the two the clear choice for becnhmarking this category of sorts is **Counting Sort**.

Space and Time Complexity

Counting Sort was proposed by Harold H.Seward in 1954. Counting Sort allows us to do something which seems impossible - sort a collection of items in (close to) linear time. To understand how is this possible, several assumptions must be made about the types of input instances which the algorithms will have to handle for example, assume an input of size , where each item has a non-negative integer key, with a range of k(if using zero-indexing, the keys are in the range [0,...,k-1])

Best-, worst- and average-case time complexity of n +k, space complexity is also n+k The potential running time advantage comes at the cost of having an algorithm which is not a widely applicable as comparison sorts. Counting sort is stable(if implemented in the correct way!)

How algorithm works

Counting Sort procedure Determine key range k in the input array(if not already known) Initialise an array count size k, which will be used to count the number of times that each key value appears in the input instance. Initialise an array result of size n, which will be used to store the sorted output. Iterate through the input array, and record the number of times each distinct key values occurs in the input instance. Construct the sorted result array, based on the histogram of key frequencies stored in count. Refer to the ordering of keys in input to ensure that stability is preserved.

In much simpler terms, counting sort determines the biggest number in the list, it then creates a counter of the number of occurrences for every number in the list and finally creates a summed list of all the previous occurrence counted in the list. This summed indexed list is the used as the key to arrange the values in the list to their respective positions.

This is a very time and space efficient algorithm for sorting values is order.

In the first cycle the key piece of code that does all the crucial work is:

```
output[count[arr[i]]-1] = arr[i]
```

So using the example of sorting the Array of [6,7,3,1,9,7,4,3] as depicted to the right, the Counting sort algorithm sets up a counting array of 10 values, i.e. the biggest value of 9 plus 1.

The next step is then to populate the array by index with the occurrences of each value, so the values three and seven occurs twice as shown in the highlighted columns on line three in the table to the right.

The third step is then to create a summed value of all the counts. The summed values are then used as an indexed position of where the value should be placed in the final the sorted output.

So the first value in the unsorted input list, 6 is assigned to position six in the sorted array and the summed index array is decremented incase a duplicate exists. This ensures the stability of the sorting algorithm.

Initial Array	6	7	3	1	9	7	4	3		
Array index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**		
Counting Array	0	1	0	2	1	0	1	2	0	1
Counting Array summed	0	1	1	3	4	4	5	7	7	8
Output initialated	0	0	0	0	0	0	0	0		
loop 0 Output Arr	0	0	0	0	6	0	0	0		
Count Arr	0	1	1	3	4	4	4	7	7	8
loop 1 Output Arr	0	0	0	0	6	0	7	0		
Count Arr	0	1	1	3	4	4	4	6	7	8
loop 2 Output Arr	0	0	3	0	6	0	7	0		
Count Arr	0	1	1	2	4	4	4	6	7	8
loop 3 Output Arr	1	0	3	0	6	0	7	0		
Count Arr	0	0	1	2	4	4	4	6	7	8
loop 4 Output Arr	1	0	3	0	6	0	7	9		
Count Arr	0	0	1	2	4	4	4	6	7	7
loop 5 Output Arr	1	0	3	0	6	7	7	9		
Count Arr	0	0	1	2	4	4	4	5	7	7
loop 6 Output Arr	1	0	3	4	6	7	7	9		
Count Arr	0	0	1	2	3	4	4	5	7	7
loop 7 Output Arr	1	3	3	4	6	7	7	9		
Count Arr	0	0	1	1	3	4	4	5	7	7
Sorted Array	1	3	3	4	6	7	7	9		

The loop cycles through the index counter and places the next value into it's position using the same logic and moves the value pointed to by i=1 which is the value 7 into position 7 and decrement the summed index by one.

The cycles continues placing the numbers into their appropriate positions until we get to the next seven, where this will be placed in position six now due to the decrementing of the summed counting array indexer and thus preserving the order and therefore deemed to be stable.

