Comparison-based Sorting Algorithms

PROJECT DOCUMENTATION REPORT

PROJECT 1

ITCS 6114 – Algorithm and Data Structures

DEPARTMENT OF COMPUTER SCIENCE

SUBMITTED TO
Dewan T. Ahmed, Ph.D.

SUBMITTED BY:

Urma Haldar
801074045

Sahil Sood
801082267

Table of Contents

INSERTION SORT

1.1 Understanding

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**. This algorithm is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where **n** is the number of items.

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

1.2 Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 – If it is the first element, it is already sorted. return 1;
Step 2 – Pick next element
Step 3 – Compare with all elements in the sorted sub-list
Step 4 – Shift all the elements in the sorted sub-list that is greater than the
         value to be sorted
Step 5 – Insert the value
Step 6 – Repeat until list is sorted
```

MERGE SORT

2.1 Understanding

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

To understand merge sort, we take an unsorted array as the following –

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted

positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this –

Now we should learn some programming aspects of merge sorting.

2.2 Algorithm

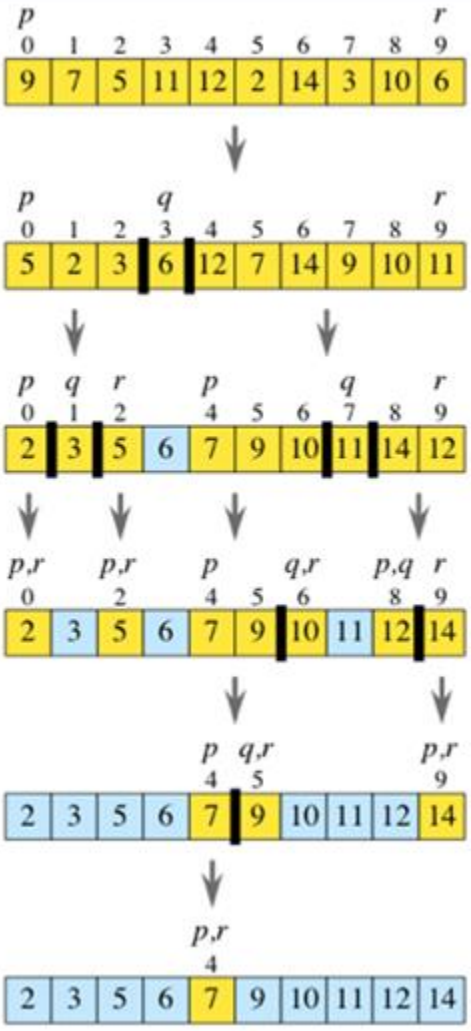
Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

```
Step 1 – if it is only one element in the list it is already sorted, return.
Step 2 – divide the list recursively into two halves until it can no more be divided.
Step 3 – merge the smaller lists into new list in sorted order.
```

IN-PLACE QUICKSORT

3.1 Understanding

Another divide and conquer algorithm, but with the hard work, comparison, done in the divide stage. storage requirements makes in-place techniques attractive. *Expected* running time is O(*n lg n*), *worst* case is O(n^2).



3.2 Algorithm

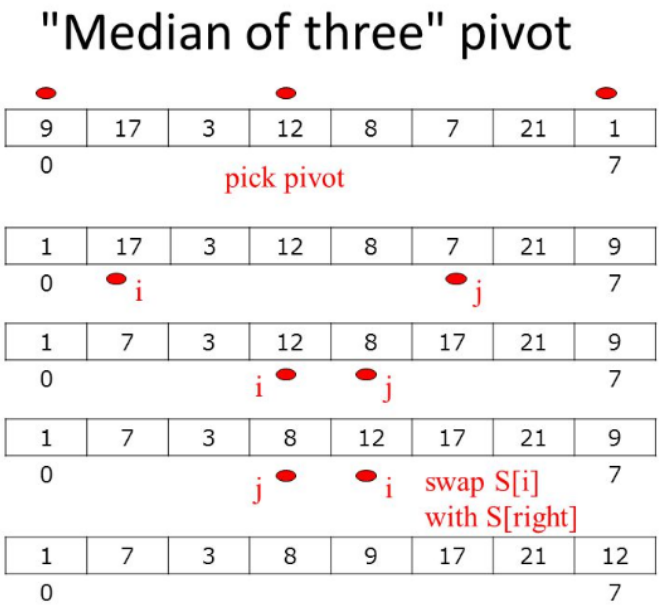
```
QuickSort( double[] a )
  {
    if ( a.length ≤ 1 )
      return;    // Don't need sorting