The full implementation and test code for the counting sort follows below:

counting_sort.py

```
# Counting sort
debug = False
def printArray(arr):
  # create a nice looking output of the array seperated by spaces
  return(' '.join(str(i) for i in arr))


def countingsort(arr):
  N = len(arr)
  maxval=max(arr) + 1
  # create an array for every possible number from 0 to the biggest
  # number in the list passed in to the function
  count = [0] * maxval  # can store the count of positive numbers <= maxval
  # count and catalog the occurance count for every number in the list
  for i in range(0, N):
    count[arr[i]] += 1
  if debug == True:
    print ("         Counting Array :", printArray(count))
  # sum the count by adding the countvalues to the sum of the previous column
  # this  reates the positional index for the values
  for i in range(1, len(count)):
    count[i] += count[i - 1]
  if debug == True:
    print ("  Counting Array summed :", printArray(count))
  # initialise the output array
  output = [0] * N
  if debug == True:
    print ("       Output inisiated :", printArray(output))
  # move the input array values to the positions in the output array based on
  # the indexes created in the summed count array above. Decrement the positional index
  # value for positioning of duplicates
  for i in range(len(arr)):
    output[count[arr[i]] - 1] = arr[i]
    count[arr[i]] -= 1
    if debug == True:
      print('    loop:',i,' Output Arr :',printArray(output))
      print('              Count Arr :',printArray(count))
  if debug == True:
    print ("  Output Array arranged :", printArray(count))
    print ("Counting Array less one :", printArray(output))
  if debug == True:
    print ("         After Sorting  :", printArray(output))
  # return the sorted list to the calling function
  return output

if __name__ == '__main__':
  debug = True
  a1 = [6, 7, 3, 1, 9, 7, 4, 3]
  a2 = [7, 3, 1, 9, 7, 4, 3, 6]

  sorts=[a1,a2]
  for arr in sorts:
    print ("         Initial Array  :", printArray(arr))
    arrs=countingsort(arr)
    print ("          Sorted Array  :", printArray(arrs))
```

Quicksort Stable

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Merge Sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$	Yes
Quicksort	$n \log n$	n^2	$n \log n$	n (worst case)	No *
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No

The standard Quicksort algorithm is unstable, although **stable variations do exist*

The first free choice option select is **quicksort** but with the addition of the stability option. This allows for the comparison of quicksort I the benchmark testing and eliminates the stability issue.

Space and Time Complexity

The worst case time complexity for Quicksort in n^2 and the average case $n \log n$. This predicted behavior is in line with the observed behavior.

The space complexity for quicksort is $n\ (worst\ case\ )$ and will end up splitting and stacking the array values until all the values are left with single values to merge back, regardless of the prior state of the array, so the space complexity will always be a function of the size of the array.

How algorithm works

The quick sort algorithm selects the middle value in the array, referred to as the pivot in the code. The algorithm then places the remaining values in the list into a “smaller” or “greater” array based on their values in relation to the pivot value.

So in the sample above, with six as the pivot, the smaller list is populated with 5 and 2 and the “greater” list with 7 and 9.

Initial Array		[5, 7, 6, 9, 2]		
		smaller	pivot	bigger
Sorting steps	1	[5, 2]	6	[7, 9]
	2	[]	2	[5]
	3	[7]	9	[]
Sorted Array		[2, 5, 6, 7, 9]		

Then the “smaller and greater list is passed through the same process and form 5 and 2 , two becomes the pivot, 5 lands in greater and nothing in smaller. Similarly with 7 and 9, 9 lands in pivot and seven ends in greater than.

Everything is then concatenated when the calls return from the stack and we end up with a sorted array.