    Select a pivot;  // It's usually the last elem in a[]

    Partition a[] in 2 halves:
      left[]: elements ≤ pivot
          right[]: elements > pivot;

    Sort left[];
    Sort right[];

    Concatenate: left[] pivot right[]
  }
```

MODIFIED QUICKSORT

4.1 Understanding

For numbers less than 10 we call insertion sort (discussed before), for greater values we call median of 3 quick sort:

The median of three has you look at the first, middle and last elements of the array, and choose the median of those three elements as the pivot.To get the "full effect" of the median of three, it's also important to *sort* those three items, not just use the median as the pivot -- this doesn't affect what's chosen as the pivot in the current iteration, but can/will affect what's used as the pivot in the next recursive call, which helps to limit the bad behavior for a few initial orderings (one that turns out to be particularly bad in many cases is an array that's sorted, except for having the smallest element at the high end of the array (or largest element at the low end).



4.1 Algorithm

```
# Get the median of three of the array, changing the array as you do.
# arr = Data Structure (List)
# left = Left most index into list to find MOT on.
# right = Right most index into list to find MOT on

def MedianOfThree(arr, left, right):
    mid = (left + right)/2
    if arr[right] < arr[left]:
        Swap(arr, left, right)
    if arr[mid] < arr[left]:
        Swap(arr, mid, left)
    if arr[right] < arr[mid]:
        Swap(arr, right, mid)
    return mid

# Generic Swap for manipulating list data.
def Swap(arr, left, right):
    temp = arr[left]
    arr[left] = arr[right]
    arr[right] = temp
```

CODE APPROACH

5.1 Approach Towards Code

We have generated random numbers for our codes, so when we give the length of the dataset [500, 1000, 2000, 5000 and so on] and randomly numbers get generated. We took same dataset for all codes, and so we took one program and run the functions one by one over the same set of random numbers generated. For insertion sort, application is simple, just swapping numbers one by one. For Merge sort, we are splitting the data set into two in every set and then merging them while sorting, each step shown. For In place quick sort, we are repeatedly swapping the numbers based on the random pivot chosen. And in Modified Quick sort, we are implementing insertion sort for subarrays and arrays <=10 and for the rest larger numbers we are implementing median of 3 pivot quick Sort.
We have then done a comparison using graphs and noted conclusion.

5.2 Challenges Faced And Solution Proposed

The major challenge we faced was for In Place Quick Sort, where for huge number dataset like 20000, the program was failing. We did try using set/get recursionlimit(), but for this, the kernel died. Hence, we changed our approach to non recursive method, which works absolutely fine with huge numbers like 70000 even. This could have been done using other methods as well, like handling the smaller subarray separately so that it doesn't go on a loop or on reducing the number of recursions, as the issue is too many recursions taking place.

CODE FOR SORTING ALGORITHMS

6.1 Random Order

```
import random
import time


######### INSERTION SORT RANDOM #################
def insertionSort(alist):
   for index in range(1,len(alist)):

     currentvalue = alist[index]
     position = index

     while position>0 and alist[position-1]>currentvalue:
        alist[position]=alist[position-1]
        position = position-1

     alist[position]=currentvalue
#########################################################


######### MERGE SORT RANDOM ############################


def mergeSort(alist):
```

```
#print("Splitting ",alist)
if len(alist)>1:
    mid = len(alist)//2
    lefthalf = alist[:mid]
    righthalf = alist[mid:]

    mergeSort(lefthalf)
    mergeSort(righthalf)

    i=0
    j=0
    k=0
    while i < len(lefthalf) and j < len(righthalf):
        if lefthalf[i] < righthalf[j]:
            alist[k]=lefthalf[i]
            i=i+1
        else:
            alist[k]=righthalf[j]
            j=j+1
        k=k+1

    while i < len(lefthalf):
        alist[k]=lefthalf[i]
        i=i+1
        k=k+1
```

```
        while j < len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
    #print("Merging ",alist)
##############################################################


########## IN-PLACE QUICK SORT RANDOM ########################


def quickSort(items):
    def sort(lst, l, r):
        # base case
        if r <= l:
            return
        # choose random pivot
        pivot_index = random.randint(l, r)
        # move pivot to first index
        lst[l], lst[pivot_index] = lst[pivot_index], lst[l]
        # partition
        i = l
        for j in range(l+1, r+1):
            if lst[j] < lst[l]:
                i += 1
```

```
            lst[i], lst[j] = lst[j], lst[i]
        # place pivot in proper position
        lst[i], lst[l] = lst[l], lst[i]

        # sort left and right partitions
        sort(lst, l, i-1)
        sort(lst, i+1, r)

    if items is None or len(items) < 2:
        return
    sort(items, 0, len(items) - 1)
#########################################################################

###### MODIFIED QUICK SORT RANDOM ######################
def quickSortM(alist):
   quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
   if first<last:

      splitpoint = partition(alist,first,last)

      quickSortHelper(alist,first,splitpoint-1)
      quickSortHelper(alist,splitpoint+1,last)
```

```
def partition(alist,first,last):
    pivotindex = median(alist, first, last, (first + last) // 2)
    alist[first], alist[pivotindex] = alist[pivotindex], alist[first]
    pivotvalue = alist[first]

    leftmark = first
    rightmark = last

    done = False
    while not done:

        while leftmark <= rightmark and \
                alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1

        while alist[rightmark] >= pivotvalue and \
                rightmark >= leftmark:
            rightmark = rightmark -1

        if rightmark < leftmark:
            #if ((leftmark<=10)|(rightmark<=10)):  # we check for subarray<=10
            #if true call for insertion sort
                #print("Now calling insertion sort as last subset has 10 numbers",
                leftmark,rightmark)
```

```
            #insertionSort(alist)
        done = True
    else:
        temp = alist[leftmark]
        alist[leftmark] = alist[rightmark]
        alist[rightmark] = temp

  temp = pivotvalue
  alist[alist.index(pivotvalue)] = alist[rightmark]
  alist[rightmark] = temp


  return rightmark

def median(a, i, j, k):
 if a[i] < a[j]:
   return j if a[j] < a[k] else k
 else:
   return i if a[i] < a[k] else k
##########################################################


alist = []
print("Please enter the random number range to sort")
new=input()
```

```
print('\n---------------------------------------------------------------------')
numbers = random.sample(range(1, 100000), int(new))
alist = numbers
##### CALLING INSERTION SORT RANDOM ##########
print('Sorting using Insertion Sort\n')
t1=time.time()
insertionSort(alist)
t2=time.time()
print(alist)
print ("The algorithm took " + str((t2-t1) * 1000)  + " ms of time.")
print('\n---------------------------------------------------------------------')

######## CALLING MERGE SORT RANDOM ##########################
print('Sorting using Merge Sort\n')
t3=time.time()
mergeSort(alist)
t4=time.time()
print('Using the same Dataset...')
print ("The algorithm took " + str((t4-t3) * 1000)  + " ms of time.")
print('\n---------------------------------------------------------------------')
##### CALLING IN-PLACE QUICK SORT RANDOM ####################
print('Sorting using In-Place Quick Sort\n')
t5=time.time()
quickSort(numbers)
t6=time.time()
```

```
print('Using the same Dataset...')

print ("The algorithm took " + str((t6-t5) * 1000)  + " ms of time.")

print('\n----------------------------------------------------------------------')