Quicksort Code

```
verbose = False
# Python code to implement Stable QuickSort.
# The code uses middle element as pivot.
def quickSort(ar):

    # Base case
    if len(ar) <= 1:
        return ar

    # Let us choose middle element a pivot
```

```
    else:
        mid = len(ar)//2
        pivot = ar[mid]

        # key element is used to break the array
        # into 2 halves according to their values
        smaller,greater = [],[]

        # Put greater elements in greater list,
        # smaller elements in smaller list. Also,
        # compare positions to decide where to put.
        for indx, val in enumerate(ar):
            if indx != mid:
                if val < pivot:
                    smaller.append(val)
                elif val > pivot:
                    greater.append(val)

                # If value is same, then considering
                # position to decide the list.
                else:
                    if indx < mid:
                        smaller.append(val)
                    else:
                        greater.append(val)
        if verbose == True:
            print('{}<--{}-->{}'.format(smaller,pivot,greater))
        return quickSort(smaller)+[pivot]+quickSort(greater)

# Driver code to test above
if __name__ == '__main__':
    verbose=True
    # ar = [1, 3, 5, 9, 8, 3, 4, 6, 7]
    ar = [5, 7 ,6 ,9, 2]
    print('Initial Array: ',ar)
    sortedAr = quickSort(ar)
    print(' Sorted Array: ',sortedAr)
```

Hybrid Sort

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Timsort [8]	n	$n \log n$	$n \log n$	n	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No

For the second free choice category a hybrid sort is selected for a best case benchmark comparison. Timsort is stable and marginally better so in keeping with the stability criteria, **Timsort** is the clear choice.

As an addede benefit, Timsort is the native implementation in the standard Python sort features. [9]

Timsort is a sorting algorithm that is efficient for real-world data. Tim Peters created Timsort for the Python programming language in 2001. Timsort first analyses the list it is trying to sort and then chooses an approach based on the analysis of the list.

Since the algorithm has been invented it has been used as the default sorting algorithm in Python, Java, the Android Platform, and in GNU Octave.

Timsort's big O notation is $O\left(n \log\left(n\right)\right)$.

Timsort's sorting time is the same as Mergesort, which is faster than most of the other sorts you might know. Timsort actually makes use of Insertion sort and Mergesort.

Peters designed Timsort to use already-ordered elements that exist in most real-world data sets. It calls these already-ordered elements "natural runs". It iterates over the data collecting the elements into runs and simultaneously merging those runs together into one.

To run Timsort in Python code simply call **list.sort()** or **sorted(list)**. [10)]

Space and Time Complexity

How algorithm works

- using diagrams
- different example inputs

The original Timsort code is quite intimidating and lengthy as there are lots of bits to it, but when looking at it I'm detail it is essentially merger sort with a lot of variations applied to it. [11)]

The key aspects exploited in Timsort is to improve merge sort in the following key areas and approaches.

1. Can we make merges faster?
2. Can we perform fewer merges?
3. Are there cases where we're actually better off doing something different and not using mergesort?

Timsort implemented in Python Code

This is not an exact implementation of Timsort, because Timsort relies on *natural* merge sort that exploits naturally ascending and descending **runs** in the data to speed up the process, however the code sample provides great insight into the inner workings and mutual dependencies in the sorting algorithm.[12)]

```
# Python3 program to perform TimSort.
RUN = 32

# This function sorts array from left index to
# to right index which is of size atmost RUN
def insertionSort(arr, left, right):

    for i in range(left + 1, right+1):

        temp = arr[i]
        j = i - 1
        while arr[j] > temp and j >= left:

            arr[j+1] = arr[j]
            j -= 1

        arr[j+1] = temp

# merge function merges the sorted runs
def merge(arr, l, m, r):

    # original array is broken in two parts
    # left and right array
    len1, len2 = m - l + 1, r - m
    left, right = [], []
    for i in range(0, len1):
        left.append(arr[l + i])
```

```
    for i in range(0, len2):
        right.append(arr[m + 1 + i])

    i, j, k = 0, 0, l
    # after comparing, we merge those two array
    # in larger sub array
    while i < len1 and j < len2:

        if left[i] <= right[j]:
            arr[k] = left[i]
            i += 1

        else:
            arr[k] = right[j]
            j += 1

        k += 1

    # copy remaining elements of left, if any
    while i < len1:

        arr[k] = left[i]
        k += 1
        i += 1

    # copy remaining element of right, if any
    while j < len2:
        arr[k] = right[j]
        k += 1
        j += 1

# iterative Timsort function to sort the
# array[0...n-1] (similar to merge sort)
def timSort(arr, n):

    # Sort individual subarrays of size RUN
    for i in range(0, n, RUN):
        insertionSort(arr, i, min((i+31), (n-1)))

    # start merging from size RUN (or 32). It will merge
    # to form size 64, then 128, 256 and so on ....
    size = RUN
    while size < n:

        # pick starting point of left sub array. We
        # are going to merge arr[left..left+size-1]
        # and arr[left+size, left+2*size-1]
        # After every merge, we increase left by 2*size
        for left in range(0, n, 2*size):

            # find ending point of left sub array
            # mid+1 is starting point of right sub array
            mid = left + size - 1
            right = min((left + 2*size - 1), (n-1))

            # merge sub array arr[left.....mid] &
            # arr[mid+1....right]
            merge(arr, left, mid, right)

        size = 2*size

# utility function to print the Array
def printArray(arr, n):
```

```
    for i in range(0, n):
        print(arr[i], end = " ")
    print()


# Driver program to test above function
if __name__ == "__main__":

    arr = [5, 21, 7, 23, 19]
    n = len(arr)
    print("Given Array is")
    printArray(arr, n)

    timSort(arr, n)

    print("After Sorting Array is")
    printArray(arr, n)

# This code is contributed by Rituraj Jain
```