######## CALLING MODIFIED QUICK SORT ########################

print('Sorting using Modified Quick Sort\n')

if(int(new)<=10):

   print("Since number of elements to be sorted is lesser than 10, insertion sort needs to be implemented")

  t7=time.time()

   insertionSort(alist)

else:

   print("Since number of elements to be sorted is greater than 10, quick sort needs to be implemented")

  t7=time.time()

   quickSortM(alist)

t8=time.time()

print('Using the same Dataset...')

print ("The algorithm took " + str((t8-t7) * 1000)  + " ms of time.")

print('\n----------------------------------------------------------------------')
```

Sample Output For Random Numbers-

```
Please enter the random number range to sort

10

---------------------------------------------------------------------

Sorting using Insertion Sort

[11028, 11037, 14381, 16291, 29153, 38088, 43669, 51606, 92965, 99861]

The algorithm took 0.0980201721191406 ms of time.

---------------------------------------------------------------------

Sorting using Merge Sort

Using the same Dataset...

The algorithm took 0.9962787628173828 ms of time.

---------------------------------------------------------------------

Sorting using In-Place Quick Sort

Using the same Dataset...

The algorithm took 0.9962787628173828 ms of time.

---------------------------------------------------------------------

Sorting using Modified Quick Sort

Since number of elements to be sorted is lesser than 10, insertion sort needs to be
implemented

Using the same Dataset...

The algorithm took 0.9962787628173828 ms of time.
```

6.2 Sorted/Reverse Order

The code for the Sorted/Reverse number input remains same as for random numbers, just that one statement changes:
sort=sorted(numbers) - for sorted input (we pass sorted list using function sorted)
sort=sorted(numbers,type=int,reverse=True)- for reverse sorted input

Sample Output for Sorted Input-

```
Please enter the random number range to sort

5

---------------------------------------------------------------------

Sorting using Insertion Sort

[16678, 42594, 50988, 77250, 95479]

The algorithm took 0.00 ms of time.

---------------------------------------------------------------------

Sorting using Merge Sort

Using the same Dataset...

The algorithm took 0.097875213623047 ms of time.

---------------------------------------------------------------------

Sorting using In-Place Quick Sort

Using the same Dataset...

The algorithm took 0.007875213623047 ms of time.

---------------------------------------------------------------------

Sorting using Modified Quick Sort

Since number of elements to be sorted is lesser than 10, insertion sort needs to be
implemented

Using the same Dataset...

The algorithm took 0.00 ms of time.
```

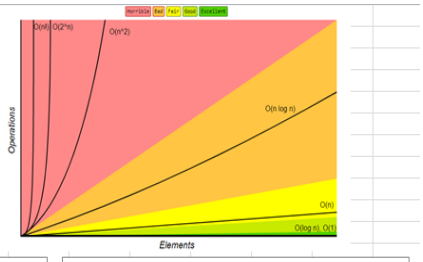
Sample Output For Reverse Sorted Input:

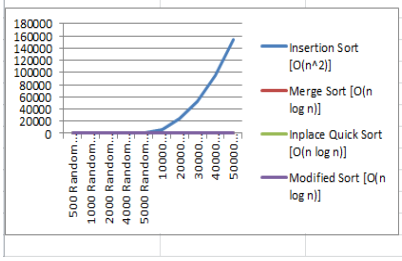
```
Please enter the random number range to sort

10

---------------------------------------------------------------------

Sorting using Insertion Sort

[11882, 18059, 19903, 39291, 46880, 59744, 73508, 76453, 78481, 99183]

The algorithm took 0.9834766387939453 ms of time.

---------------------------------------------------------------------

Sorting using Merge Sort

Using the same Dataset...

The algorithm took 0.99834766387939453 ms of time.

---------------------------------------------------------------------

Sorting using In-Place Quick Sort

Using the same Dataset...

The algorithm took 0.99834766387939453 ms of time.

---------------------------------------------------------------------

Sorting using Modified Quick Sort

Since number of elements to be sorted is lesser than 10, insertion sort needs to be
implemented

Using the same Dataset...

The algorithm took 0.90834766387939453 ms of time.
```

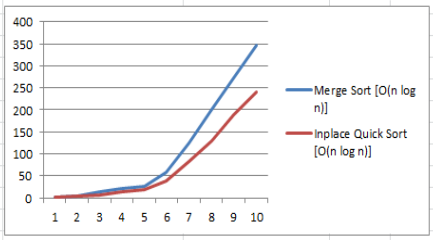
GRAPHS:

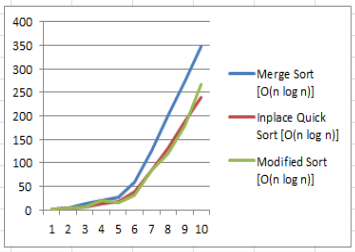
Graph For Random Input:

Time in ms	Insertion Sort [O(n^2)]	Merge Sort [O(n log n)]	Inplace Quick Sort [O(n log n)]	Modified Sort [O(n log n)]
500 Random Numbers	13.09	1.99	0.99	0.99
1000 Random Numbers	47.97	3.99	2.99	3.01
2000 Random Numbers	224.87	13.99	6.99	5.98
4000 Random Numbers	865.48	19.98	13.99	19.93
5000 Random Numbers	1326.22	26.98	17.98	15.86
10000 Random Numbers	5758.67	58.96	38.95	30.6
20000 Random Numbers	23737.623	124.91	83.93	85.5
30000 Random Numbers	53047.538	199.88	129.94	120.05
40000 Random Numbers	94528.768	272.82	188.86	178.56
50000 Random Numbers	154252.516	346.777	239.85	267.79



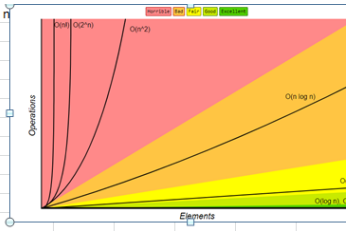


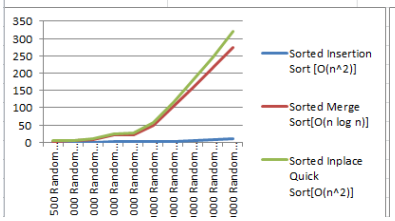


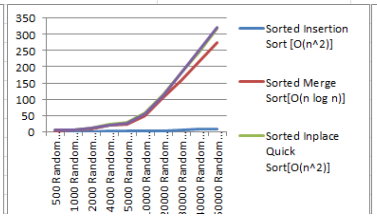


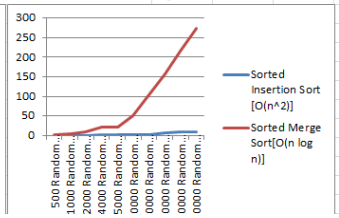
Graph for Sorted Input:

Time in ms	Sorted Insertion Sort [O(n^2)]	Sorted Merge Sort[O(n log n)]	Sorted Inplace Quick Sort[O(n^2)]	Sorted Modified Sort[O(n log n
500 Random Numbers	0	1.01	4.99	4.01
1000 Random Numbers	0.09	3.99	5.01	5.99
2000 Random Numbers	0	7.99	10.97	11
4000 Random Numbers	0.99	20.988	23.98	20.94
5000 Random Numbers	0.99	21.98	27.98	25.22
10000 Random Numbers	1.99	49.97	56.96	53.67
20000 Random Numbers	2.99	103.94	113.93	110.67
30000 Random Numbers	5.99	157.88	181.89	180.55
40000 Random Numbers	7.99	216.87	245.87	250.16
50000 Random Numbers	8.99	273.84	319.83	320.58



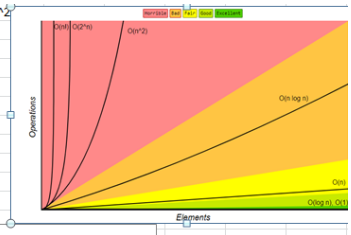


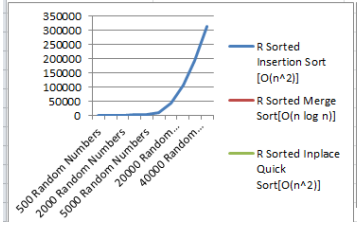


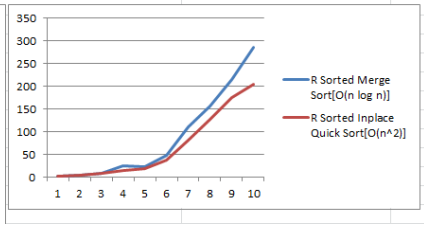


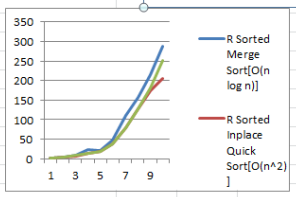
Graph For Reverse Sorted Input:

Time in ms	R Sorted Insertion Sort [O(n^2)]	R Sorted Merge Sort[O(n log n)]	R Sorted Inplace Quick Sort[O(n^2)]	R Sorted Modified Sort[O(n^2
500 Random Numbers	20.98	0.99	1.99	1.011
1000 Random Numbers	96.42	2.99	3.01	4.99
2000 Random Numbers	409.74	7.99	6.97	8.98
4000 Random Numbers	1670.04	23.98	12.99	13.94
5000 Random Numbers	2620.49	22	17.99	18.87
10000 Random Numbers	10634.88	48.17	36.97	38.98
20000 Random Numbers	43608.97	109.91	79.95	80.11
30000 Random Numbers	107317.46	155.91	126.94	128.16
40000 Random Numbers	193583.98	214.89	173.89	180.68
50000 Random Numbers	314768.89	286.81	204.88	250.44









A QUICK OBSERVATION FROM GRAPHS:

We have taken a standard graph showing all common comparisons, O(n), O(n log n) and so on. Based on that we can see the performance of each sorting type, the best and the worst. We can see difference based on input size as well. Runtimes may vary system to system.

We can clearly see from the graphs, how, quick sort is the best sorting technique, even on keeping the worst case into picture. Lets discuss these in detail in the next section of our observation.

CONCLUSION

Insertion Sort

The **worst** case for Insertion Sort occurs when the array is in reverse order. To insert each number, the algorithm will have to shift over that number to the beginning of the array. Therefore, The maximum number of comparisons for an insertion sort will be the sum of the first n−1 integers which is O(n2).

In the **best case**, where the array was already sorted, no element will need to be moved, so the algorithm will just run through the array once and return the sorted array. The running time would be directly proportional to the size of the input, so we can say it will take O(n) time.