3. Implementation & Benchmarking

Load the separate sorting libraries into memory using from imports and proceed to create a list of orts to call in the code below and saving the averaged times to file.

The commented benchmarking code in python is as below.

```
from time import time
from s01_bubblesort import bubblesort as bs
from s04_merge_sort import mergesort as ms
from s07_counting_sort import countingsort as cs
from s05a_quick_sort_stable import quickSort as qs
from s09_timsort import timsort as ts
from numpy.random import randint
import numpy as np


if __name__ == '__main__':
    # create a dictionary of sort types to loop through
    types={bs:'  Bubble Sort',ms:'   Merge Sort',cs:'Counting Sort',qs:'   Quick Sort',ts:'
Timsort'}
    headers='ArraySize'
    for key in types:
        headers+=', '+ types[key].strip()
    print(headers)
    valstrArr=[] ##store the rows for saving to a csv file
    # loop through the values in the array t and use the value to create an array of random
values
    #for t in (10,20,30,50,100,200,300,500):
    #for t in (100,200,300,400,500,600,700,800,900,1000):
    for t in (100, 250, 500, 750, 1000, 1250, 2500, 3750, 5000, 6250, 7500, 8750, 10000):
    #for t in range(1000,30001,1000):
        test = randint(1,t*2,t)
        #print('  Input len,min,max: ', len(test),',',min(test),',',max(test))
        funcAvgTim=[] # create an emty array for storing the average times
        # read the sort types to complete from the types list
        for sortfunc, funcname in types.items():
            times=[] # create an empty list to store the times in fr the cycles
            for i in range(10): # repeat tests to get better average
                #print('         Test Cycle: ',i)
                start=time() # mark the start time of the test
                arr=test.copy() #copy the test array created above and re-use for every
following test
                #print('  input: ', arr)
                ret=sortfunc(arr) # run the sort function and return the sorted result
                #print(' sorted: ', ret)
                end=time() # record the end time of the test
                #print(funcname,' time: ',(end-start)*1000)
                times.append((end-start)*1000) # append the test time to the times list
            print (t, funcname, np.average(times)) # prnt the average time to the screen
            funcAvgTim.append(np.average(times)) # save the avarage time to a list for saviing
to file
        valstr='' # initiate a val string for writing to file
        sep='' # nitiate the seperator for seperation of the values
        for val in funcAvgTim:
            valstr+=sep+str(val) # ppend the values to the string to write out to files
            sep=', ' # add the seperator
        #print('{}{}{}'.format(t,sep,valstr))
        valstrArr.append('{}{}{}'.format(t,sep,valstr)) #append the strin to a list for saving
below
```