On **average**, half the elements in a list A1 ... Aj are less than element Aj+1, and half are greater. Therefore, the algorithm compares the (j + 1)th element to be inserted on the average with half the already sorted sub-list, so tj = j/2. Working out the resulting average-case running time yields a quadratic function of the input size, just like the worst-case running time.

worst case: reverse-ordered elements in array.

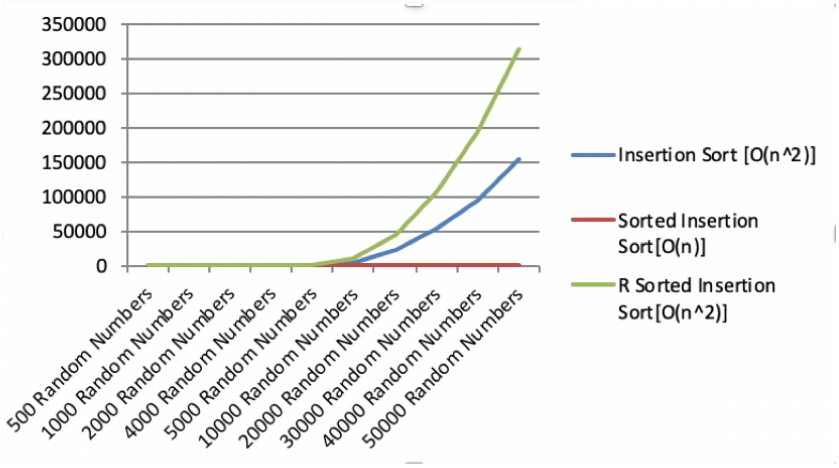
$$\sum_{i=1}^{N-1} i = 1+2+3+\ldots+(N-1) = \frac{(N-1)N}{2}$$
$$= O(N^2)$$

best case: array is in sorted ascending order.

$$\sum_{i=1}^{N-1} 1 = N-1 = O(N)$$

average case: each element is about halfway in order.

$$\sum_{i=1}^{N-1} \frac{i}{2} = \frac{1}{2}(1+2+3\ldots+(N-1)) = \frac{(N-1)N}{4}$$
$$= O(N^2)$$



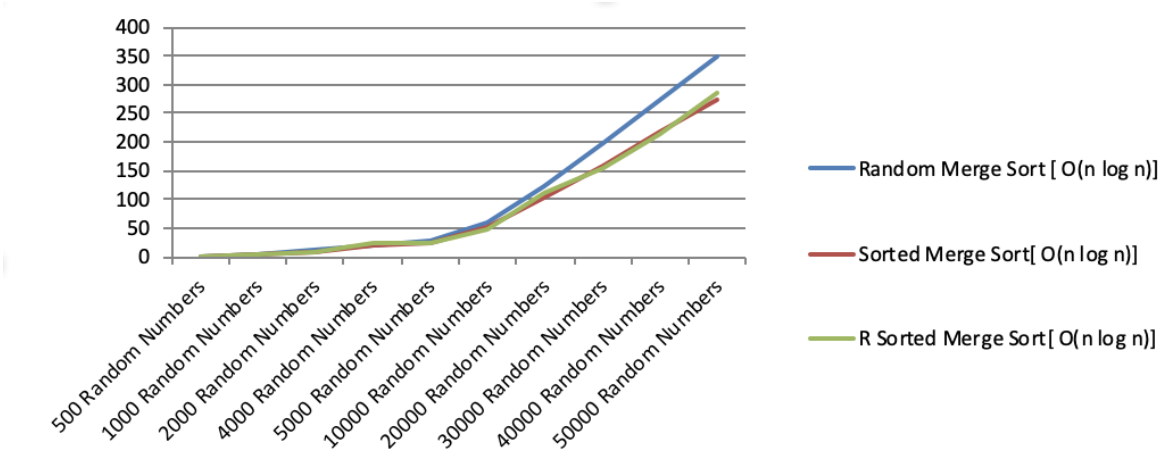
Merge Sort

In Merge Sort, we need to consider the two distinct processes that make up its implementation. First, the list is split into halves. We already computed (in a binary search) that we can divide a list in half logn times where n is the length of the list. The second process is the merge. Each item in the list will eventually be processed and placed on the sorted list. So the merge operation which results in a list of size n requires n operations. The result of this analysis is that logn splits, each of which costs n for a total of nlogn operations. In, merge sort, the best, worst, and average cases are similar.

The Θ(n log n) **best case**, **average case**, and **worst-case** complexity because the merging is always linear. Recall the basic recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \quad \Rightarrow \quad T(n) = cn \lg n$$