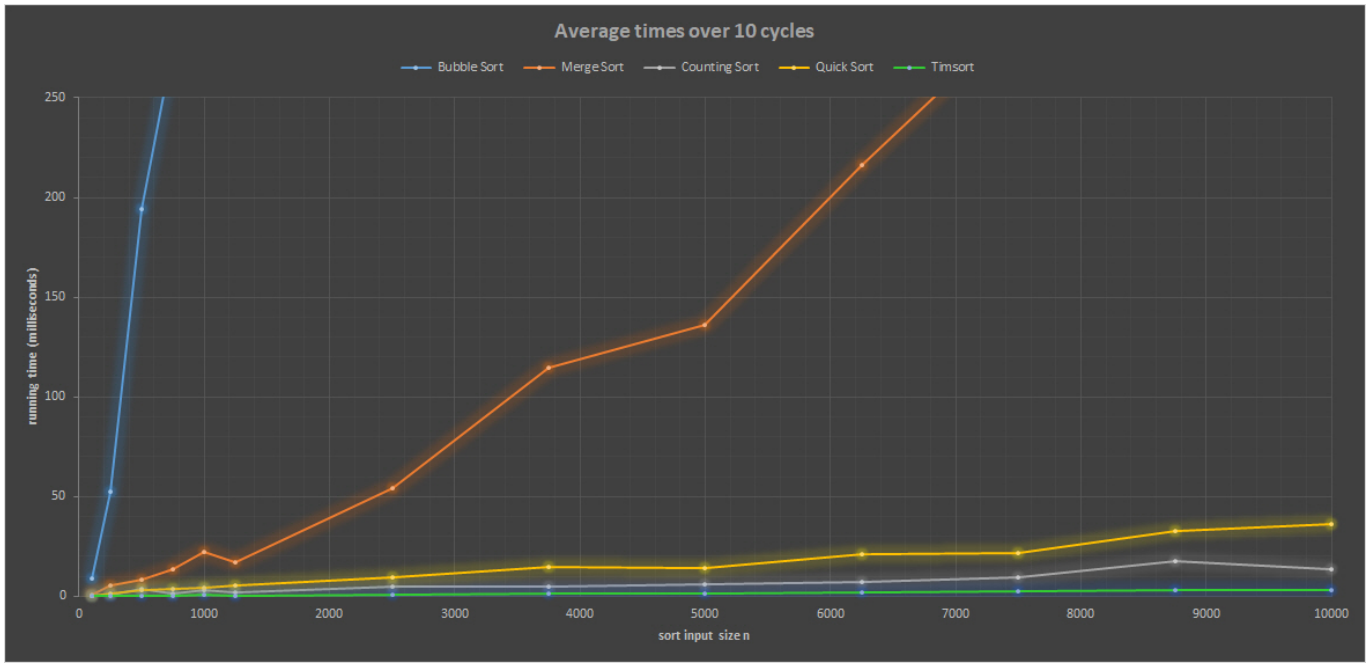
```
    for line in valstrArr:
        #print(line) # print the result to screen for validation purposes
        pass

    #write the cycle test resuts to file
    with open('sort_cycle_times_100-10k_by_10cycles.csv', 'w') as file:
        file.write(headers+'\n') # write a header line
        for line in valstrArr: # loop through the lines in the list
            file.write(line+'\n') # and write them out to the file
```

The benchmarking code essentially complete the benchmark process by employing three cascaded loops. The outer loop determines the varying array sizes from small to big.

The next loop then cycles through the five sorting functions and the last loop repeats every sorting function ten times. During the ten inner loop cycles the sort functions are benchmark against the timers and averaged over the ten cycles. All the test data is stored in variables during the tests and output to a comma separated files for tabling and graphing the results.

The table below is then generated by transposing the output of the csv files.

ArraySize:	100	250	500	750	1000	1250	2500	3750	5000	6250	7500	8750	10000
Bubble Sort	8.924	52.644	194	275	429	625	1778	4303	7240	10485	15131	20620	26922
Merge Sort	1.103	5.362	8.569	13.815	22.296	17.031	54.144	114.505	136.173	216.461	290.058	401.269	479.006
Counting Sort	0.176	0.702	3.811	1.404	2.908	1.905	4.903	4.970	6.009	7.211	9.425	17.941	13.680
Quick Sort	0.301	1.404	3.102	4.011	4.401	5.515	9.726	14.539	14.234	21.142	21.962	32.988	36.161
Timsort	0.401	0.191	0.498	0.295	0.611	0.501	0.682	1.265	1.534	2.006	2.384	3.185	3.030



Appendix A - SVG Source for sorting complexity graphs

```
<svg xmlns="http://www.w3.org/2000/svg" id="chart" width="800" height="500">
            <!-- horrible region -->
            <path fill="#ffaec9" d="M 50 450 L 50 0 L 800 0 L 800 440 Z" />
            <!-- bad region -->
            <path fill="#ffc543" d="M 50 450 L 800 0 L 800 450 Z" />
            <!-- fair region -->
            <path fill="#fff200" d="M 50 450 L 800 450 L 800 330 Z" />
            <!-- good region -->
            <path fill="#c8ea00" d="M 50 450 L 800 450 L 800 407 Z" />
            <!-- excellent region -->
            <path fill="#53d000" d="M 50 450 L 800 450 L 800 433 Z" />
            <!-- axes -->
            <path fill="transparent" stroke="black" stroke-width="2" d="M 50 0 L 50 450 L 800
450" />

            <path fill="transparent" stroke="black" stroke-width="2" d="M 50 446 L 800 446" />
            <text style="font-size:16px;" fill="black" x="680" y="438">O(log n), O(1)</text>
            <path fill="transparent" stroke="black" stroke-width="2" d="M 50 450 L 800 400" />
            <text style="font-size:16px;" fill="black" x="760" y="390">O(n)</text>
            <path fill="transparent" stroke="black" stroke-width="2" d="M 50 450 Q 400 350 800
150" />
            <text style="font-size:16px;" fill="black" x="630" y="190">O(n log n)</text>
            <path fill="transparent" stroke="black" stroke-width="2" d="M 50 450 Q 180 380 250
0" />
            <text style="font-size:16px;" fill="black" x="260" y="30">O(n&#178;)</text>
            <path fill="transparent" stroke="black" stroke-width="2" d="M 50 450 C 100 430 120
350 120 0" />
            <text style="font-size:16px;" fill="black" x="125" y="20">O(2&#x207F;)</text>
            <path fill="transparent" stroke="black" stroke-width="2" d="M 50 450 C 105 420 150
350 180 0" />
            <text style="font-size:16px;" fill="black" x="180" y="40">O(n&#x2074;)</text>
            <text style="dominant-baseline: middle; text-anchor: middle; font-size:20px; color:
#555; font-style: italic;" fill="black" transform="translate(30 230) rotate(-90)" x="0" y="0">n
Operations</text>
            <text style="dominant-baseline: middle; text-anchor: middle; font-size:20px; color:
#555; font-style: italic;" fill="black" transform="translate(420 470)" x="0" y="0">n Elements
someting</text>
</svg>
```

References

1)
https://www.studytonight.com/data-structures/introduction-to-sorting
2)
http://www.cs.cmu.edu/~clo/www/CMU/DataStructures/Lessons/lesson8_1.htm
3)
https://en.wikipedia.org/wiki/Sorting_algorithm
4)
https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2011/06/Sorting-Algorithm.pdf
5)
https://medium.com/@info.gildacademy/time-and-space-complexity-of-data-structure-and-sorting-algorithms-588a57edf495
6) , 8) , 9)
https://corte.si/posts/code/timsort/index.html

7)

https://www.geeksforgeeks.org/stability-in-sorting-algorithms/

10)

https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399?gi=a51e1047735f

11)

https://www.drmaciver.com/2010/01/understanding-timsort-1adaptive-mergesort/

12)

https://www.geeksforgeeks.org/timsort/

Last update: **2019/05/09 08:23**