Therefore, merge sort is an O(nlogn) algorithm.



In-Place Quicksort

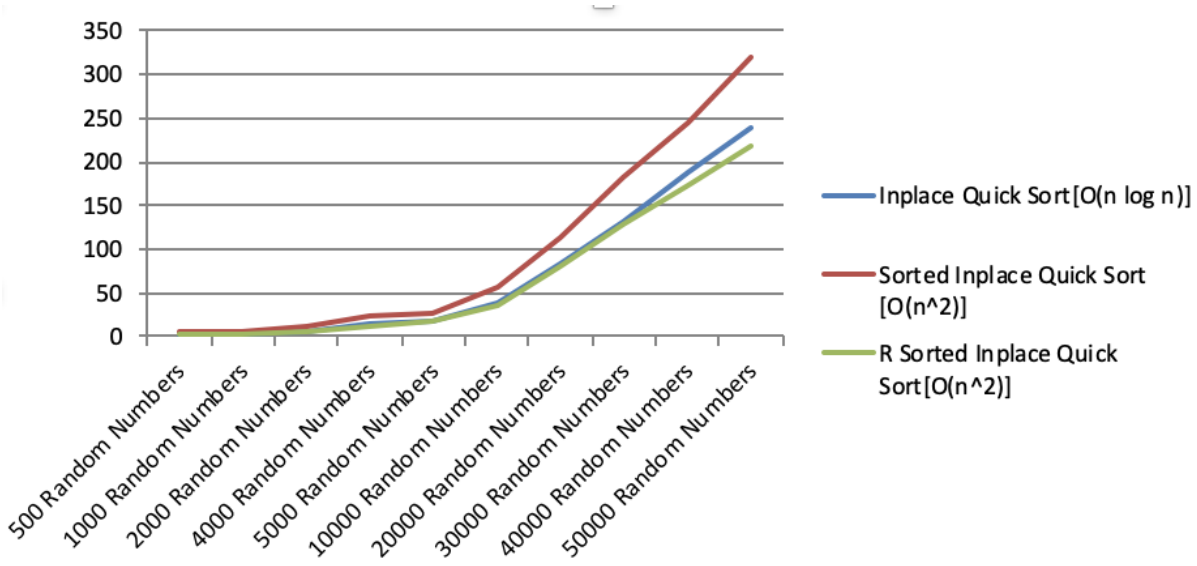
The **worst case** occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in **increasing or decreasing order**.

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and n - 1 elements in the other subarray.
  - Get the recurrence

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) . \end{aligned}$$

The **best case** occurs when the partition process always picks the middle element as pivot

In **average case,** to sort an array of n distinct elements, quicksort takes O(n log n) time in expectation, averaged over all n! permutations of n elements with equal probability.



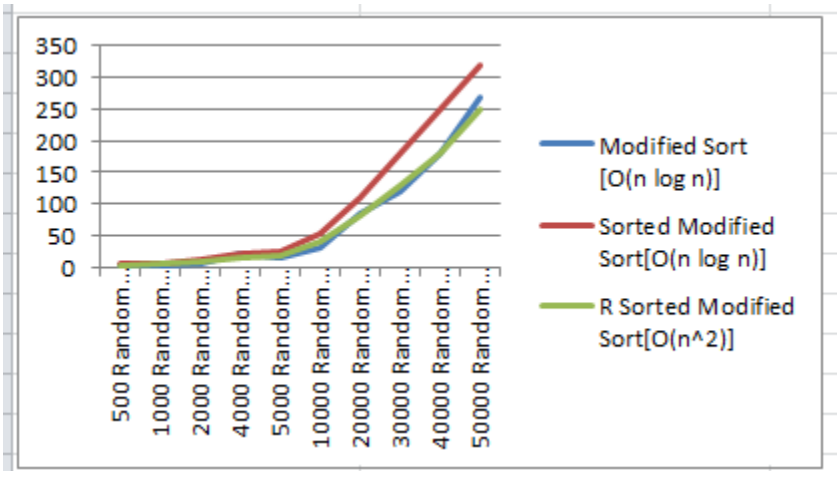
Modified Quick Sort

In Modified Quick Sort, we select the pivot using the "median of three" method instead of picking the middle element as the pivot and also when subarray or array is <=10, goes for insertion sort.

The **worst case** quick-sort happens when the pivot we picked turns out to be the least element of the array to be sorted, in every step (i.e. in every recursive call). A similar situation will also occur if the pivot happens to be the largest element of the array to be sorted.

The **best case** of quick sort occurs when the pivot we pick happens to divide the array into two exactly equal parts, in every step.

In **average case**, we assume that each of the sizes for S1 is equally likely. This assumption is valid for our pivoting (median-of-three) strategy. Therefore, On average, the running time is O(N log N).



REFERENCES

[1] Sorting algorithms, Wikipedia. Accessed: October 2018.
[2] Analysis of Sorting Algorithms, Geeks for Geeks Accessed: October 2018.